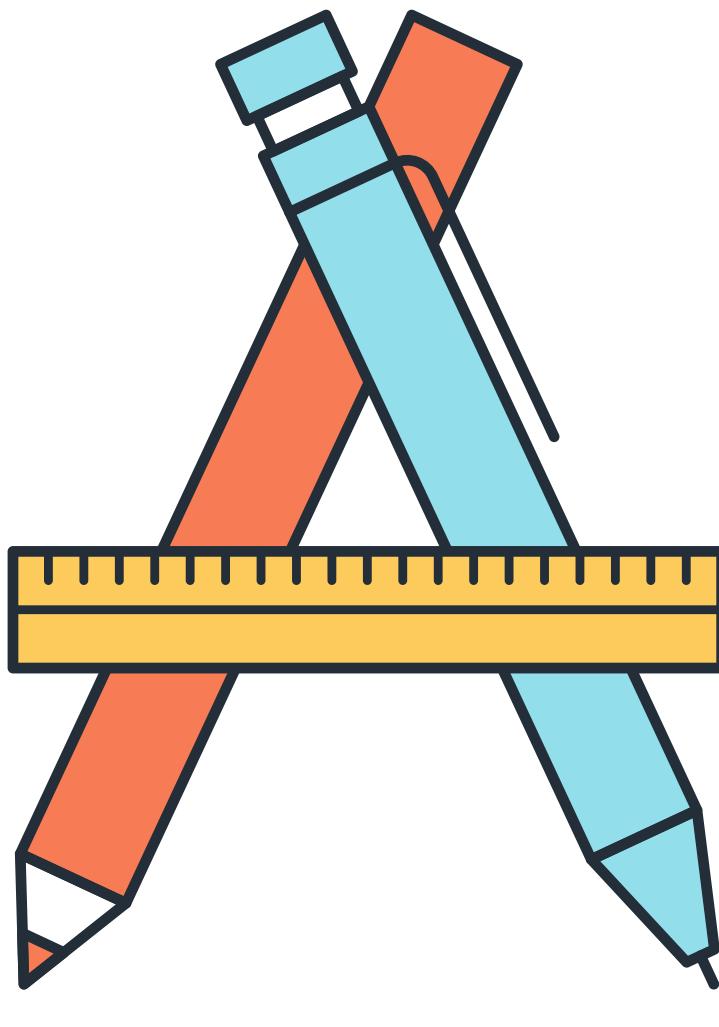


For Xcode 10, Swift 4.2 & iOS 12

MODERN AUTO LAYOUT

Building Adaptive Layouts
For iOS

Keith Harrison



Modern Auto Layout

Building Adaptive Layouts For iOS

Keith Harrison

Web: useyourloaf.com

Version: 1.0 (2018-10-01)

Copyright © 2018 Keith Harrison

Contents

1	Introduction	1
Why Learn Auto Layout?	1	
Modern Auto Layout	2	
Before We Get Started	2	
What We Will Cover	4	
Get The Code	5	
2	Layout Before Auto Layout	7
Our First Layout	8	
Autoresizing	11	
Creating A Custom Subclass Of UIView	18	
Layout Without Storyboards	25	
Key Points To remember	36	
Test Your Knowledge	37	
3	Getting Started With Auto Layout	41
What Is Auto Layout?	41	
What Is A Constraint?	42	
Who Owns A Constraint?	47	
How Many Constraints Do I Need?	50	
Test Your Knowledge	58	
4	Using Interface Builder	62
The Many Ways To Create A Constraint	62	
Editing A Constraint	68	
Creating Outlets For Constraints	70	
Viewing Layout Warnings And Errors	72	
Interface Builder Example	75	
Interface Builder Tips And Tricks	86	
Test Your Knowledge	92	
5	Creating Constraints In Code	97
Activating and Deactivating Constraints	97	

Disabling The Autoresizing Mask	99
Creating Constraints With NSLayoutConstraint	100
Visual Format Language	103
Layout Anchors	106
Which Should You Use?	111
Constraints In A Custom View	112
Key Points To Remember	115
Test Your Knowledge	116
6 Safe Areas And Layout Margins	119
Safe Area Layout Guide	120
Layout Margins	136
Layout Guides	150
Key Points To Remember	157
Test Your Knowledge	158
7 Layout Priorities And Content Size	163
Layout Priorities	163
Intrinsic Content Size	174
Content Mode	179
Content Hugging And Compression Resistance	183
Key Points To Remember	190
Test Your Knowledge	191
8 Stack Views	196
Getting Started With Stack Views	197
A Closer Look At Stack Views	205
Stack Views And Layout Priorities	215
Dynamically Updating Stack Views	222
Adding Background Views	227
Stack View Oddities	230
Key Points To Remember	235
Test Your Knowledge	236
9 Understanding The Layout Engine	241
The Layout Pass	241
Should You Use updateConstraints?	244
Animating Constraints	245
Custom Layouts	249
Alignment Rectangles	254
Key Points To Remember	259
Test Your Knowledge	259

10 Debugging When It Goes Wrong	264
Unsatisfiable Constraints	264
Adding Identifiers To Views And Constraints	270
Ambiguous Layouts	272
Using The View Debugger	273
Private Debug Methods	276
Layout Loops	280
Key Points To Remember	283
11 Scroll Views And Auto Layout	284
Creating Constraints For A Scroll View	284
Scrolling A Stack View	293
Frame And Content Layout Guides (iOS 11)	297
Managing The Keyboard	301
Key Points To Remember	306
Test Your Knowledge	306
12 Dynamic Type	310
Using Dynamic Type	310
Readable Content Guides	322
Text Views	327
Scaling Dynamic Type	330
Custom Fonts With Dynamic Type	334
Key Points To Remember	345
Test Your Knowledge	345
13 Working With Table Views	351
Self-Sizing Table View Cells	351
Readable Table Views	362
Key Points To Remember	366
Test Your Knowledge	367
14 Adapting For Size	371
Trait Collections	372
Size Classes	373
Supporting iPad Multitasking	375
Using Size Classes With Interface Builder	377
Using Traits In Code	390
Using Traits With The Asset Catalog	399
Variable Width Strings	404
When Size Classes Are Not Enough	409
Key Points To Remember	415
Test Your Knowledge	415

A Tour Of Interface Builder	422
Xcode Toolbar	423
Inspectors	423
Object Library	424
Document Outline	425
Preview Controls	426
Assistant Editor	426
Auto Layout Tools	431
B Layout Essentials	433
The View Hierarchy	433
View Geometry	438
C Points vs. Pixels	444
One More Thing	446

Chapter 1

Introduction

You may have heard Auto Layout described as a constraint-based layout engine. What does that mean? Do you need to know math and write equations? Why is that any better than manually calculating the size and position of each view in your layout?

Have you been resisting using Auto Layout? Maybe you tried it and gave up in frustration? Or maybe you're new to iOS development and wondering how to get started. Well, this book is for you.

Why Learn Auto Layout?

Apple first introduced us to Auto Layout in OS X 10.7 Lion. It took a while longer to come to iOS developers as part of iOS 6 unveiled at WWDC 2012. Auto Layout promises to make your layouts simpler to write, easier to understand, and less effort to maintain.

Using Auto Layout can feel a little abstract at first. Instead of manually setting the frame of each view you describe the relationships between your views with constraints and Auto Layout sets the frames for you. The advantage comes when your layout needs to respond and adapt to changes.

Dynamic sizing needs a dynamic layout. A modern iOS App needs to adapt to a broad set of user interface situations:

- Layouts need to scale from the smallest device like the iPhone SE up to the largest 12.9" iPad Pro and work in slide over and split screen modes.
- Text size can change significantly with localization and even more dramatically with dynamic text. Paragraphs of text that fit comfort-

ably at small text sizes can grow to where one word fills the screen at the largest of the accessibility sizes.

- You need to be able to quickly adapt when Apple introduces new devices like the iPhone X with a top sensor housing and home screen indicator.

You don't have to use Auto Layout, but many of the above challenges become manageable when you describe the relationships between your views with constraints. For example, layouts built with Auto Layout for the iPhone X in 2017 often only need rebuilding with the iOS 12 SDK to support the new screen sizes of the iPhone XR and iPhone XS Max in 2018.

Modern Auto Layout

What do I mean by "Modern Auto Layout"? A lot has changed over the years since Apple introduced Auto Layout in iOS 6. For me, Modern Auto Layout began with iOS 9:

- In iOS 9, Apple added layout anchors and layout guides. They also added stack views and using Auto Layout got a whole lot less painful.
- In iOS 10, adopting Dynamic type became less work with automatic font adjustments to content size changes.
- In iOS 11, safe area layout guides and safe area relative margins replaced top and bottom layout guides. You can change the margins of the root view. Scroll views got layout guides, and stack views got custom spacing.
- In iOS 12, Apple improved the performance of Auto Layout.

Before We Get Started

I assume you have a basic knowledge of iOS app development. You should be comfortable using Xcode to create an app and run it on the simulator or device.

This book doesn't teach you Swift or Objective-C programming or much about App architecture. I've used Swift for the code examples but don't worry if you're not an expert Swift programmer.

If you're new to Xcode and iOS development I recommended you first study an introductory tutorial such as Apple's own "Start Developing iOS Apps (Swift)" guide:

- <https://developer.apple.com/library/content/referencelibrary/GettingStarted/DevelopiOSAppsSwift/>

You may also want to read [Appendix A: Tour Of Interface Builder](#).

Interface Builder Or Code?

Get two or more iOS developers in a room and sooner or later somebody asks the question. Do you create your views using Interface Builder or in code? There are pros and cons to each approach, and you will no doubt have your own opinion.

For this book, I don't care which way you choose. I aim to teach you Auto Layout. You can learn with Interface Builder or with layouts created in code. The choice is yours but here's the approach I suggest and use in this book:

- If you're new to Auto Layout start with Interface Builder. I find it easier to play around, prototype and get a feel for the key concepts using Interface Builder.
- As soon as you feel comfortable with the basics create some layouts in code. Do this even if you prefer Interface Builder. It's a great way to test your understanding.
- As your experience grows, you'll learn for yourself what works best. Knowing both approaches has other advantages. You never know when you may find yourself working on a codebase where somebody else chose for you.
- Resist the temptation to dive into one of the many popular third-party frameworks for Auto Layout until you have mastered the basics for yourself. You may find you don't need them.

Which Versions Of Xcode, iOS, And Swift?

I wrote this edition of the book using Xcode 10, iOS 12 and Swift 4.2. The Auto Layout API's have not changed significantly in iOS 12 and most of the code should work unchanged on iOS 11. I do my best to point out changes and how to fall back for iOS 10 and iOS 9 along the way.

Finally, while the concepts and API's for Auto Layout mostly also apply to macOS I wrote this book primarily for iOS developers.

What We Will Cover

The first part of the book covers the fundamentals of Auto Layout. The key concepts that make it work and the tools to apply it to your layouts:

- [Chapter 2](#): We start by looking at how we did layout before Auto Layout by manually managing the frames of views and relying on autoresizing or when that's insufficient by overriding `layoutSubviews`. We also look at how to create an Xcode project to work without storyboards.
- [Chapter 3](#): Introduces you to constraints, what they are, who owns them and how many you need to create common layouts.
- [Chapter 4](#): We dive into using Interface Builder to create and manage constraints. We look at the many ways to create constraints and what the warnings and errors mean. We also have lots of useful tips and tricks to help you master Interface Builder.
- [Chapter 5](#): You don't have to use Interface Builder to use Auto Layout. In this chapter, we look at the three ways Apple gives us to create constraints in code and why you want to use layout anchors.
- [Chapter 6](#): Safe Area Layout Guides became a hot topic when Apple launched the iPhone X. In this chapter, we look at how to use them to keep the system from clipping or covering your content and how to fallback to the older top and bottom layout guides for iOS 9 and 10. We also look at using layout margins for extra padding and layout guides as an alternative to spacer views.
- [Chapter 7](#): We cover the topics that cause many people to hate Auto Layout. The tricky concepts of layout priorities, intrinsic content size, and content-hugging and compression-resistance.
- [Chapter 8](#): Stack views were a welcome and overdue addition in iOS 9. Build layouts without having to manually create every constraint in Interface Builder or with pages of boilerplate code. We also cover some useful improvements that came in iOS 11 and some oddities to avoid.
- [Chapter 9](#): Time to dig deep into how the layout engine works to translate your constraints into a working layout. Why you probably shouldn't be using `updateConstraints`, how to animate changes to constraints and how to override `layoutSubviews` to take control of the layout.
- [Chapter 10](#): How do you debug your layouts when they go wrong? We look at the tools and techniques to understand and fix your Auto

Layout problems.

With the foundation built the second part of the book looks at how to use Auto Layout with related API's to build adaptive layouts.

- [Chapter 11](#): The scroll view is an essential view to master when building layouts with content too big for the available space. We use it frequently in later chapters to build more adaptive layouts. It improved in iOS 11, but it can still be confusing to use with Auto Layout. In this chapter, you learn how to create your constraints when adding content to a scroll view.
- [Chapter 12](#): Dynamic type puts the user in control of the size of text in your App. In this chapter, we learn how to use dynamic type and adapt our layouts to cope with dramatic changes in text size. We also see how to use custom fonts with Dynamic type.
- [Chapter 13](#): Self-sizing table view cells are a regular source of pain and confusion. In this chapter, we learn how they work and how to use readable content guides with table views.
- [Chapter 14](#): In the final chapter we bring everything together to look at how to build layouts that adapt to the size of the screen. Learn how to use trait collections and size classes, create asset variations and variable width strings. Finally, we go beyond size classes to build adaptive layouts based on the available space.

Can I offer some extra words of advice? It's hard to learn Auto Layout just by reading about it. You need to use it. I suggest you set aside time for deliberate, focused learning. Read a chapter or a section of the book and then apply the knowledge. The challenges at the end of each chapter get you started, but you also need to practice for yourself.

Get The Code

You can download the sample code used in this book together with the solutions to the challenges from my GitHub repository for the book:

- <https://github.com/kharrison/albookcode>

Xcode Project And File Templates

In this book when I create a layout using Interface Builder, I start from the Xcode single view app iOS template. For programmatic layouts, I remove the storyboard from that template. I describe the steps to do that

in the next chapter but if you prefer you can use my already customized templates.

You can find full instructions and download the templates from my GitHub repository:

- <https://github.com/kharrison/Xcode-Templates>

Chapter 2

Layout Before Auto Layout

Let's go back in time to early 2012. If you were an iOS developer, you were using Xcode 4 to develop apps for iOS 5. Apple didn't bring Auto Layout to iOS until later that year so how did we manage to create the user interfaces for our Apps?

In this chapter, we look at how we used to do layout before Auto Layout. Why take a trip down memory lane? Auto Layout builds on top of these basic view management and layout systems. Learning these foundations now helps us understand what Auto Layout is doing under the covers. Auto Layout is also not perfect - what is? There are times when you may choose to drop down from Auto Layout to manage the size and position of your views directly.

Topics you learn in this chapter:

- How to manually layout views with Interface Builder.
- How to automatically resize views with autoresizing masks and the limitations of autoresizing.
- How to create custom views and override `layoutSubviews`.
- How to create a layout in code without storyboards.

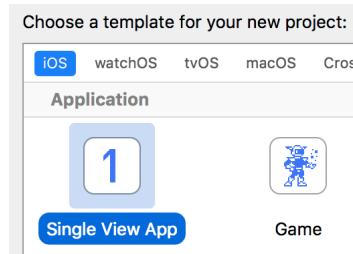


If you're new to iOS development or not sure you ever fully understood how iOS manages views take a few minutes to review [Appendix B: Layout Essentials](#).

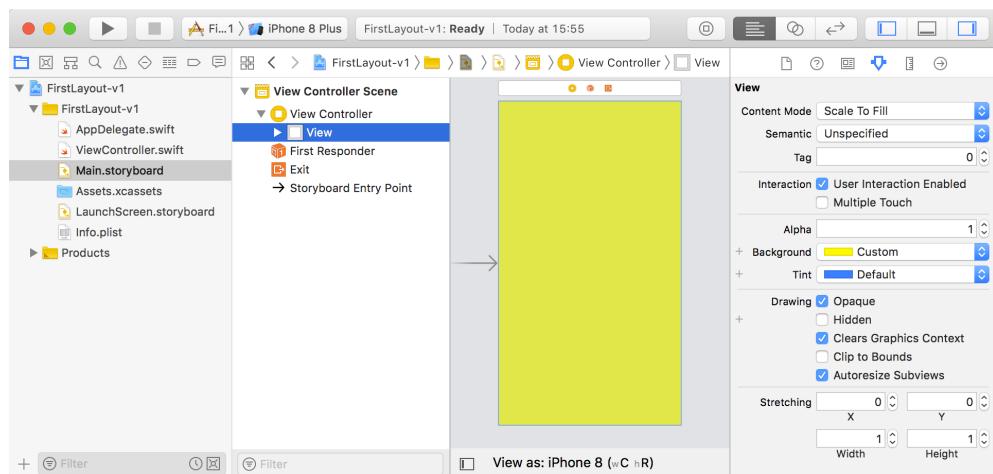
Our First Layout

Open Xcode and follow along (see sample code: [FirstLayout-v1](#)):

1. Create a new Xcode project (File > New > Project...), and choose the Single View App iOS application template:

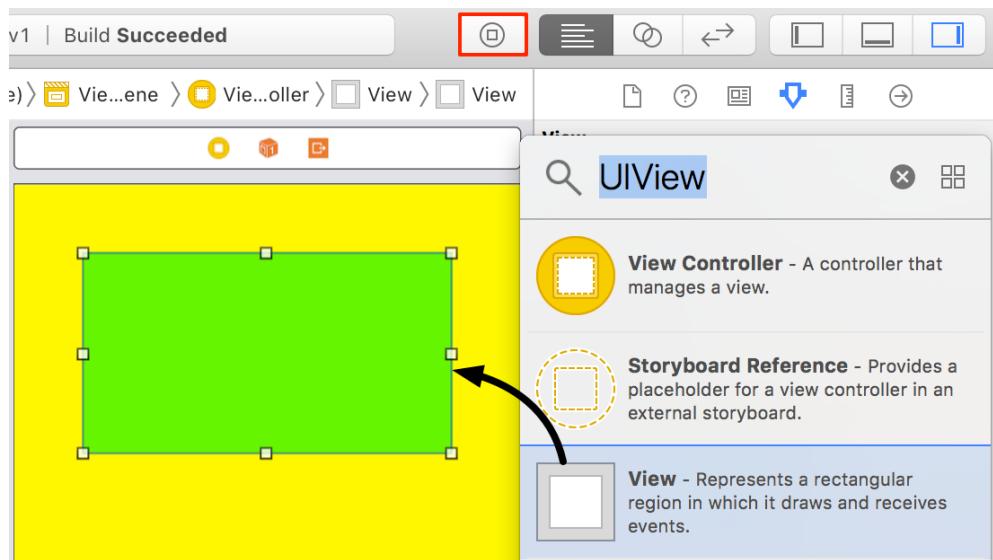


2. Name your project and save it. Find the `Main.storyboard` in the Xcode navigator and click on it. The Xcode editor area switches to Interface Builder and shows us the single view controller scene.

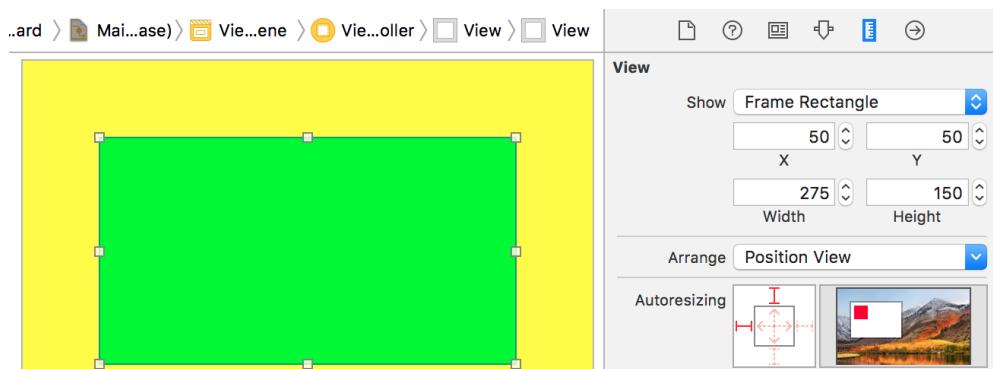


I changed the color of the root view to yellow to make it easier to see.

3. Open the Object Library using the button in the toolbar and drag a plain old view onto the yellow root view in the Interface Builder canvas. Use the attributes inspector to give it a green background color:



4. Drag the view around to set its size and position or select the view and use the size inspector to change the frame rectangle:



To keep my green view away from the edges I gave it 50 points of space to the leading, top and trailing edges. We'll see better ways of doing this when we look at [Safe Area Layout Margins](#).

5. I set Interface Builder to show me the layout on an iPhone 8 which is 375 points wide. So with an origin of (x=50, y=50), the width of the green view is 275 points. Let's also make it 150 points high.



When working with layouts we nearly always use points and not pixels. To understand how iOS maps points to the physical pixel sizes of each device see [Appendix C: Points vs. Pixels](#).

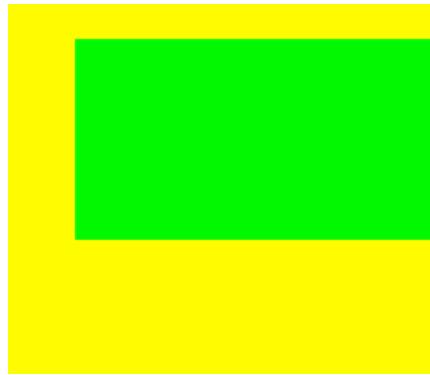
6. Build our fantastic app and run it on the iPhone 8 simulator. The layout matches our setup in Interface Builder which is a good start (full height not shown):



7. The result when we rotate the device is disappointing. I would like my green view to resize with the device to keep the 50 point spacing to the screen edges. Instead, it sticks to the initial size and position we set up in Interface Builder:



8. On the small screen of the iPhone SE, which is only 320 points in portrait, the result is even worse. The right side of my green view disappears off-screen:



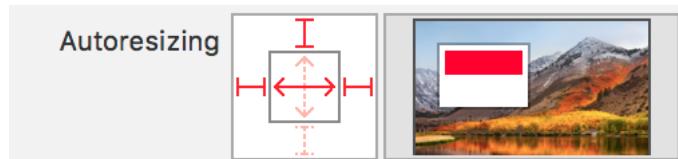
Luckily there are ways to control how our green view resizes when its yellow superview changes size.

Autoresizing

When a view's size changes it will, by default, resize each of its subviews. The way it resizes depends on each subview's autoresizing mask.

Springs And Struts

The autoresizing mechanism is often referred to as **springs and struts** as that's how Interface Builder shows the mask in the size inspector next to an animation that simulates the result:

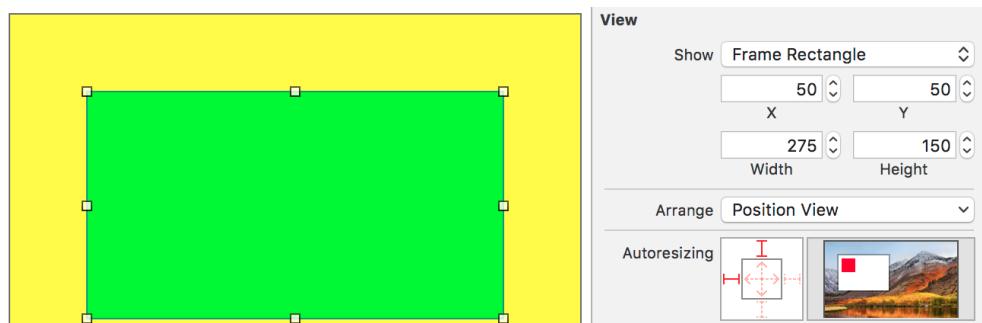


The **external struts** fix the space between each edge of the subview and its superview. If you don't select a strut, the spacing on that edge is flexible and can change when the superview resizes.

The **internal springs** give the subview a flexible width and height that resizes with the superview. If you don't select a spring, it fixes the subview size for that dimension.

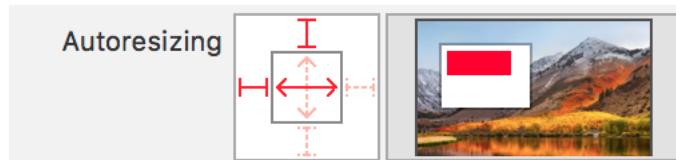
We can use springs and struts to make our green view resize with its superview (see sample code: [FirstLayout-v2](#)):

1. Open the Xcode project we created at the start of this chapter.
2. Select the green view in Interface Builder so that it shows up in the size inspector:

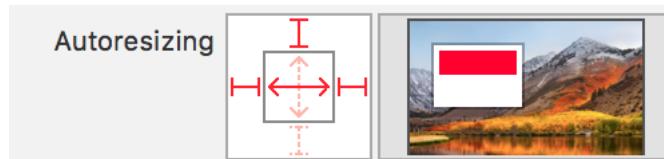


If you look carefully at the autoresizing section in the inspector, you can see that Interface Builder sets the left and top struts by default fixing the green view to the top left corner.

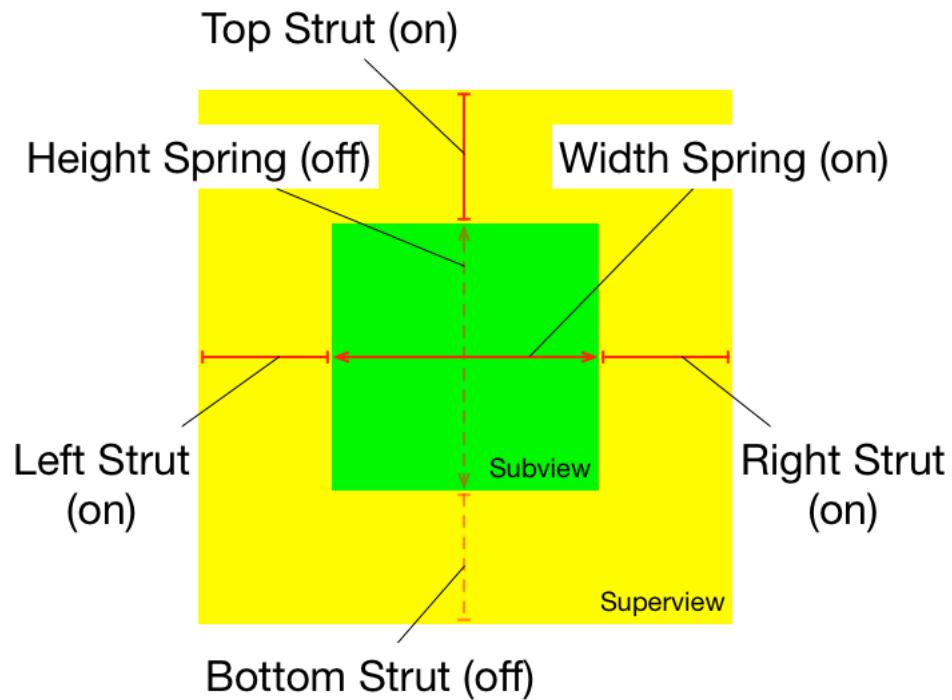
3. To make the width of the green view flexible select the internal horizontal width spring.



4. To fix the space from the green view to the right edge of the superview select the right strut:



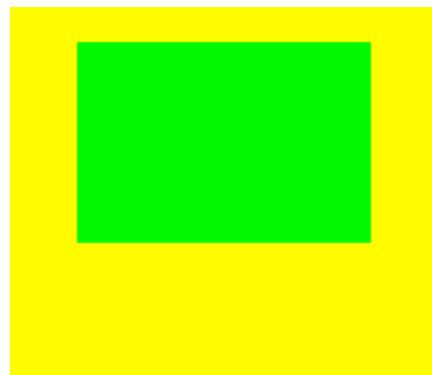
5. You should end up with the left, top and right external struts and the internal width spring selected:



6. Build and run again on the iPhone 8 simulator. When you rotate the device to landscape you should see the green view width increase to maintain the 50 point margin on the right:

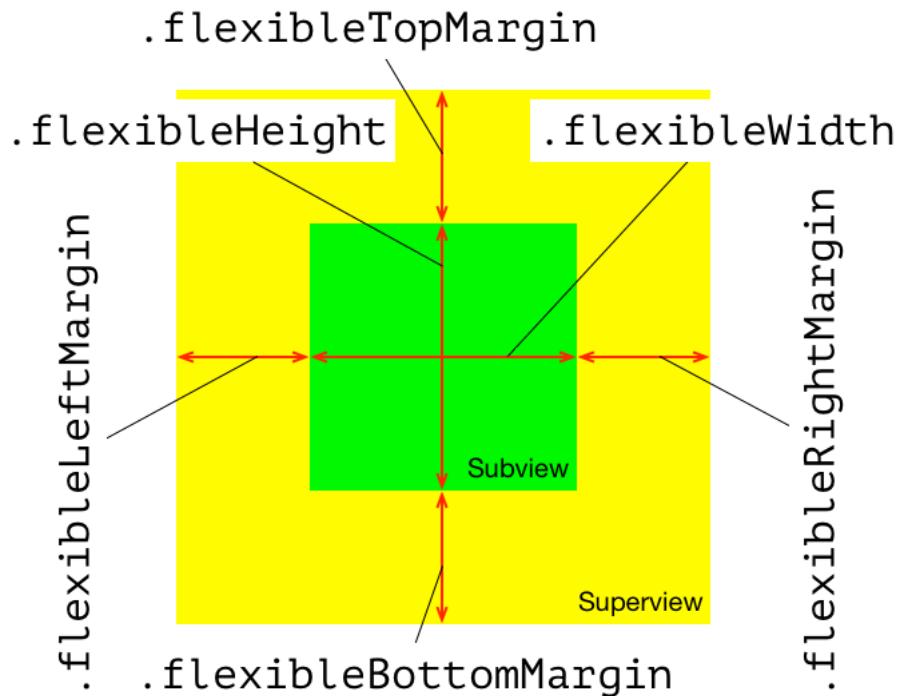


7. Try it on the iPhone SE and check that the green view now fits in the reduced width:



Autoresizing Mask

When you use Interface Builder to set springs and struts, you're changing that view's autoresizingMask. Each bit in the mask corresponds to one of the view's external margins or internal width and height:



When setting the mask in code Swift uses an option set. For example, a mask with flexible width and height:

```
myView.autoresizingMask = [.flexibleWidth,.flexibleHeight]
```

Comparing the autoresizing mask to springs and struts in Interface Builder can be confusing. Setting the width spring in Interface Builder is the same as including the .flexibleWidth in the mask. Setting the

top strut in Interface Builder fixes the margin, the opposite of including `.flexibleTopMargin` in the mask.

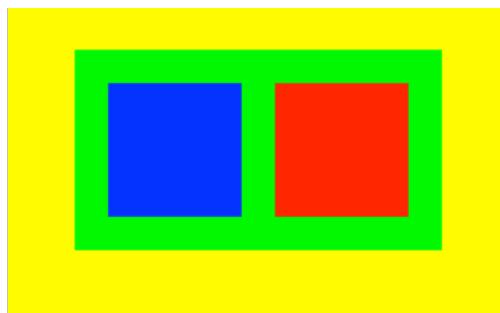
When creating views in code the default mask value is `.none` (no resizing). Interface Builder sets the top and left struts by default which is equivalent to an autoresizing mask with flexible bottom and right margins.

If you look back at the springs and struts we set for the green view can you guess what the resulting autoresizing mask was? We set the left, top and right struts and the width spring in Interface Builder. This combination gives us a flexible width and a flexible bottom margin. To create this autoresizing mask in code, we would write:

```
greenView.autoresizingMask = [.flexibleWidth,  
    .flexibleBottomMargin]
```

The Limits Of Autoresizing Masks

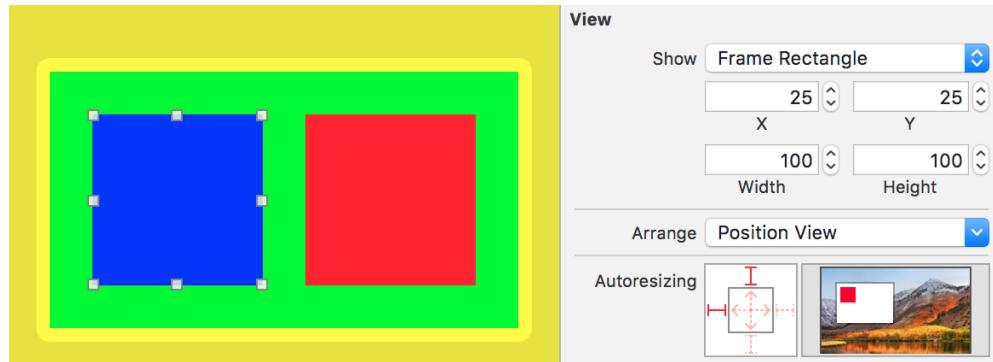
The autoresizing mask works well, but sometimes it's not enough. Let's add two new subviews to our single-view layout. Here's the layout I want to create:



- I set the Interface Builder canvas to use the iPhone 8 which has a width of 375 points in portrait.
- The green view has margins of 50 points to the screen edges which makes it 275 points wide. It has a fixed height of 150 points.
- I want the blue and red subviews to have a margin of 25 points on each side within the green view. That margin makes them 100 points high and 100 points wide in portrait on the iPhone 8.
- When the width of the green view changes I want my blue and red subviews to resize so that they keep a 25 point margin on all sides (they will not always be square).

Continuing our Xcode project after we added the springs and struts to the green view (see sample code: [FirstLayout-v3](#)):

1. Drag two plain views onto the canvas and drop them on the green view. The views become subviews of the green view. Use the attributes inspector to change the background colors of the two views. I made one blue and the other red.
2. Select the blue view and use the size inspector to set its origin to (25,25) and size to 100×100 :



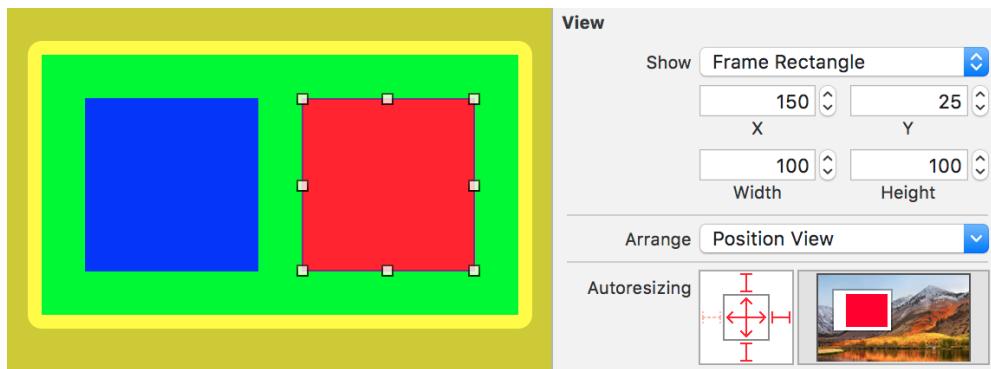
Do the same for the red view using an origin of (150,25). Note that the coordinates for the origins of the red and blue views are relative to the green view.

3. How should we set the springs and struts for the blue view? For sure I want the struts for the top and left margins to fix the 25 points of spacing to the edge of the green view. I also want the width of the blue view to shrink and stretch with the green view, so it needs to have a flexible width. Also add the bottom strut and flexible height spring to make sure we keep the bottom margin fixed:

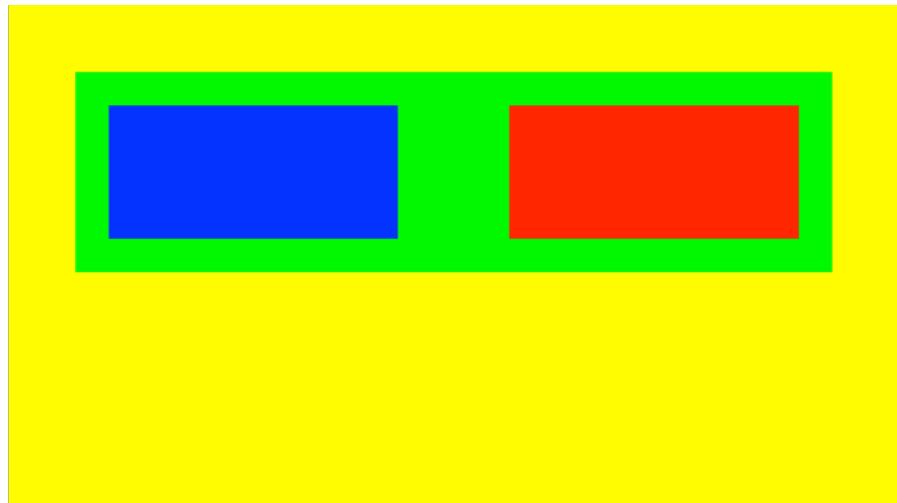


What about the right margin? Adding the right strut fixes the space to the right edge of the green view but I want that space to be flexible.

4. We can repeat this process for the red view. For the resizing, we can use top, right and bottom struts with flexible width and height springs but we don't want the left strut:



5. If we run on an iPhone 8, it produces this layout in landscape. The width of the blue and red views has increased but so has the spacing between them.



I want to keep the 25 point space between the blue and red subviews, but that's not possible with just springs and struts.

The problem is that the autoresizing mask limits you to a parent-child relationship from a superview to its subview. We can only have struts from the green superview to either the blue or red subviews. What we needed and cannot have is a strut to fix the spacing between the blue and red subviews.

To overcome this limitation of autoresizing we need to manually intervene when the green view is laying out its subviews. Before we do that, let's learn how to create custom views.

Creating A Custom Subclass Of UIView

When UIKit needs to update the layout of your user interface it works its way from top to bottom of the view hierarchy. For each view, it first applies the autoresizing mask for each of the subviews and then calls the `layoutSubviews` method of the view itself. If we create a subclass of `UIView`, we can override `layoutSubviews` and manually adjust the frames of our views after the autoresizing action has happened.

Let's go back to our last example and see how this helps. We need to fix the frames of the blue and red subviews. We can do that by creating a custom subclass of `UIView` for the green superview. The code snippet below gives us a starting point:



I include a file template for creating a custom view in my Xcode templates. See [Xcode Project And File Templates](#) for details.

```
// TileView.swift
import UIKit

class TileView: UIView {
    override init(frame: CGRect) {
        super.init(frame: frame)
        setupView()
    }

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
        setupView()
    }

    private func setupView() {
        // add view setup code here
    }

    override func layoutSubviews() {
        super.layoutSubviews()
        // Adjust subviews here
    }
}
```

I named the class `TileView`. It's a subclass of `UIView`. Note that it has two initializers:

- `init(frame: CGRect)` called when we create the view in code.
- `init?(coder: NSCoder)` called when loading the view from a nib file or storyboard.

To allow the view to work with storyboards or programmatic layouts, I keep the common view setup code in a separate private method that I call from both initializers.

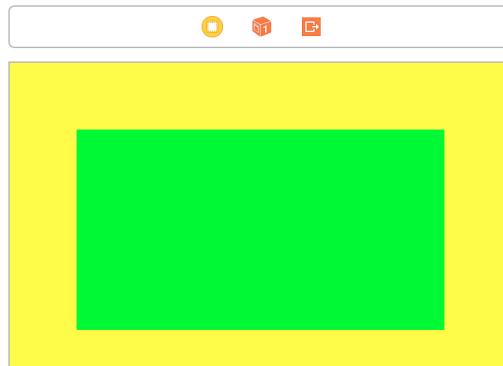
The `init?(coder: NSCoder)` method is a required initializer for `UIView`. Xcode reports an error if you don't include it even if you never use the view with a nib or storyboard.

It's common to see developers, who are not using nibs or storyboards, avoid this error by having the initializer create a fatal error. Xcode offers to add it for you if you forget:

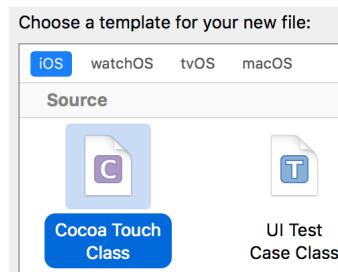
```
required init?(coder aDecoder: NSCoder) {
    fatalError("init(coder:) has not been implemented")
}
```

To fix our layout we can replace the green view in our Xcode project with a custom view (see sample code: [FirstLayout-v4](#)):

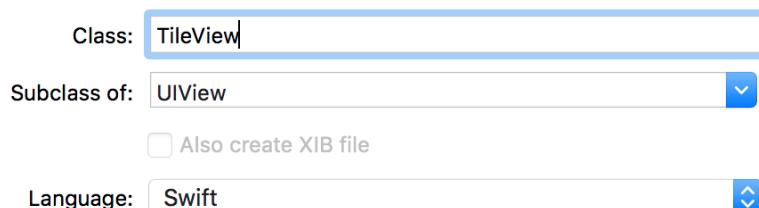
1. Delete the red and blue views from Interface Builder leaving just the green view:



2. Add a new file to the project (File > New > File...) and choose the iOS Cocoa Touch Class file template:



3. Name the file `TileView`, make sure it's a subclass of `UIView` and then create it in the same folder as the `ViewController.swift` file:



4. Find the new `TileView.swift` file in the project navigator and then paste our custom view code snippet into the body of the class:

```

1 import UIKit
2
3 class TileView: UIView {
4
5     override init(frame: CGRect) {
6         super.init(frame: frame)
7         setupView()
8     }
9
10    required init?(coder aDecoder: NSCoder) {
11        super.init(coder: aDecoder)
12        setupView()
13    }
14
15    private func setupView() {
16        // add view setup code here
17    }
18
19    override func layoutSubviews() {
20        super.layoutSubviews()
21        // Adjust subviews here
22    }
23 }

```

5. We're no longer using Interface Builder to create the red and blue subviews. We need to create them in `TileView`. Add `redView` and `blueView` private properties to the view. These both return a plain

`UIView` with the background color set:

```
private let redView: UIView = {
    let view = UIView()
    view.backgroundColor = .red
    return view
}()

private let blueView: UIView = {
    let view = UIView()
    view.backgroundColor = .blue
    return view
}()
```

6. In our common setup method add both views as subviews of the tile view:

```
private func setupView() {
    addSubview(blueView)
    addSubview(redView)
}
```

7. Add a property for the amount of padding with a default value of 25 points:

```
var padding: CGFloat = 25.0 {
    didSet {
        setNeedsLayout()
    }
}
```

Should the user of our view change the padding, we need to update the layout. You never call `layoutSubviews` yourself. Instead we tell `UIKit` our layout needs updating with `setNeedsLayout()` and it calls our `layoutSubviews` during the next update cycle.

8. In `layoutSubviews` set the position and size of the two subviews allowing for the padding:

```
// Size of this container view
let containerWidth = bounds.width
let containerHeight = bounds.height

// Calculate width and height of each item
// including the padding
let numberofItems: CGFloat = 2
```

```

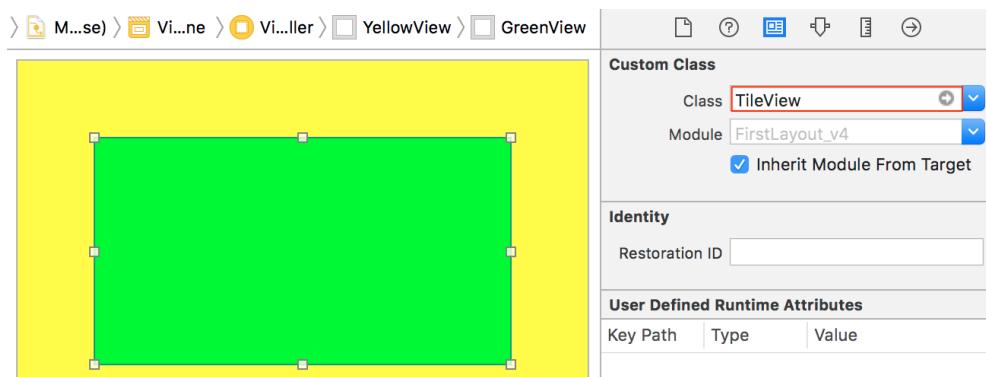
let itemWidth = (containerWidth - (numberOfItems + 1) * padding) / numberOfItems
let itemHeight = containerHeight - 2 * padding

// Set the frames of the two subviews
blueView.frame = CGRect(x: padding, y: padding, width: itemWidth, height: itemHeight)
redView.frame = CGRect(x: 2 * padding + itemWidth, y: padding, width: itemWidth, height: itemHeight)

```

If you need a refresh on how to work with view bounds and frames see [Appendix B: Layout Essentials](#).

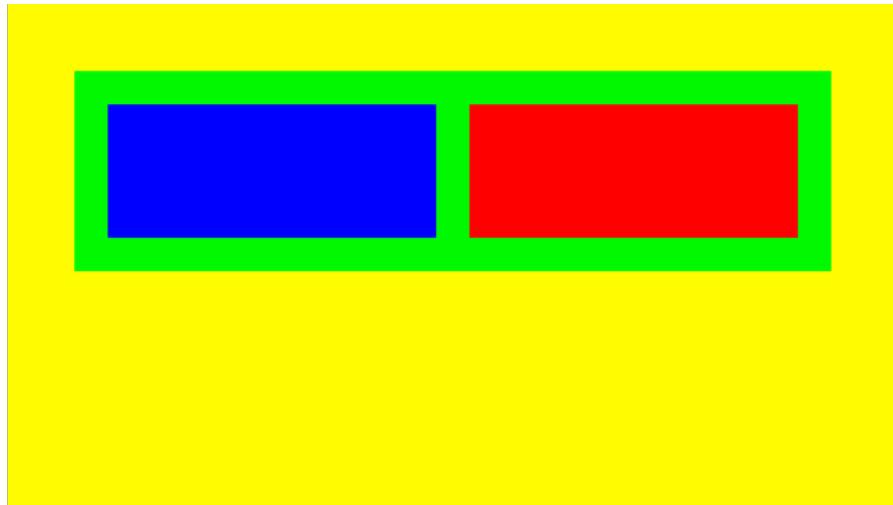
9. We still need the green view in Interface Builder, but it's no longer a plain `UIView`. Use the identity inspector to change the class of the view to our custom `TileView` class:



10. We've not changed the frame or the resizing of the green view. Check with the size inspector that the frame of the green view is still (x: 50, y: 50, w: 275, h: 150) as before. The left, top, and right struts and flexible width spring should all be on.



11. Build and run. The layout should now resize the blue and red subviews keeping the 25 point padding between the views in landscape:



Designable And Inspectable Custom Views

One disadvantage of our custom view is that we no longer see the design of our blue and red views in Interface Builder. Xcode supports live previewing of “designable” custom views in Interface Builder by adding the keyword `@IBDesignable` before the class definition. You can also make properties of the view editable in Interface Builder by adding the keyword `@IBInspectable`.

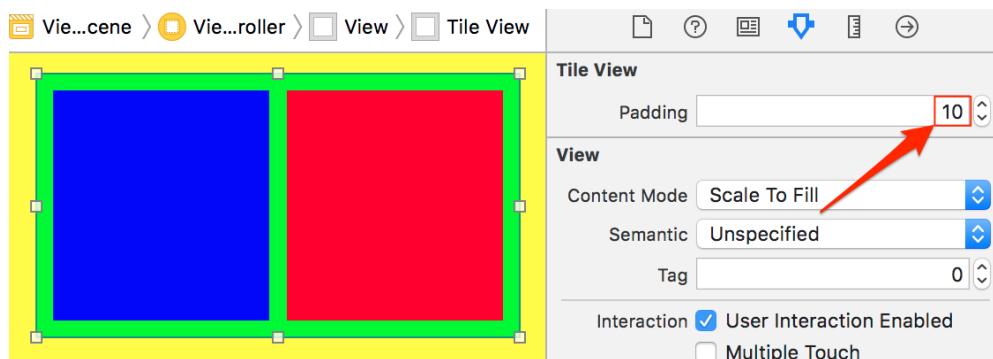
1. For bonus points, let’s make our custom view class designable so we can play with it in Interface Builder (see sample code: [FirstLayout-v5](#)):

```
@IBDesignable  
class TileView: UIView {
```

2. We can also make the padding parameter inspectable so we can change it in Interface Builder:

```
@IBInspectable var padding: CGFloat = 25.0 {
```

3. The red and blue subviews should now show up in Interface Builder. Try changing the padding in the inspector:



Using The View Controller To Layout Subviews

If you have a simple layout, it's possible to avoid creating a custom `UIView` subclass just to override `layoutSubviews`. The `UIViewController` class has two methods you can use to adjust layout:

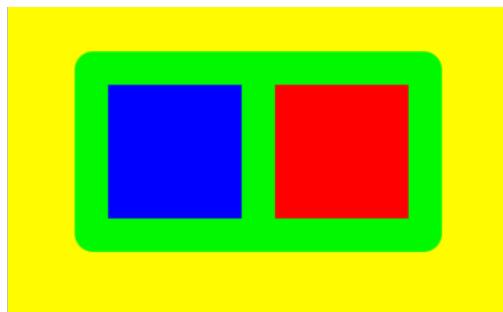
- `viewWillLayoutSubviews`: called before the view controller's view starts to layout its subviews.
- `viewDidLayoutSubviews`: called after the view controller's view has finished layout of its subviews.

The default implementation of both these methods does nothing. As with `layoutSubviews`, the system can call these methods many times during the life of a view controller so avoid doing unnecessary work in them. For any significant amount of layout, I prefer to create a custom view, but you might use `viewDidLayoutSubviews` to make a small change to a view after the view controller has finished its layout.

For example, to add a corner radius to our tile view that's a percentage of the view width (see sample code: [FirstLayout-v6](#)):

```
override func viewDidLayoutSubviews() {
    super.viewDidLayoutSubviews()

    // 5% radius
    let radius = tileView.bounds.width / 20
    tileView.layer.cornerRadius = radius
}
```



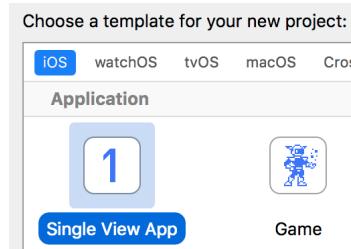
Layout Without Storyboards

The default Xcode iOS application templates assume you use storyboards to build your layout, but it's not the only way. You're free to create your layout entirely in code if you wish. Let's build a layout in code without a storyboard.

Removing The Main Storyboard

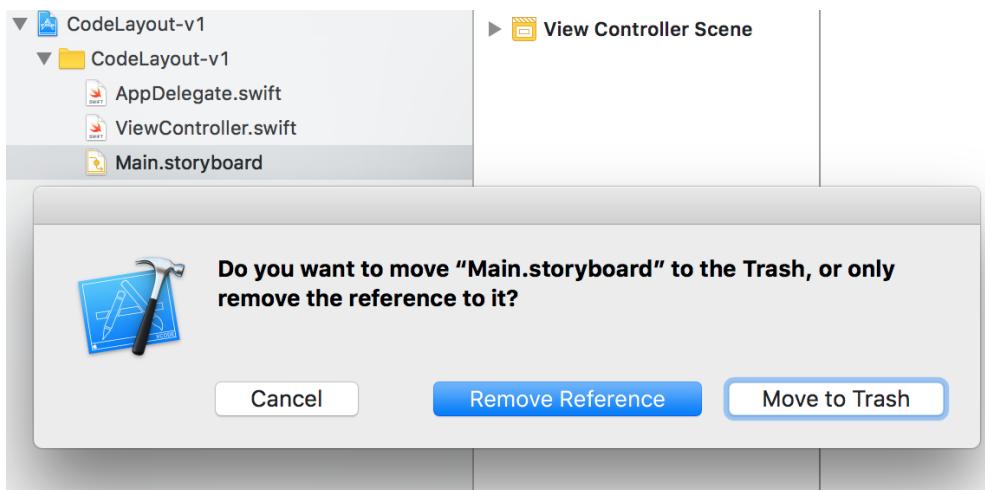
Start by cleaning up the Xcode single view app template to remove all traces of the main storyboard (see sample code: [CodeLayout-v1](#)):

1. Create a new Xcode project (File > New > Project...), and choose the Single View App iOS application template:



Name your project and save it in a folder of your choosing.

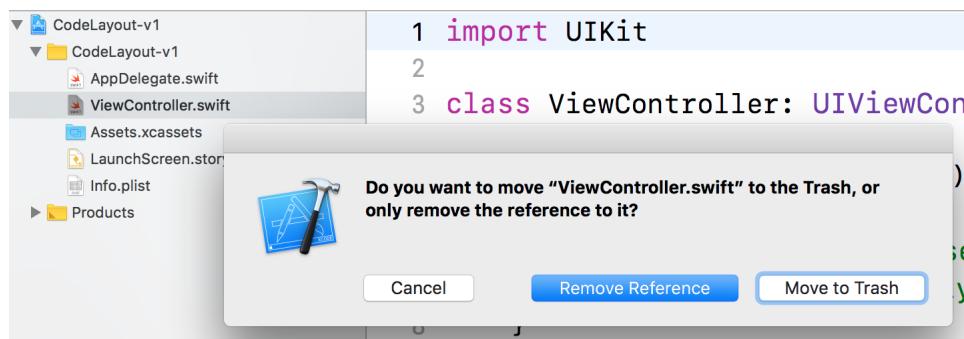
2. Find the `Main.storyboard` in the Xcode navigator and delete it. Select "Move to Trash" to delete the file:



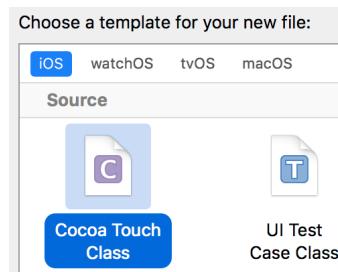
3. To stop the app from trying to load the main storyboard when it launches, we also need to delete the `UIMainStoryboardFile` key from its `Info.plist` file. Find the entry labeled “Main storyboard file base name” in the `Info.plist` file and delete it:

Application requires iPhone environment	Boolean	YES
Launch screen interface file base name	String	LaunchScreen
Main storyboard file base name	String	Main
► Required device capabilities	Array	(1 item)
► Supported interface orientations	Array	(3 items)

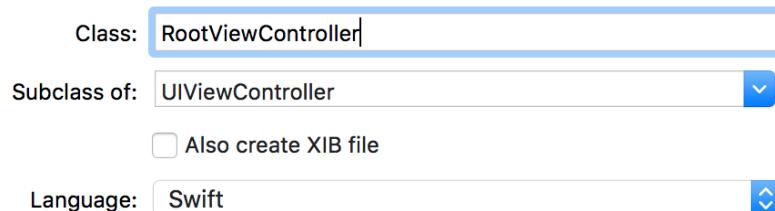
4. While we’re cleaning up the Xcode template, let’s also delete the `ViewController.swift` file. I prefer to use more meaningful class names. Make sure to select “Move to Trash” to both remove the project reference and delete the file.



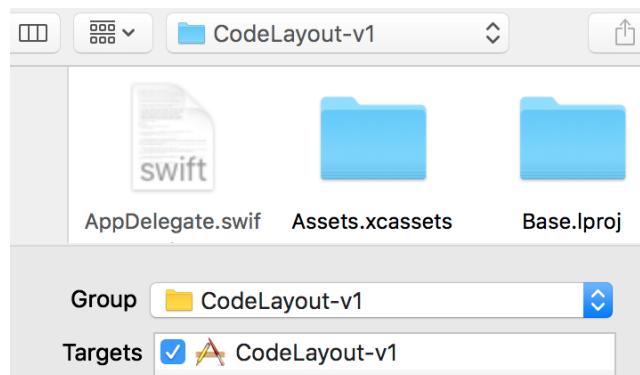
5. Add a new file to the project (File > New > File...) and choose the “Cocoa Touch Class” file template:



I named the class `RootViewController`. Make sure it's a subclass of `UIViewController`:



Save the file in the same folder and group as the `AppDelegate.swift` file:



- Without the main storyboard, we need to take care of creating the main window. Find the `AppDelegate.swift` file and delete the template comments leaving the `didFinishLaunchingWithOptions` method:

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?
```

```
func application(_ application: UIApplication,  
    didFinishLaunchingWithOptions launchOptions:  
    [UIApplication.LaunchOptionsKey: Any]?) -> Bool {  
    // Add code here...  
    return true  
}
```

7. In the body of the method create the main window using the size of the main screen. (By default a new window uses the main screen of the device):

```
window = UIWindow(frame: UIScreen.main.bounds)
```

8. Optionally set the color of the window (default is black).

```
window?.backgroundColor = .white
```

9. Create your root view controller and add it to the window. This action takes care of adding the view controller's view to the window.

```
window?.rootViewController = RootViewController()
```

Replace `RootViewController` with the class name of your view controller if you named it differently.

10. To show the window set it as the key window and make it visible:

```
window?.makeKeyAndVisible()
```

11. Here's the complete method when done:

```
func application(_ application: UIApplication,  
    didFinishLaunchingWithOptions launchOptions:  
    [UIApplication.LaunchOptionsKey: Any]?) -> Bool {  
    window = UIWindow(frame: UIScreen.main.bounds)  
    window?.backgroundColor = .white  
    window?.rootViewController = RootViewController()  
    window?.makeKeyAndVisible()  
    return true  
}
```

12. Build and check the app runs. A common mistake is to forget to remove the main storyboard entry from `Info.plist` which causes your app to crash on launch. You should see a white screen (or

whatever color you chose for the window).

Setting the root view controller of the window in the app delegate adds the view belonging to our root view controller to the window. So why do we see the window? Since we did nothing to set up the view it's transparent by default, so we see through it to the window beneath.

At this point, we have a working Xcode template for storyboard free layouts, but we still need to set up the view controller's view. Before we do that, let's take a look at how a view controller loads its view.

How View Controllers Load Their View

A view controller stores its root view in its `view` property. A newly created view controller doesn't load its view right away, so the `view` property is `nil` by default. If you access the `view` when it's `nil` the view controller calls the aptly named `loadView()` method to load the view. This "lazy loading" of the view means it's only loaded when needed (typically because it's about to come on screen).

It doesn't matter how you create your view controller the `loadView()` method always gets called to load the view. If the `loadView()` method finds a nib or storyboard file, it loads the view and any subviews from it. If there's no file, it creates a plain `UIView`.



Don't call `loadView` yourself. If you want to force the loading of the view call `loadViewIfNeeded()`. To test if a view controller has loaded its view, without triggering `loadView()` to load it, use `isViewLoaded`.

How does `loadView()` find a nib file for a view controller? It first checks the `nibName` property of the view controller. If you create your view controller with a storyboard UIKit sets the `nibName` for you using a nib file stored in the storyboard.

If you're creating your view controller in code, you must eventually call the designated initializer `init(nibName:bundle:)` to specify the nib file name and bundle to use. Some examples:

```
// Load from RootViewController.xib in main bundle
let controller = RootViewController(nibName:
    "RootViewController", bundle: Bundle.main)
```

```
// Default to main bundle
let controller = RootViewController(nibName:
    "RootViewController", bundle: nil)
```

If the view controller class and nib file are in a framework bundle:

```
let controller = RootViewController(nibName:
    "RootViewController", bundle: Bundle(for:
    RootViewController.self))
```

If you don't set the `nibName`, `loadView()` searches for a file using the name of the view controller class. For example, if our view controller class is `RootViewController` it tries in this order:

- `RootView.nib` - this only works for classes that end in `Controller`.
- `RootViewController.nib`

You can also use platform-specific nib files:

- `RootViewController~ipad.nib` - iPad specific nib file
- `RootViewController~iphone.nib` - iPhone specific nib file

Note that using the default `UIViewController` initializer is the same as calling the designated initializer with both `nibName` and `bundle` as `nil`:

```
// The following are equivalent
let controller = RootViewController(nibName: nil, bundle: nil)
let controller = RootViewController()
```

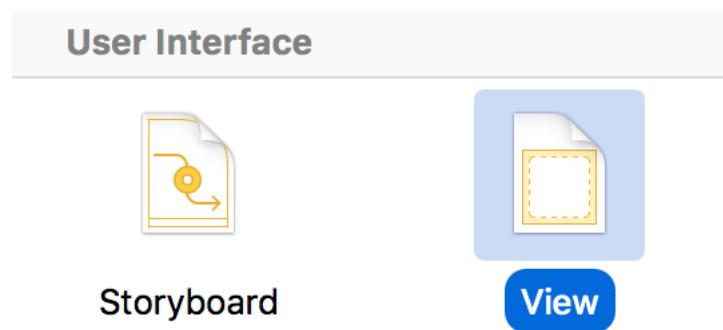
For view controllers created in a storyboard the view is loaded from the storyboard when you instantiate it:

```
if let vc = storyboard?.instantiateViewController(
   (withIdentifier: "MyViewController")) {
    // setup and present
}
```

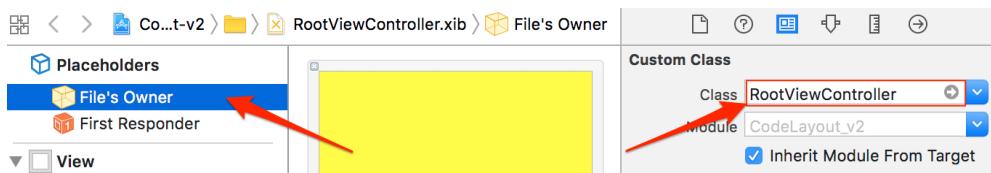
Using A Nib File

If you're not creating your view controller with a storyboard, you can still use Interface Builder to create its views in a standalone nib file. Let's add a nib file to our project and create a view layout for our view controller (see sample code: [CodeLayout-v2](#)):

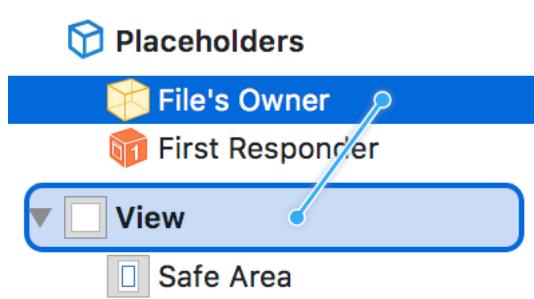
1. Add a new file to the project (File > New > File...) and choose the View template from the User Interface section of the template browser:



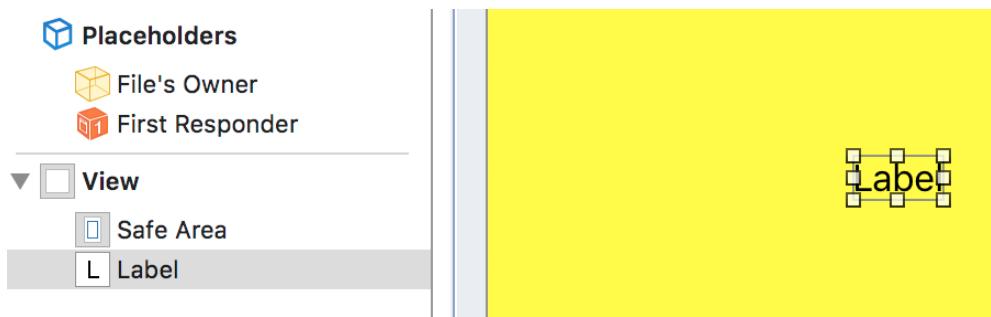
2. Name the file using the same base name as your view controller, for example `RootViewController.xib`, and save it in the project folder. Xcode adds the `.xib` extension for you.
3. Select the “File’s Owner” placeholder in the document outline to the left of the canvas. Using the Identity Inspector set the class to the class of the view controller (`RootViewController` in my case):



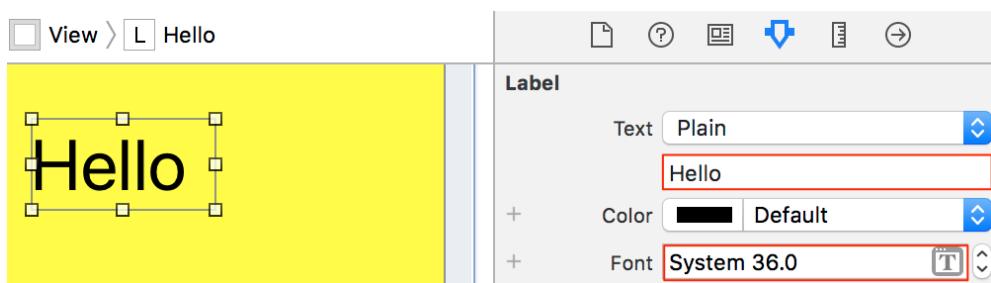
4. Connect the view in the nib file to the `view` property of the view controller. Control-drag from the File’s Owner placeholder to the view and select the `view` outlet:



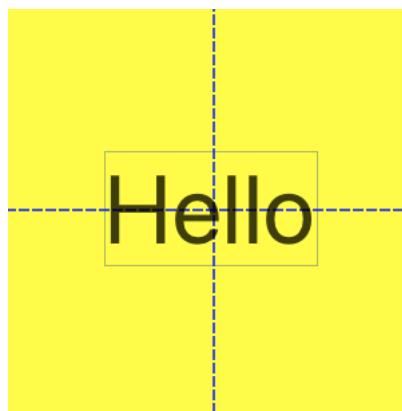
5. We can now design our view layout with Interface Builder. I changed the background color of the view to yellow and dragged a label from the object library onto the canvas:



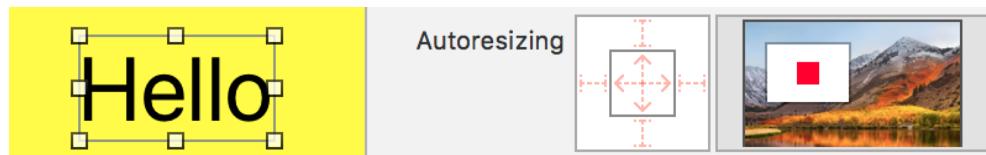
6. Change the text and increase the font size of the label to 36 points. You need to resize the label at this point to see the larger text:



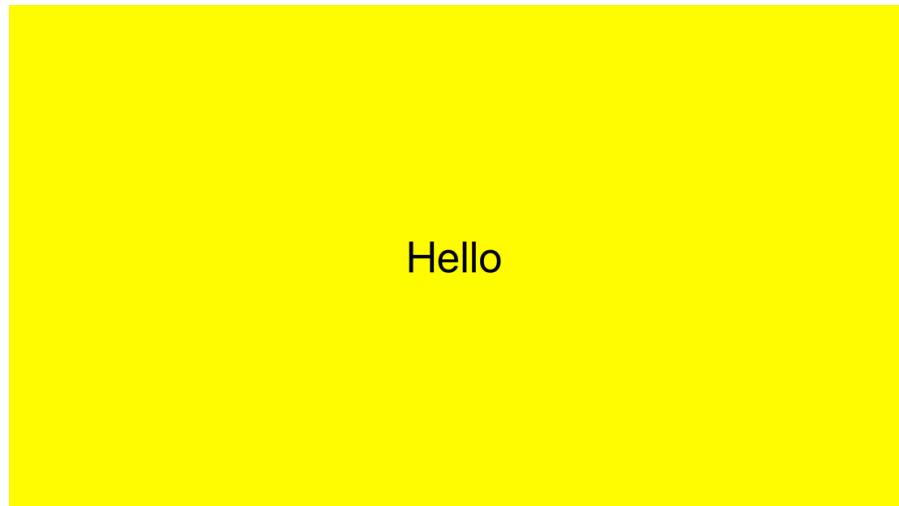
7. To make the size of the label exactly fit the text, select the label and use Editor > Size to Fit Content . You can then position the label in the horizontal and vertical center of the view using the center guidelines:



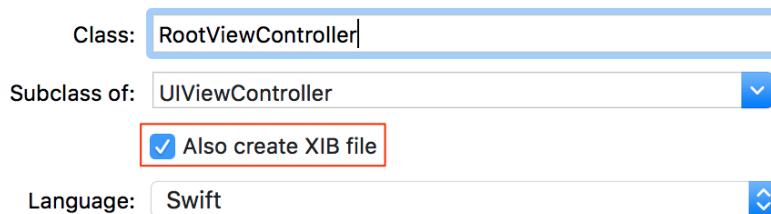
8. Fix the label in the center of the view and prevent it from being resized. Use the size inspector to remove the fixed leading and top struts that Xcode sets by default:



9. Build and run. You should see our yellow root view with a label centered in the screen. Here it's in landscape on an iPhone 8:



10. Xcode can create the nib file for you with the file's owner and view already set up. When you create the view controller select the "Also create XIB file" option:



Overriding loadView

If you don't load your views from a storyboard or nib file, you can create them manually in code in the view controller. One possible way to do this is to override `loadView()` (see sample code: [CodeLayout-v3](#)):

```
class RootViewController: UIViewController {
    override func loadView() {
        let rootView = UIView()
```

```
    rootView.backgroundColor = .yellow
    view = rootView
    // other view setup...
}
}
```

If you override `loadView()` you must create at least a root view and assign it to the `view` property of the view controller. The view doesn't have to be a plain `UIView`. It could be a `UIScrollView` or any other subclass of `UIView`. It can contain as many subviews as you want as long as you eventually assign it to `view`.



Don't override `loadView()` if you're using Interface Builder to create the view. If you do override `loadView()` don't call `super`.

You may have noticed that I didn't set a frame for the root view. It's the job of whatever is presenting the view controller to set the size and position of the view controller's root view. It might fill the screen, or be only a small part of the screen if the view controller is a child of a parent containing view controller.

We cannot rely on the size of the root view in `loadView` when calculating the sizes and positions of any subviews we want to add. Think back to our green view. It has the width of the root view minus the leading and trailing padding:

```
let greenWidth = view.bounds.width - 2 * padding
```

We can't do this calculation in `loadView()` as `view` doesn't have its final width set yet.

viewDidLoad and Friends

Let's look at some other view controller methods where we could do view setup:

- `viewDidLoad`: Called after the view controller has loaded its view but not yet added it to the view hierarchy. Called only once in the life of the view controller.
- `viewWillAppear`: Called when the view controller's view is about to be added to the view hierarchy. Unlike `viewDidLoad` this method can be called multiple times in the life of a view controller. There's

a corresponding `viewWillDisappear` called when the view is about to be removed from the view hierarchy.

- `viewDidAppear`: Called after view controller's view is added to the view hierarchy and displayed on-screen. Like `viewWillAppear` this method can be called multiple times and has a corresponding method, `viewDidDisappear`, called after removing the view.

Both `viewDidLoad` and `viewWillAppear` have the same problem we saw with `loadView`. The view is not part of the view hierarchy yet so we cannot rely on its size. By the time `viewDidAppear` gets called our root view is in the view hierarchy but it's already displayed so any changes we make may be visible to the user.



It's a common mistake to assume the size of views are correctly set in the `viewDidLoad` and `viewWillAppear` methods.

Where To Put Manual Layout Code?

So where should we put view controller layout code that needs the size of the root view? We've already seen the answer. A view controller calls its `viewWillLayoutSubviews` and `viewDidLayoutSubviews` methods before and after its root view lays out its subviews. We can override these methods to do our view layout.

Let's use this in our root view controller to layout our green view (see sample code: [CodeLayout-v4](#)):

```
// RootViewController.swift
import UIKit

class RootViewController: UIViewController {

    let padding: CGFloat = 50.0

    private let greenView: UIView = {
        let view = UIView()
        view.backgroundColor = .green
        view.autoresizingMask = [.flexibleWidth,
            .flexibleBottomMargin]
        return view
}()

}
```

```
override func viewDidLoad() {
    super.viewDidLoad()
    view.backgroundColor = .yellow
}

override func viewWillLayoutSubviews() {
    if greenView.superview == nil {
        view.addSubview(greenView)
        let width = view.bounds.width - 2 * padding
        greenView.frame = CGRect(x: padding, y: padding, width:
width, height: 3 * padding)
    }
}
```

Some points of interest:

1. I made the green view a private property of the view controller and used a closure to create and configure it.
2. I use `viewDidLoad` to do the one-time view setup that doesn't have any dependencies on the root view size. In this case, I only need to set the background color.
3. In `viewWillLayoutSubviews` we can finally calculate and set the width of the green view. We do the calculation once if we have not already added the green view to the view hierarchy:

```
if greenView.superview == nil {
```

If the green view has no superview, we add it to the root view and calculate its width and set the frame. The autoresizing mask takes care of the rest. It matters little in this case, but since a view controller may call `viewWillLayoutSubviews` many times, it's a good habit to be quick and avoid unnecessary work.

Key Points To remember

Some key points to remember from this chapter when using manual frame-based layout:

- Autoresizing has some limitations but is a quick way to resize a view with its parent. As we'll see autoresizing also works well alongside Auto Layout.
- Create a custom subclass of `UIView` to override `layoutSubviews`

and take full control of the layout. Custom subviews help split a complex view into manageable components and move view and layout code out of your view controller. Use `@IBDesignable` and `@IBInspectable` to preview custom views in Interface Builder.

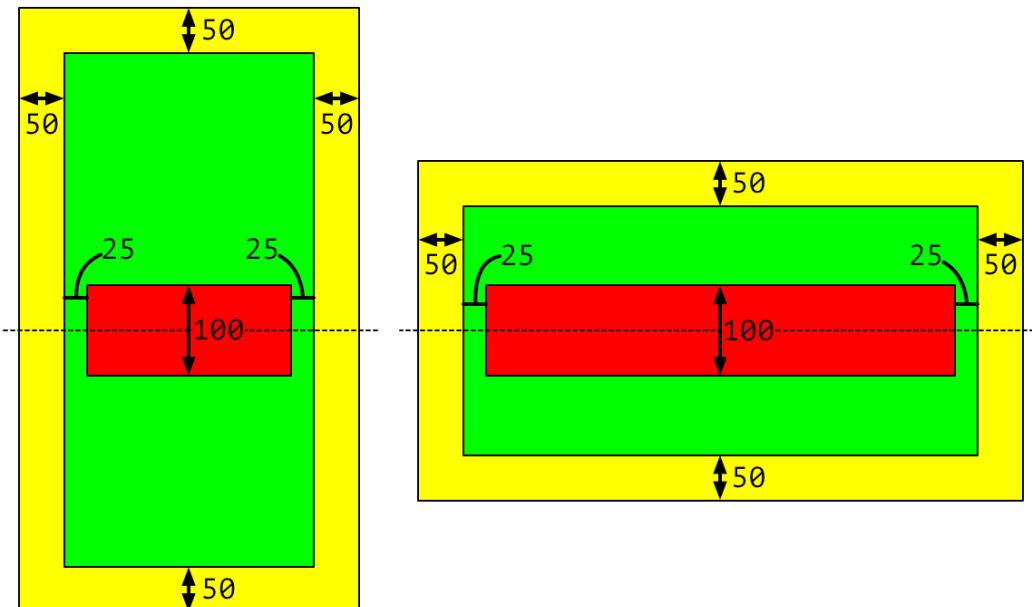
- Use `viewWillLayoutSubviews` or `viewDidLayoutSubviews` as alternatives to creating a custom view subclass.
- When using programmatic layouts, especially when manually calculating frames, be careful what you do where. Remember that a view is not yet part of the view hierarchy in `loadView`, `viewDidLoad` or `viewWillAppear` so you cannot assume it has reached its final size. Fortunately, Auto Layout mostly avoids this problem.

Test Your Knowledge

Time for you to give it a go. These challenges get you used to building layouts. You should only need techniques we've covered in this chapter. Test your layouts using the simulator to try them on a variety of devices in both portrait and landscape.

Can You Auto Resize?

For your first challenge here's the layout to create shown in portrait and landscape:



- The green container view has an outer margin of 50 points on all

sides to the edge of the device and an internal margin of 25 points.

- The red view has a fixed height of 100 points and is centered vertically in the green container view.

Challenge 2.1 - Using Interface Builder

1. Create the layout using Interface Builder to set the size and position of the views manually.
2. Use springs and struts to make the views resize. You should be able to solve this challenge without writing any code.

Challenge 2.2 - Layout In Code

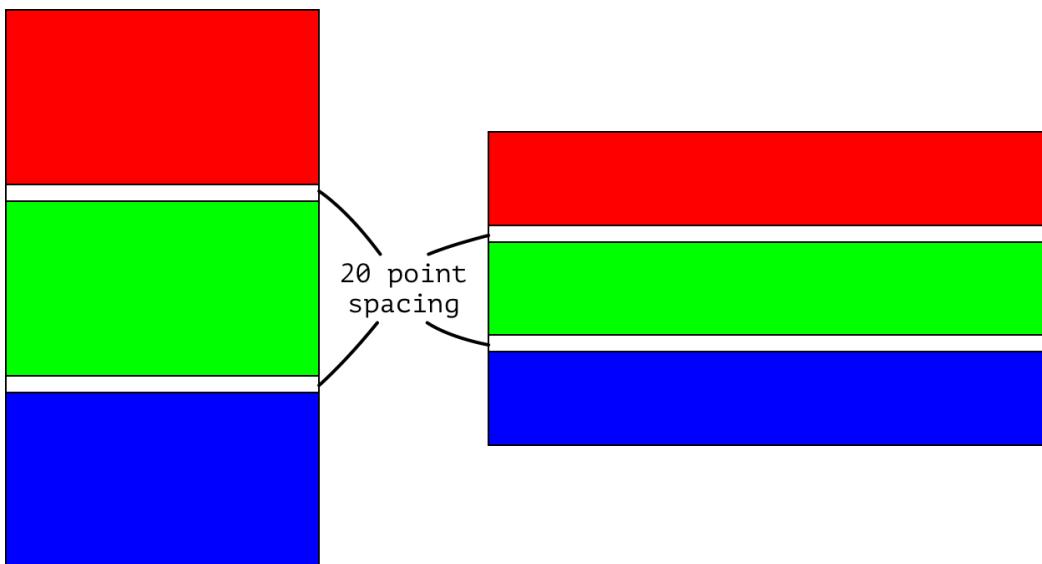
1. Create the same layout but this time without using Interface Builder.
2. Delete the main storyboard from your Xcode project and build your layout in code.
3. For this simple case, you can get by without a custom view, build your layout in the view controller.

Hints And Tips

1. I suggest you use the single view app Xcode template for these challenges when working with Interface Builder. See the walkthroughs in this chapter if you need a recap.
2. Make the red view a subview of the green view. Remember that the origin of the red view is then relative to the green view.
3. When building the layout in code you need to perform your calculations in `viewWillLayoutSubviews`. Remember to keep this method fast. Only set the frames of the green and red views once.
4. Working out the size and position of the two views and the correct autoresizing mask is a pain. The best way to avoid this is to use Auto Layout which is what we are going to do in the next chapter.

Using A Custom View

For your next challenge here's a three view layout to build shown again in both portrait and landscape:



- Red, green and blue bars fill the full width of the device screen.
- The three bars all have the same height.
- There's 20 points of spacing between the bars in both portrait and landscape orientations.

Challenge 2.3 Creating A Custom View

1. Make a custom subclass of `UIView` named `RGBView` that creates the three bar layout. You can choose to use your view with a storyboard or create the whole layout in code.
2. Implement both initializers for your custom view so that you can use it in a storyboard or create it programmatically.

Challenge 2.4 Making Your View Designable

1. Make your custom view designable and preview it in Interface Builder.
2. Make the spacing between the views a property of your custom view and then make it inspectable so that you can change the property in Interface Builder.

Hints And Tips

1. You cannot solve this one using autoresizing masks as the space between the views must stay fixed at 20 points.

2. You could build this layout in the view controller, but I think it's better to move the view layout code to a custom view.
3. You can use a storyboard for your view controller and add your custom view as a subview of the view controller's root view. If you follow this approach, you should not need to add any code to the view controller.
4. You need to size your custom view so that it fills the bounds of the view controller's view. Set the autoresizing mask for your custom view so that when the root view resizes your view continues to fill the screen.
5. Don't forget to set the class of the view in the storyboard using the identity inspector.
6. The width of each of the three bars is the width of the superview.
7. The total height of the three bars is the height of the superview less the spacing between the three bars.
8. The height of each bar is `(bounds.height - 2 * spacing)/3`.

Chapter 3

Getting Started With Auto Layout

What Is Auto Layout?

To quote from the WWDC 2012 Session that introduced us to Auto Layout with iOS 6:

Auto Layout Is a Constraint-Based, Descriptive Layout System

—WWDC 2012 Session 202 Introduction to Auto Layout for iOS and OS X

That's a fancy way of saying that you describe your view layouts with a set of relationships. Instead of directly setting the height of a view you relate it to the height of another view. For example, "the red and green views have the same height" rather than "the red view is 100 points high".

The Auto Layout engine produces a layout fitting your description. You have sufficiently described your layout when there's a single possible solution.

Compare this to how we manually managed layout in the last chapter. We directly set the frames of our views and then using either autoresizing masks or manually in `layoutSubviews` recalculated the layout each time something changed.

With Auto Layout you never directly set the size and position of a view. You describe how you want the layout to be and let Auto Layout work out the frames for you.

Auto Layout is a different way to think about layout, and getting used to it takes time. Is it worth it? Another quote from WWDC 2012 on the promise of Auto Layout:

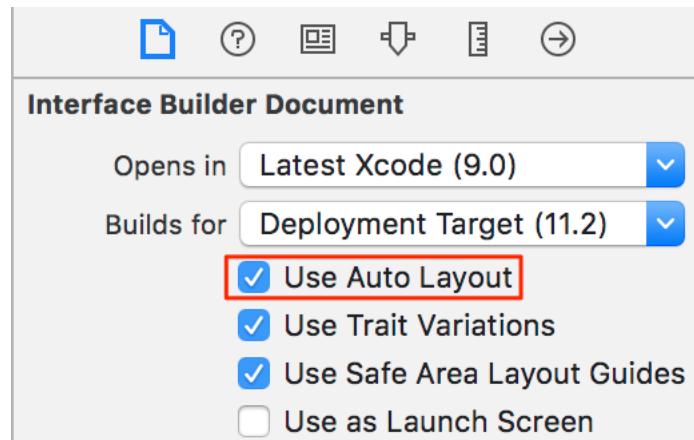
Make your layouts simpler to write, simpler to modify,
easier to understand

—WWDC 2012 Session 228 Best Practises For Mastering Auto Layout

I'll let you judge how well it keeps that promise as we dive into the details.

Switching To Auto Layout

For all new Xcode projects, Interface Builder uses Auto Layout by default. It's possible to turn support off for individual Interface Builder documents. Use the file inspector to check if you're not sure if it's on or off:



Do I Have To Use Auto Layout?

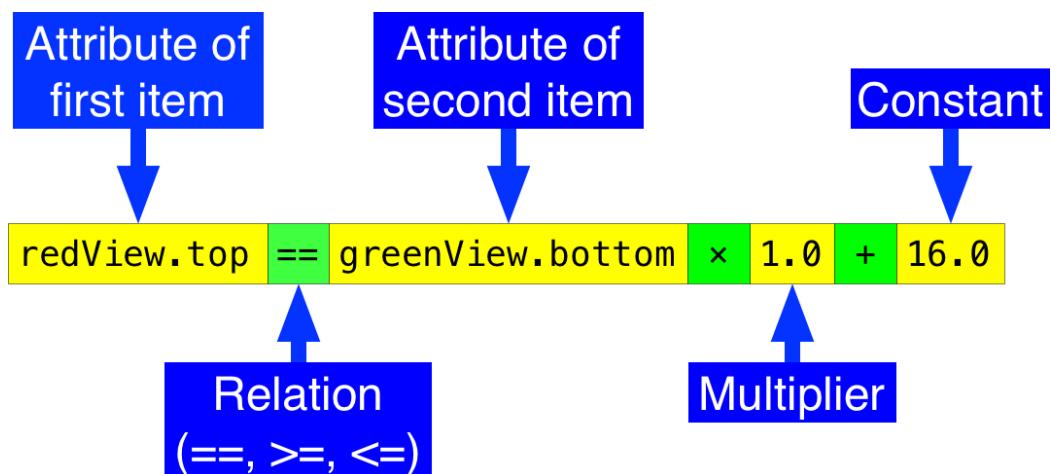
The short answer is no. I hope after reading this book you see some of the advantages of Auto Layout, but you're not forced to use it. You can also mix and match using Auto Layout for some views and autoresizing masks or `layoutSubviews` for others. Just don't set the frames for views that are using Auto Layout.

What Is A Constraint?

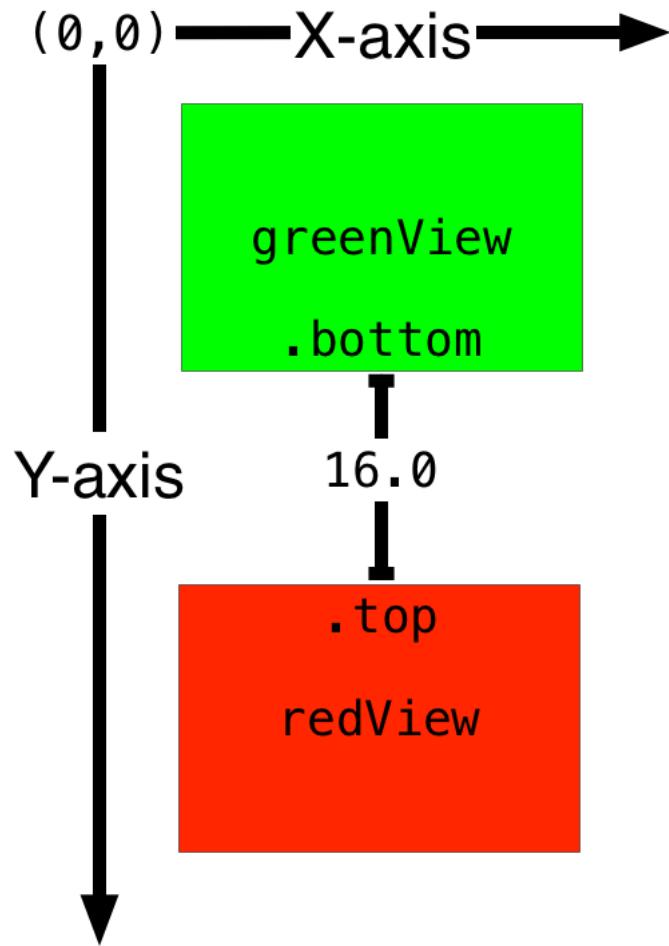
You can think of a constraint as the way you tell Auto Layout facts about your layout. Here are some examples of constraints:

- The top of the red view should be 16 points below the bottom of the green view.
- The leading edge of the green view should be greater than or equal to the leading edge of its superview.
- The green view should be twice as high as the blue view.
- The center along the Y-axis of the red view should be halfway between the center of its superview.
- The blue view should be 50 points wide.

Except for the last example, these constraints relate the attribute of one view (top of red view) to an attribute of a second view (bottom of green view). Without getting too mathematical, you can write them like this:



The coordinate system for iOS puts the origin (0,0) at the top-left corner. A positive value for the constraint constant moves you down the screen from top-to-bottom for vertical constraints and across the screen from leading-to-trailing for horizontal constraints:



We could write a constraint to put the top of the red view 16 points below the bottom of the green view:

```
redView.top == greenView.bottom x 1.0 + 16.0
```

We could also write this same constraint by placing the bottom of the green view 16 points above the top of the red view with a negative constant:

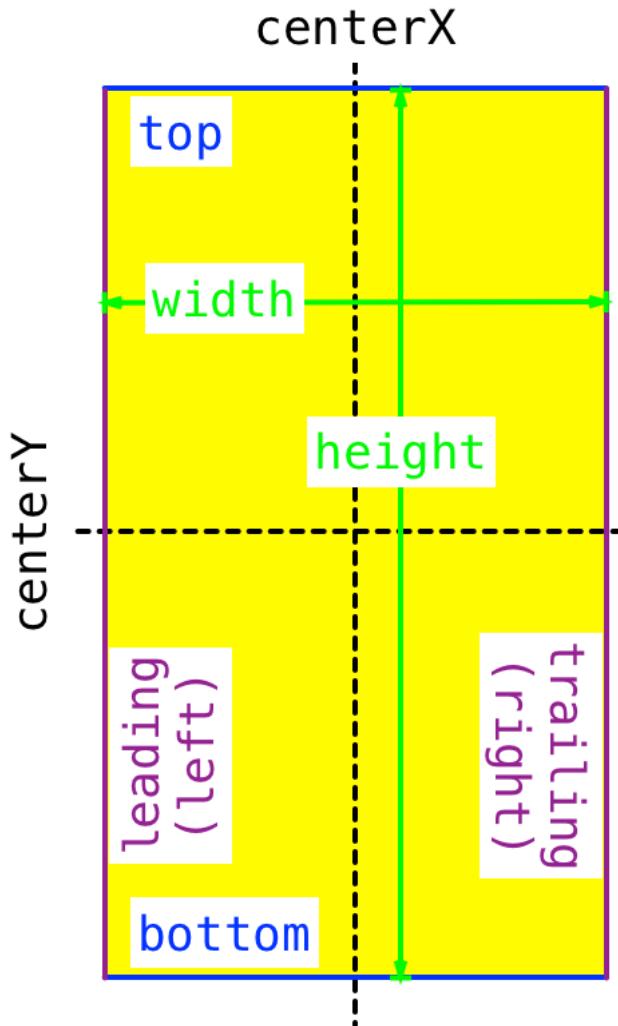
```
greenView.bottom == redView.top x 1.0 - 16.0
```

I prefer to work with positive values for the constant but either way works.

Constraint Attributes

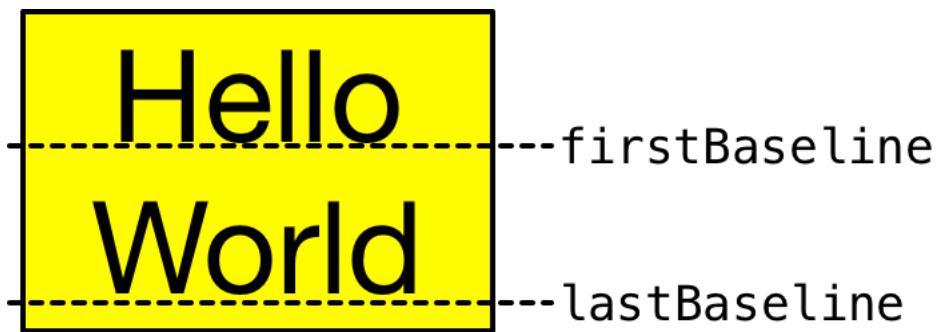
So what are the attributes you can use when creating constraints? The `NSLayoutAttribute` enum gives you a number of choices:

- Edges: `.top`, `.bottom`, `.leading`, `.trailing`, `.left` and `.right`
- Center: `.centerX` and `.centerY`
- Size: `.width` and `.height`

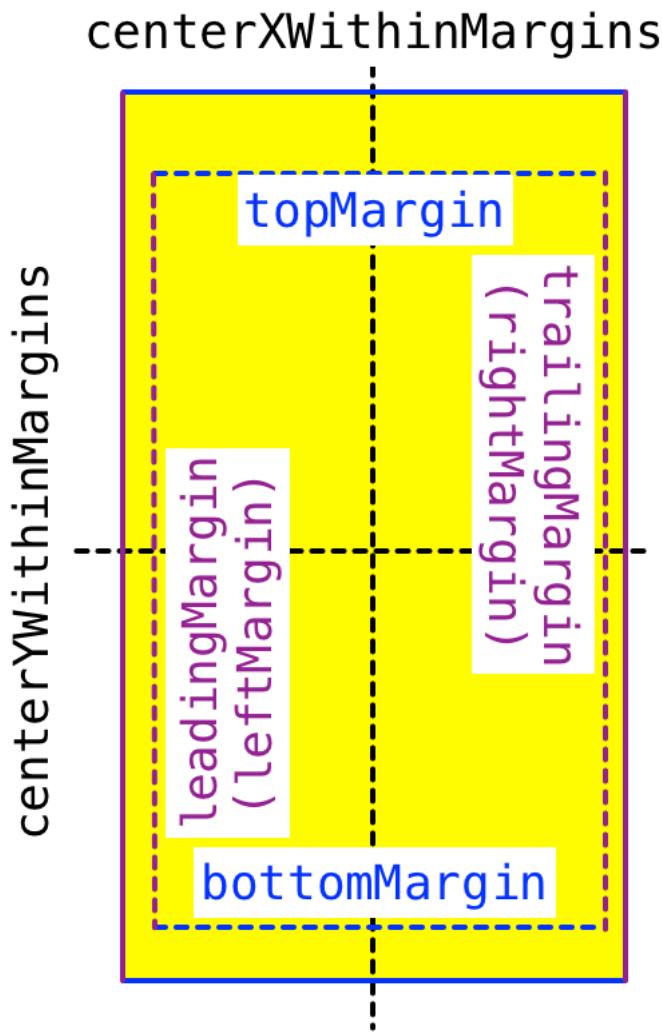


Use `.leading` and `.trailing` instead of `.left` and `.right` to support Right-To-Left (RTL) languages. When using an RTL language the `.leading` edge is on the right and the `.trailing` edge is on the left.

A text view such as a `UILabel`, `UITextField` or `UITextView` has two extra attributes so that you can align text on the first or last baseline:



A view also has a full set of margin attributes for when you want to inset content from the view edges:



You can create many different types of constraint depending on which attributes you use (I'm using pseudo-code to write constraints in this chapter, we'll see the real syntax later):

```
// Aligning the leading edges of views  
redView.leadingAnchor == greenView.leadingAnchor  
  
// Aligning view centers (x-axis)  
redView.centerX == greenView.centerX  
  
// Vertical spacing between views  
redView.topAnchor == greenView.bottomAnchor + 8.0  
  
// Two views with equal width  
redView.width == greenView.width  
  
// Height of view is half the height of another view  
redView.height == 0.5 * greenView.height  
  
// Red view is never wider than green view  
redView.width <= greenView.width  
  
// Aspect Ratio  
greenView.height = 0.5 * greenView.width  
  
// Setting a constant size  
redView.height = 50.0  
redView.width = 75.0
```

Interface Builder doesn't let you create meaningless constraints but be aware when creating constraints in code that not all combinations of attributes make sense:

```
// ERROR - don't mix horizontal with vertical  
redView.leadingAnchor == greenView.bottomAnchor + 16.0  
  
// ERROR - cannot mix size with edge or center  
greenView.width == redView.leadingAnchor  
  
// ERROR - cannot set edge or center to a constant  
redView.centerY = 100.0  
  
// ERROR - don't mix leading/trailing with left/right  
redView.left == greenView.trailingAnchor + 16.0
```

Who Owns A Constraint?

Every view has a `constraints` property which is an array of constraints owned by that view. It's rare unless debugging that you need to make use of it. The property is read-only:

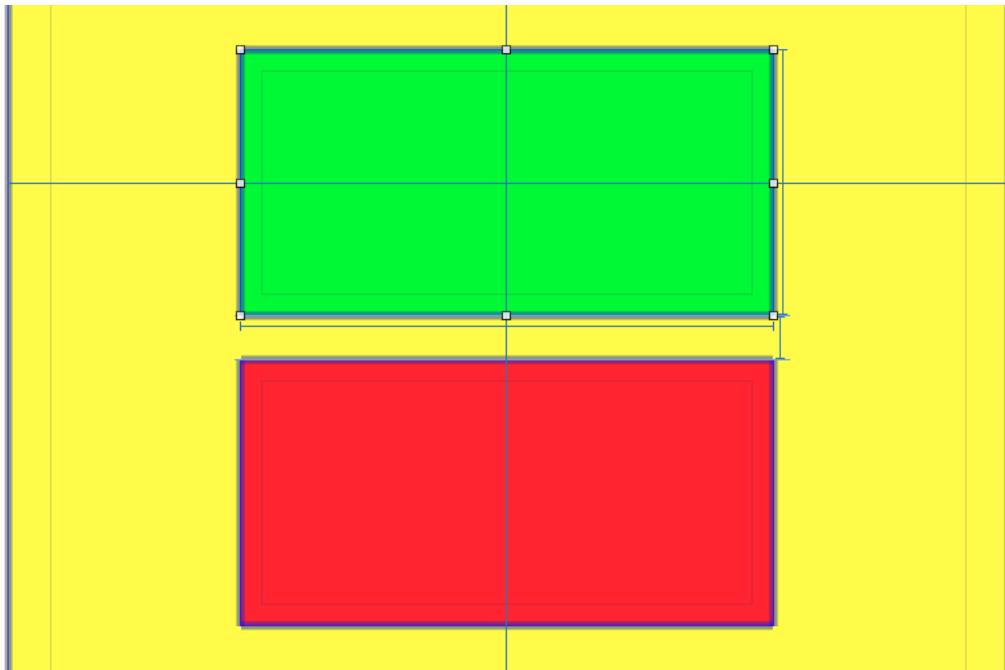
```
var constraints: [NSLayoutConstraint] { get }
```

How does a view end up owning a constraint and perhaps more importantly which view should own which constraints? There's a simple rule:

A constraint owned by a view can only involve that view itself or its subviews.

If you create a constraint between two views, they must have a common superview to own the constraint. An example should help make it clear.

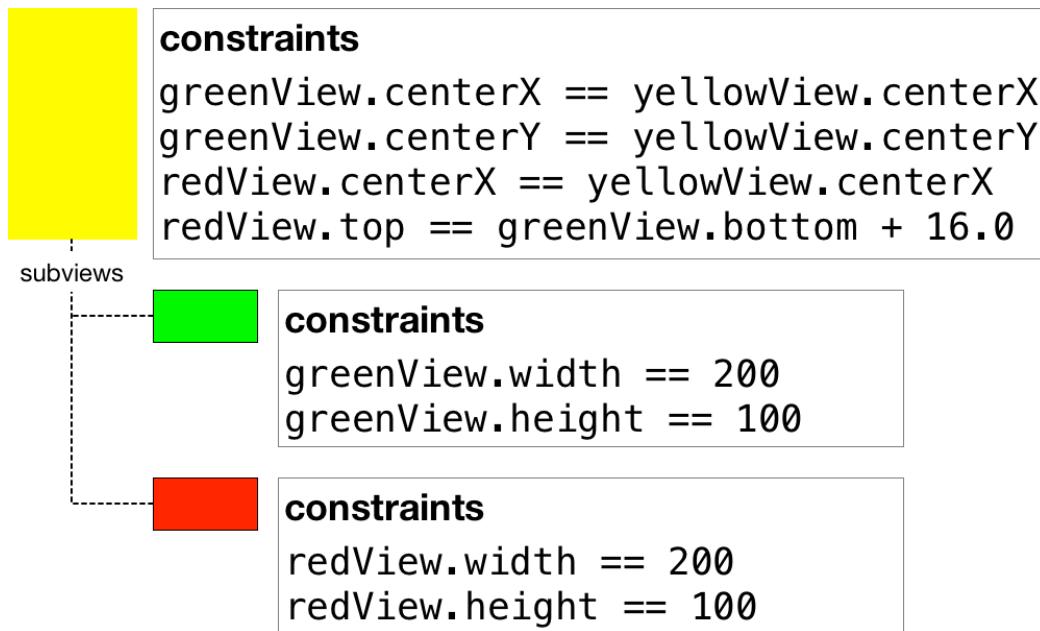
Here's a layout I created with Interface Builder by dragging two plain views onto the yellow root view of a view controller. Both the green and red views are subviews of the yellow view:



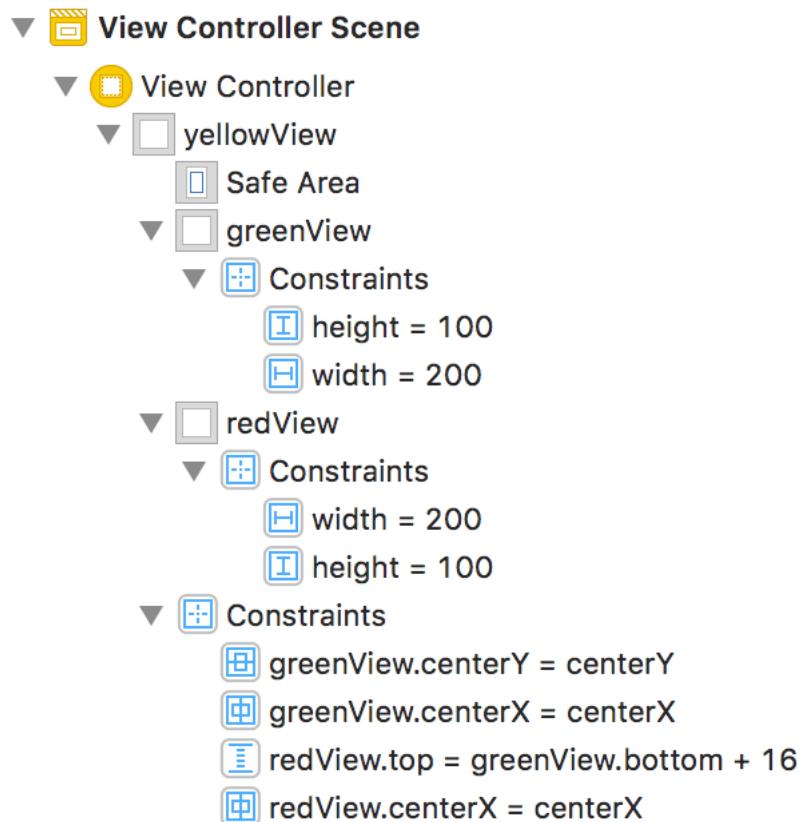
I added constraints to center the green view vertically and horizontally in the yellow superview. The red view is also centered horizontally in the yellow superview, and I spaced it 16 points below the green view. Both the green and red views have fixed height and width constraints.

So which view owns which constraint? The two constraints fixing the width and height of the green view only involve the green view, so it owns them directly. Likewise, the red view owns its width and height constraints. The other constraints are either between the green and red view or one of those views and the yellow superview, so the yellow view

owns them.



If you're creating your constraints with Interface Builder, it mostly hides these details. You can always see which view owns which constraints in the document outline.



How Many Constraints Do I Need?

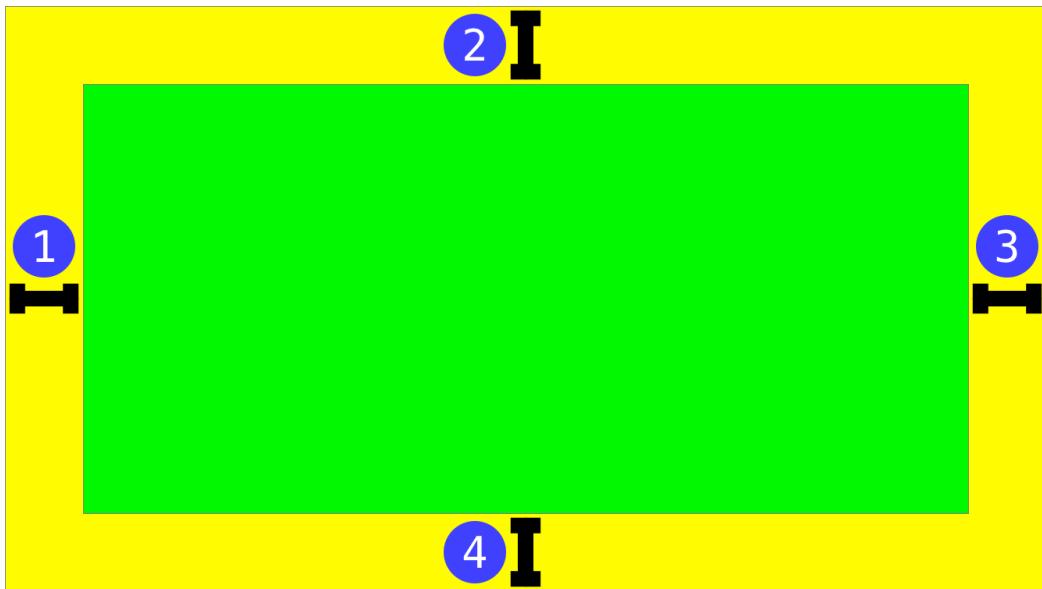
The challenge when working with Auto Layout is to add just enough constraints to describe a single, unambiguous layout. How many is enough?

You need at least enough constraints to fix the size and position of every view in your view hierarchy.

Work through your layout asking yourself if you have told the layout engine enough to fix the position (origin or center) and the size (width and height) of every view. In general, this needs two horizontal and two vertical constraints, but it's not always obvious.

Pinning The Edges

Let's start with pinning a single view to the edges of its superview with some padding:



First we can fix the position (origin) of the green view with leading and top constraints to the superview. I'm using 50 points for the padding:

```
greenView.leadingAnchor == superview.leadingAnchor + 50 // 1  
greenView.topAnchor == superview.topAnchor + 50 // 2
```

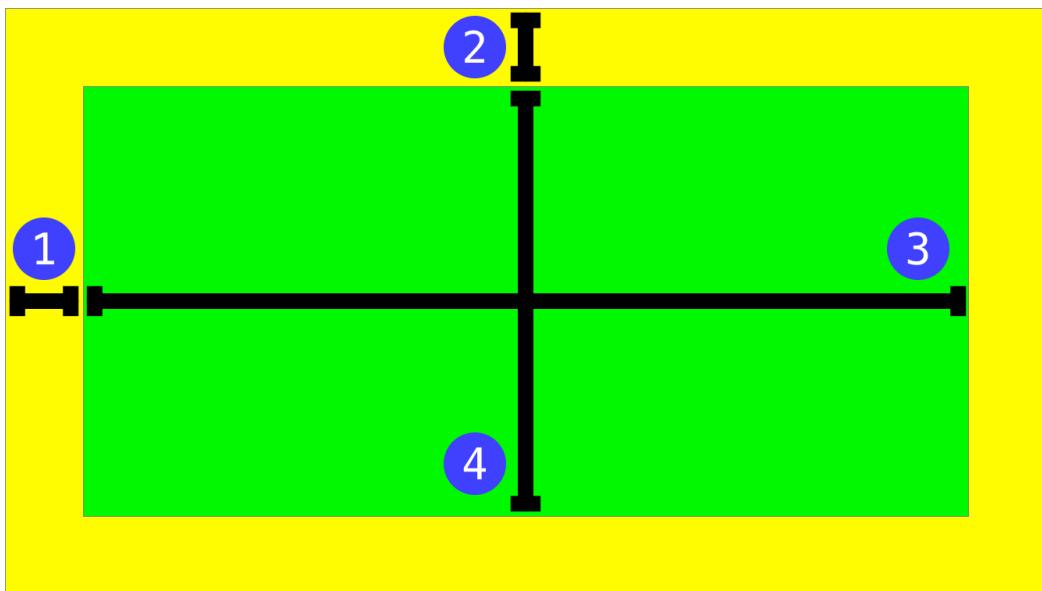
Adding trailing and bottom constraints fixes the size (width and height):

```
superview.trailing == greenView.trailing + 50 // 3
superview.bottom == greenView.bottom + 50 // 4
```

So we needed four constraints, 2 horizontal + 2 vertical, to create an unambiguous layout for one view.

Using A Constant Size

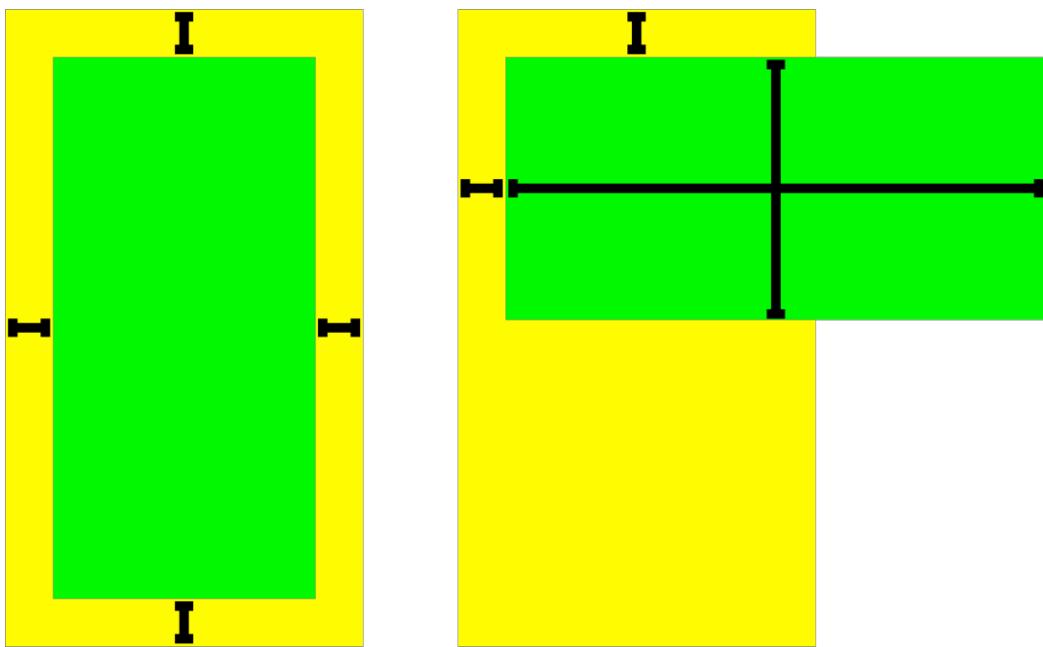
What can make Auto Layout confusing is that there's often more than one way to create a layout. We fixed the size of the green view by adding trailing and bottom constraints. What if we directly fixed the width and height instead:



Replacing the trailing and bottom constraints with constant width and height constraints:

```
greenView.leading == superview.leading + 50 // 1
greenView.top == superview.top + 50 // 2
greenView.width == 567 // 3
greenView.height == 275 // 4
```

We still need four constraints so is this any better? What would happen if we rotate the device from landscape to portrait? Our first try, shown on the left below, adapts well calculating a new width and height for the green view based on the new size of the superview. The second try using fixed width and height constraints, shown on the right, is not so successful as part of the view disappears off screen:



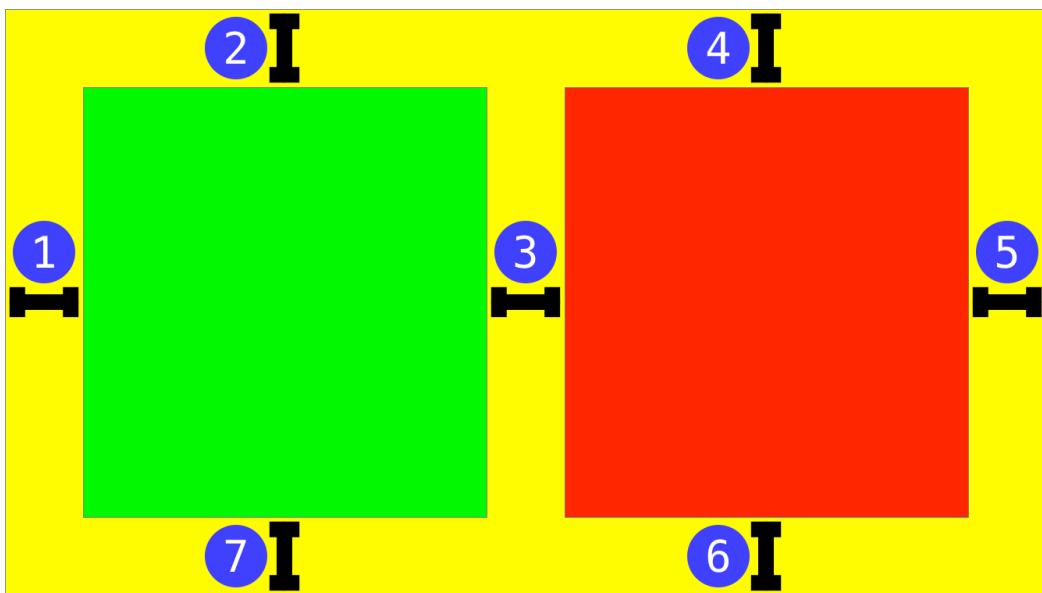
Using a fixed constant for the width and height prevents it from adapting to changes in the size of the superview.



Avoid adding constant width and height constraints for your views if possible. Prefer to make them relative to some other dimension in your layout.

Equal Sizing

Let's add a red view to our example that should be the same size as the green view. I keep the same padding around and between the views:



We can start as before adding leading, trailing, top and bottom constraints to pin the two views to the edges of the superview with 50 points of padding. There's also a constraint for the spacing between the two views:

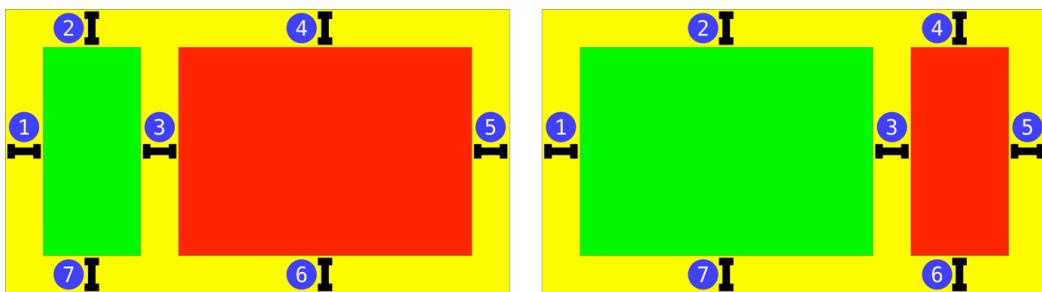
```
redView.leadingAnchor == greenView.trailingAnchor + 50 // 3
```

At this point, we have three horizontal and four vertical constraints. Is this enough? Not sure? Remember we need enough constraints to fix the position and size of each view.

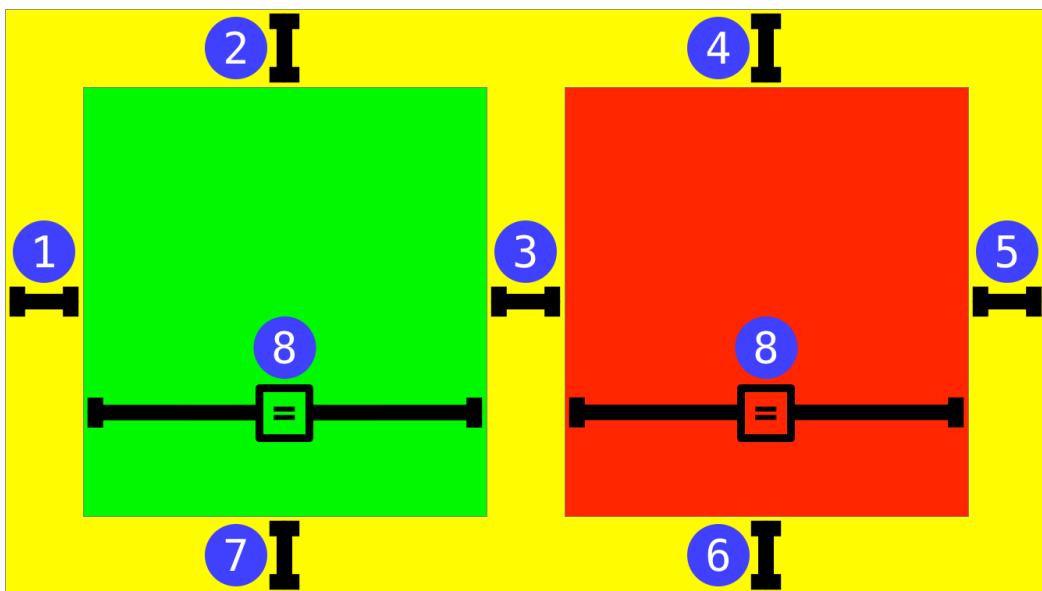
Did we fix the position of the green view? The leading (1) and top (2) constraints certainly do that.

Did we fix the size of the green view? At first, this might not be obvious. The trailing constraint (3) would seem to fix the width of the green view based on the width of the red view. What's fixing the width of the red view? Can you see why this is ambiguous?

The problem is that we don't have enough constraints to produce a single solution for the widths of the green and red views. Take a look at these two possible solutions for the layout. They both satisfy the constraints we have given but with different widths for the two views.



To fix the layout, we need another horizontal constraint that fixes the width of the views. Let's add a constraint to give the green and red views equal width:



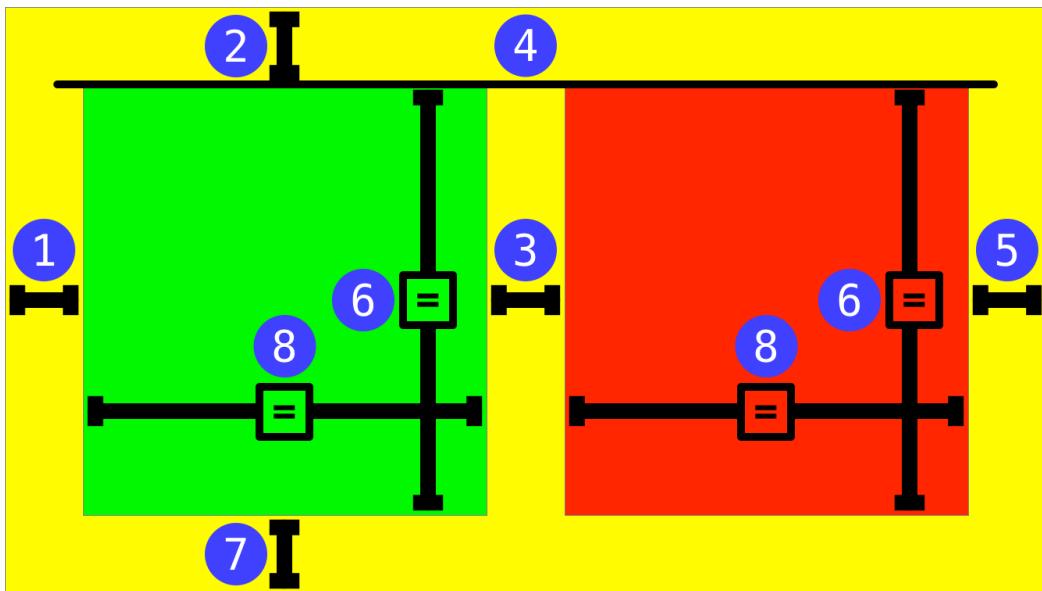
```
greenView.width == redView.width // 8
```

We have a single possible layout solution for our two views with four horizontal and four vertical constraints. There are other ways to create this layout. For example, replacing the top constraint (4) with a constraint that aligns the tops of the green view and red view with each other:

```
greenView.top == redView.top // 4
```

We could also replace the bottom constraint (6) for the red view with an equal height constraint:

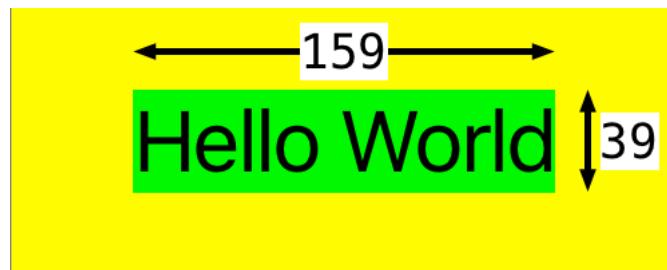
```
greenView.height == redView.height // 6
```



I find the first approach more natural to understand but either way you end up needing four horizontal and four vertical constraints.

Views With An Intrinsic Size

Some views already have a natural size that can simplify your constraints. For example, a `UILabel` has a natural size based on its text and font settings. This single line label with the text “Hello World” using the 32-point system font has a natural size of 159 points x 39 points:



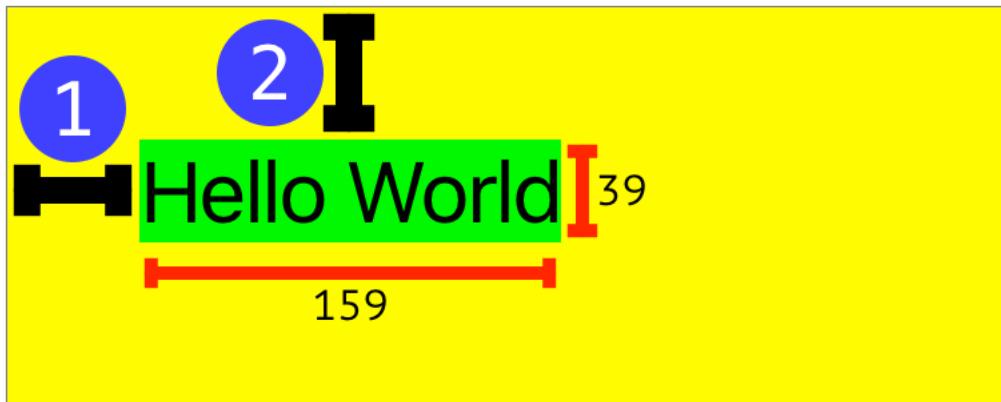
If we use this label in a layout, we only need to add constraints to fix the position. The layout engine uses the natural size of the label if no other size constraints are acting on the label. So with just two constraints to fix the position, we have a working layout:



The leading constraint (1) fixes the x-position, and the top constraint (2) fixes the y-position of the label.

```
label.leadingAnchor == superview.leadingAnchor + 50 // 1  
label.topAnchor == superview.topAnchor + 50 // 2
```

Are we cheating using only two constraints when we might expect to need four? Where are the other two constraints to fix the size? When we take a more in-depth look into [Intrinsic Content Size](#) we'll see that the layout engine adds the extra width and height constraints for us from the natural size of the label. So this layout is still using two horizontal and two vertical constraints.



Adding More Constraints

In this chapter, we have looked at adding just enough constraints to allow Auto Layout to find a single, unambiguous solution. That's a good start, but it's not the whole story. There are situations where you want to add more than the minimum set of constraints.

Think about the last example with the text label. We were able to take advantage of the natural content size of the label to create a working layout with just two constraints. That's fine but what if we want the label to fill the width of the screen? No problem, we describe what we want and let Auto Layout figure out the details.

For example, adding a trailing constraint to stretch the label across the width of the superview:



```
label.leadingAnchor == superview.leadingAnchor + 50 // 1  
label.topAnchor == superview.topAnchor + 50 // 2  
superview.trailingAnchor == label.trailingAnchor + 50 // 3
```

This three constraint pattern allows the label height to expand down the screen as the text or font size grows. For example, here's how the label looks if I increase the font size to 48 points.



Test Your Knowledge

Test your understanding of constraints. Answers at the end of the chapter:

1. Can you spot the invalid constraints?

```
// Constraint A  
redView.trailing == greenView.top + 20.0  
  
// Constraint B  
redView.top == greenView.bottom + 8.0  
  
// Constraint C  
redView.leading == greenView.width + 10.0  
  
// Constraint D  
greenView.centerX = 100.0  
  
// Constraint E  
redView.leading == greenView.right + 8.0
```

2. Which of these two constraints should you prefer? Why?

```
// Constraint A  
titleLabel.leadingAnchor = view.leadingAnchor + 8.0  
  
// Constraint B  
titleLabel.left = view.left + 8.0
```

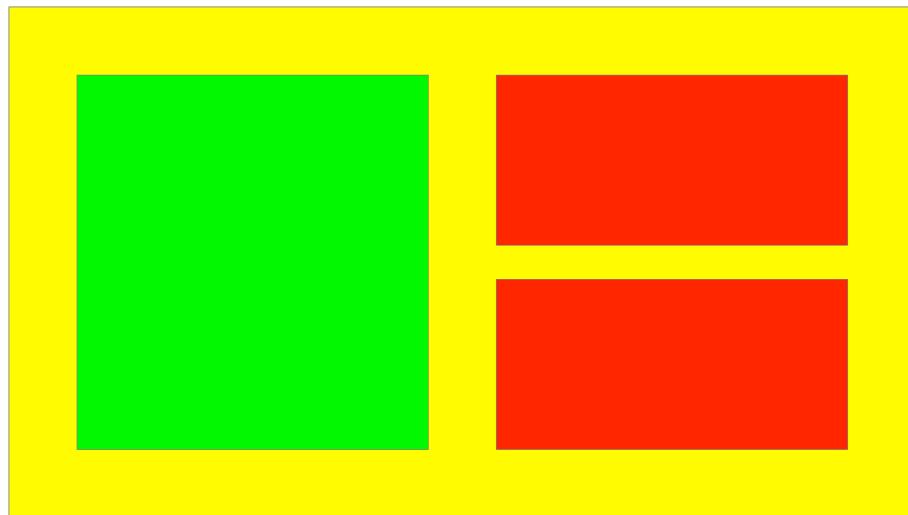
3. Match the constraints that describe the same relationship:

```
// Constraint A  
greenView.leadingAnchor == redView.trailingAnchor + 8.0  
  
// Constraint B  
redView.leadingAnchor == greenView.trailingAnchor + 8.0  
  
// Constraint C  
redView.trailingAnchor == greenView.leadingAnchor - 8.0  
  
// Constraint D  
greenView.trailingAnchor == redView.leadingAnchor - 8.0
```

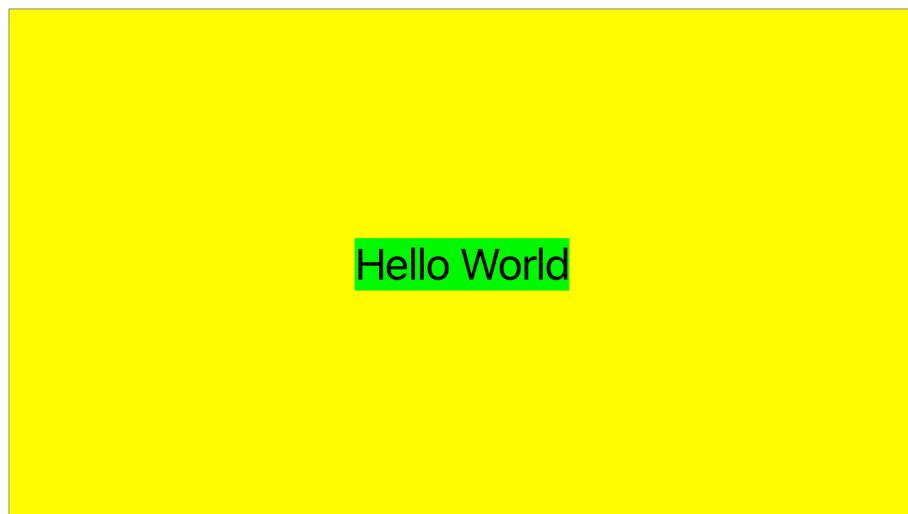
4. Which of these constraints would position the green view 8 points below the yellow view?

```
// Constraint A  
greenView.top = yellowView.top + 8.0  
  
// Constraint B  
greenView.top = yellowView.bottom - 8.0  
  
// Constraint C  
yellowView.bottom = greenView.top - 8.0
```

5. What's the minimum number of constraints you need for this layout?

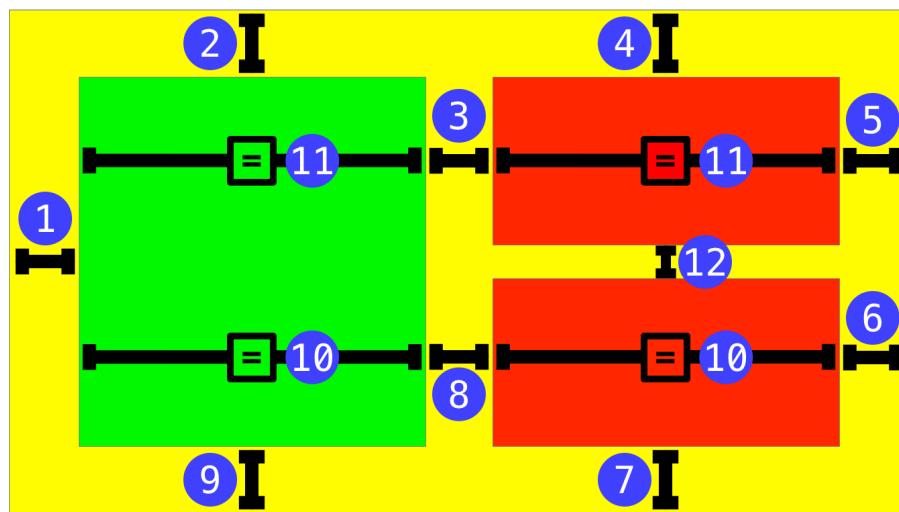


6. What's the minimum number of constraints you need to center this text label in the yellow view? Can you think of a situation where you might add more than the minimum number of constraints?

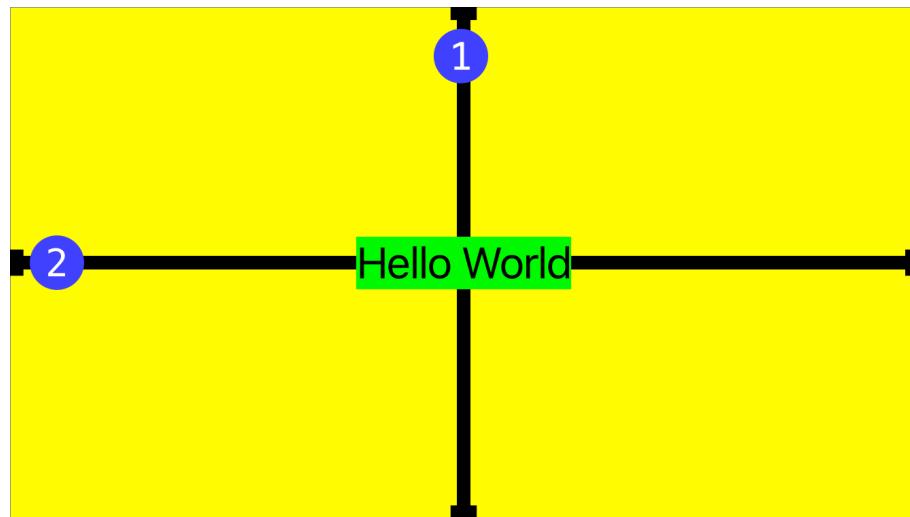


Answers

1. The invalid constraints:
 - A: mixes x-axis (trailing) and y-axis (top) attributes.
 - C: mixes x-axis (leading) and dimension (width) attributes
 - D: cannot set edge or center attribute to a constant
 - E: cannot mix leading/trailing with left/right
2. A. Prefer leading/trailing to left/right to support right-to-left languages.
3. A & C, B & D
4. C
5. 12 constraints. Lots of ways to do this. Here's my solution:



6. 2 constraints



```
textLabel.centerX == yellowView.centerX // 1  
textLabel.centerY == yellowView.centerY // 2
```

The system supplies a width and height constraint that fits the size of the text. You can add extra constraints if you want the label bigger or smaller than this “natural” size.

We are also ignoring, for now, what happens if the text size gets too big to fit within the bounds of the yellow view. A valid solution for the auto layout engine doesn’t necessarily mean your content is on screen! It means there’s a single unambiguous layout that satisfies the constraints.

Chapter 4

Using Interface Builder

In this chapter, we use Interface Builder to create our constraints. I recommend starting with Interface Builder when first learning Auto Layout. We look at creating constraints in code in the next chapter.

After reading this chapter, you should be able to:

- Navigate Interface Builder to create Auto Layout constraints.
- Use the different Auto Layout tools, canvas, and document outline.
- Use Interface Builder to edit a constraint.
- Understand the layout warnings and errors that Interface Builder reports.

If you're new to Xcode and Interface Builder see [Appendix A: Tour Of Interface Builder](#) for an introduction.

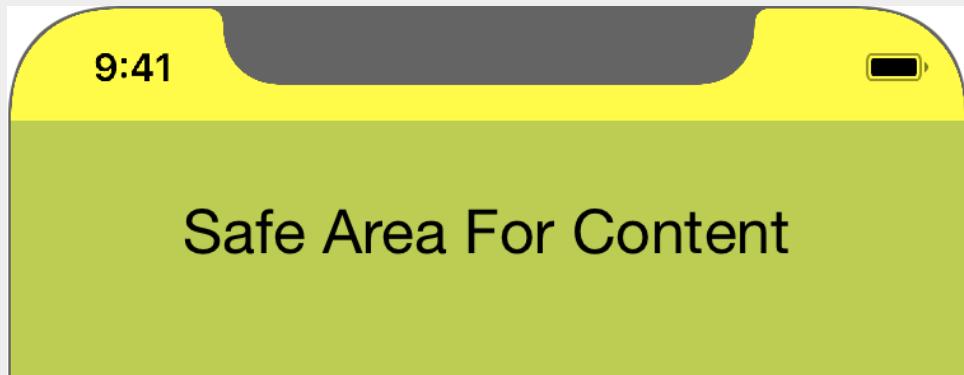
The Many Ways To Create A Constraint

Your first reaction to using Interface Builder to create constraints might be confusion. Don't worry! There are many different ways to do the same task and often no clear best way. The Xcode engineers at Apple also like to tweak and change the way Interface Builder works from time-to-time. Use this chapter to explore and find what works for you.

Safe Area For Content

Apple added the safe area in iOS 11 to define a rectangle safe for you to show content. The safe area is not clipped by the rounded

edges of the iPhone X or covered by system controls like the home screen indicator or top sensor housing.

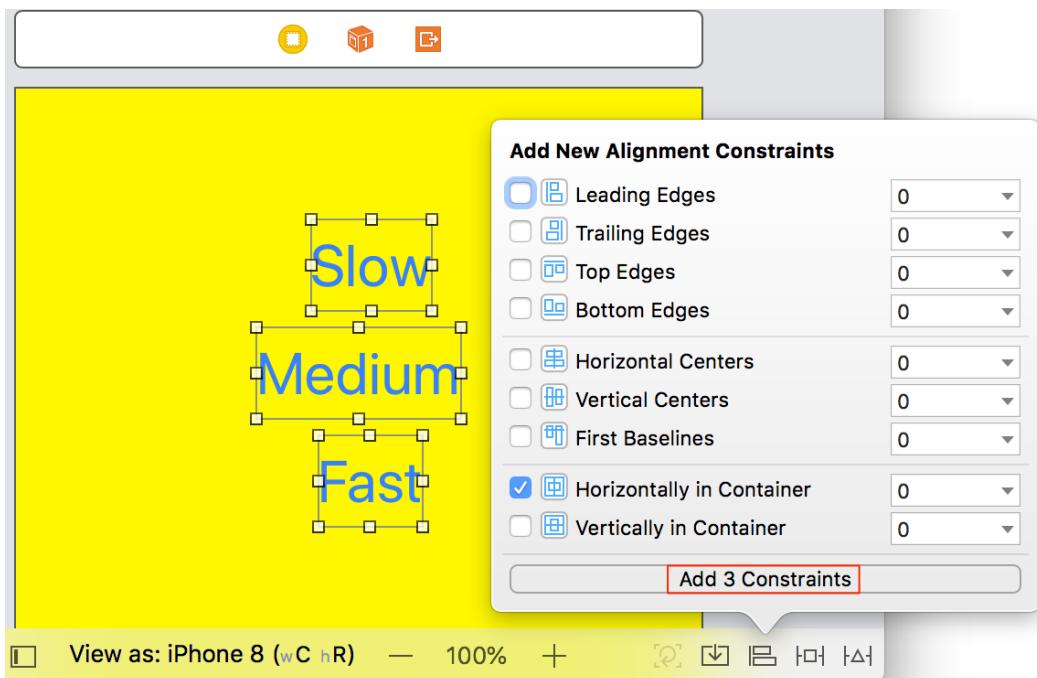


When creating constraints, we can choose to make them to the edge of a view, the safe area or the view margins. Using the edge fills the view but with the risk that the device corners or controls clip or cover our content. Using the margin gives us extra spacing inside the safe area. I cover this in more detail when we look at [Safe Areas And Layout Margins](#). In this chapter, we keep our content within the safe area.

Using The Align Tool



The align tool in the toolbar at the bottom right of the Interface Builder canvas is handy when you want to line up the edges or centers of several views. It's also a quick way to center views horizontally or vertically within their container view. For example, to horizontally center these three buttons in the yellow superview.



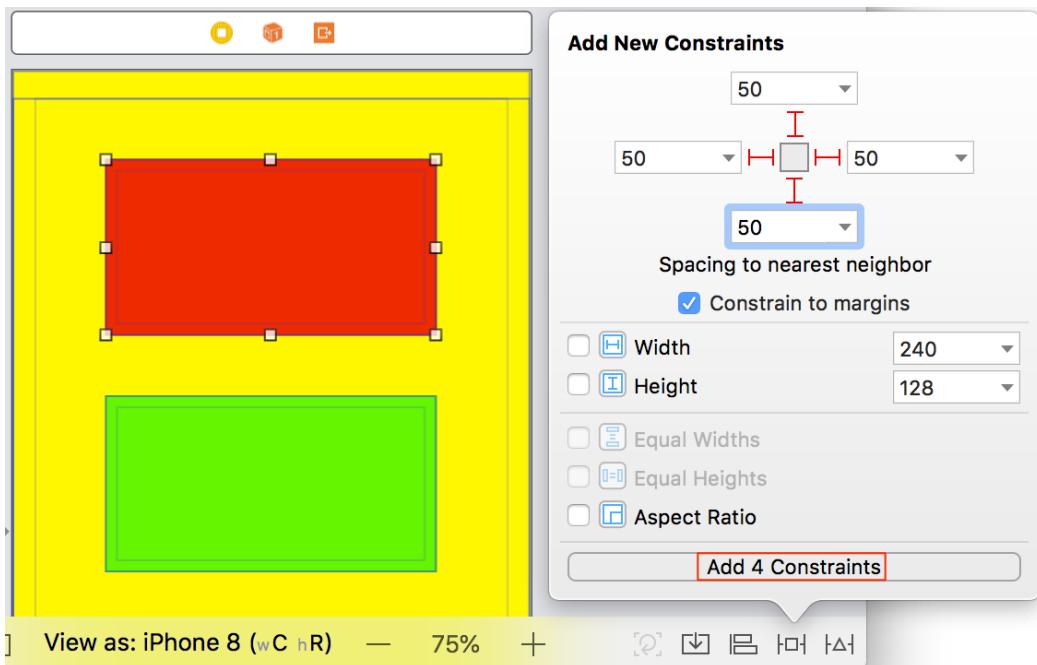
Note that this adds three constraints. Make sure you have the views selected before you open the tool.

Using The Add New Constraints Tool



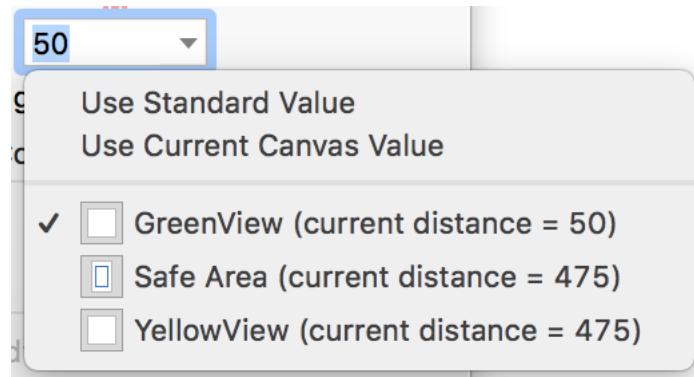
The add new constraints tool, next to the align tool in the toolbar, is a bit of a general purpose tool. It has several choices for creating constraints depending on which view or views you have selected. The equal widths and heights and the align constraints only apply when you're working on more than one view.

I use it mostly for creating horizontal and vertical spacing constraints. For example, with a single view selected use the top four boxes to add leading, top, trailing or bottom spacing between the view and its nearest neighbor or superview.



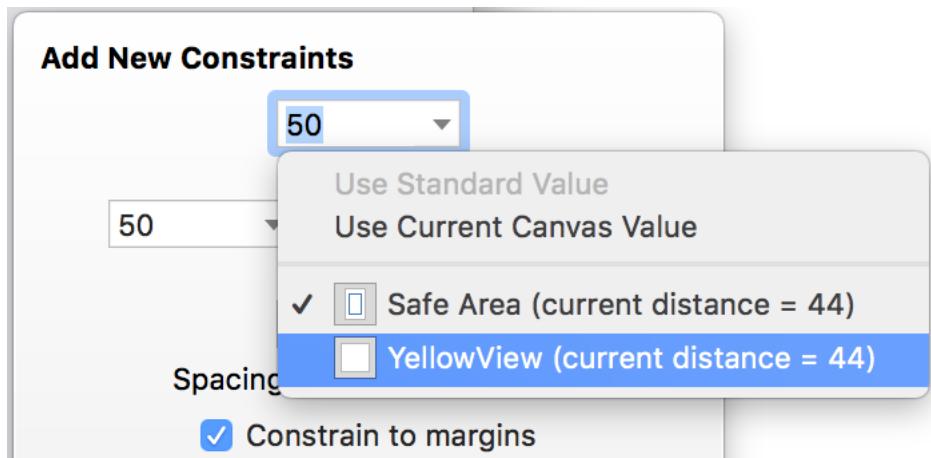
Typing a number into one of the boxes turns the red dotted I-bar solid. You can also click directly on the I-bar to turn the constraint on or off. Use the button at the bottom of the tool to add the constraints.

Each of the spacing boxes has a drop-down menu where you can choose between the possible neighbor views. When creating spacing between two child views you can also choose to use the “standard” spacing.



The constrain to margins checkbox is a little confusing. When you create a constraint to a view controller’s root view Interface Builder uses the safe area by default and the checkbox has no effect.

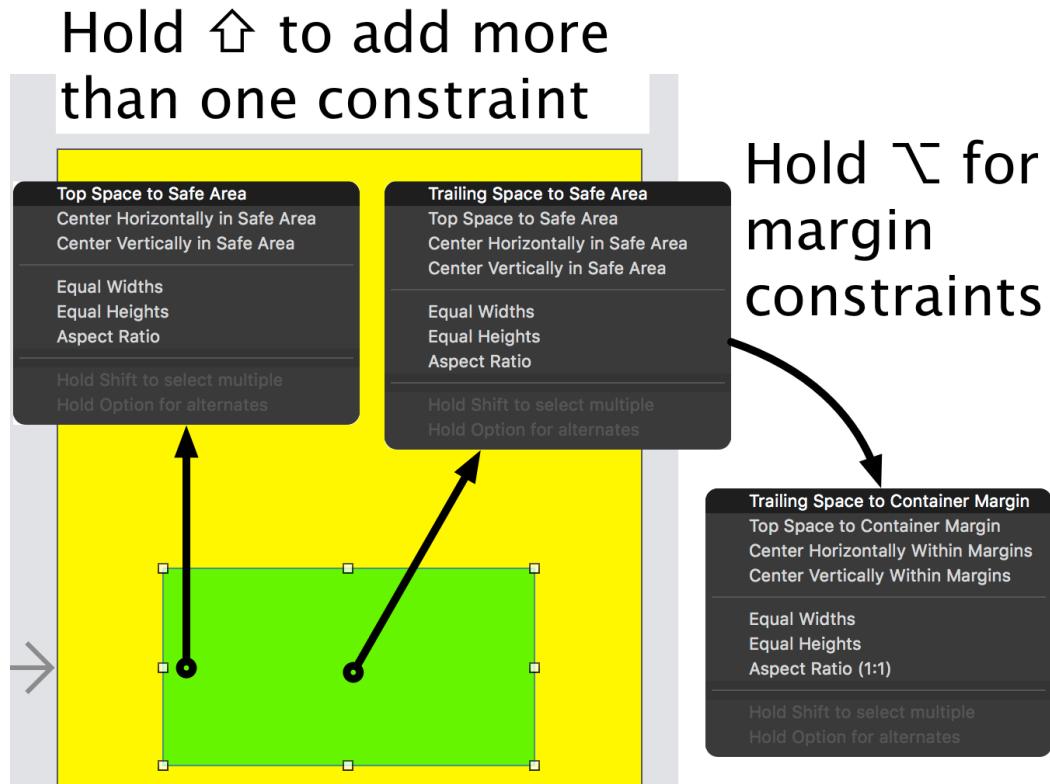
If you want to create a constraint with the margin or edge of the root view you need to use the drop-down menu and change the selection from the safe area to the view:



The constraint is then with the edge or margin of the view depending on how you set the constrain to margins checkbox. If you want to create a constraint to the edge of a view make sure you deselect the checkbox before changing to the view in the drop-down menu. Any time you select or deselect the checkbox, it resets all the drop-down menus back to using the safe area. Painful.

Control-Dragging In The Canvas

You can quickly create a constraint in the Interface Builder canvas by control-dragging within an item or between two items.



Interface Builder shows you a menu with constraint options based on the direction that you drag (horizontally, vertically or diagonally). Hold down the Option key to switch to margin-based constraints. Hold down the Shift key to select more than one constraint to add.

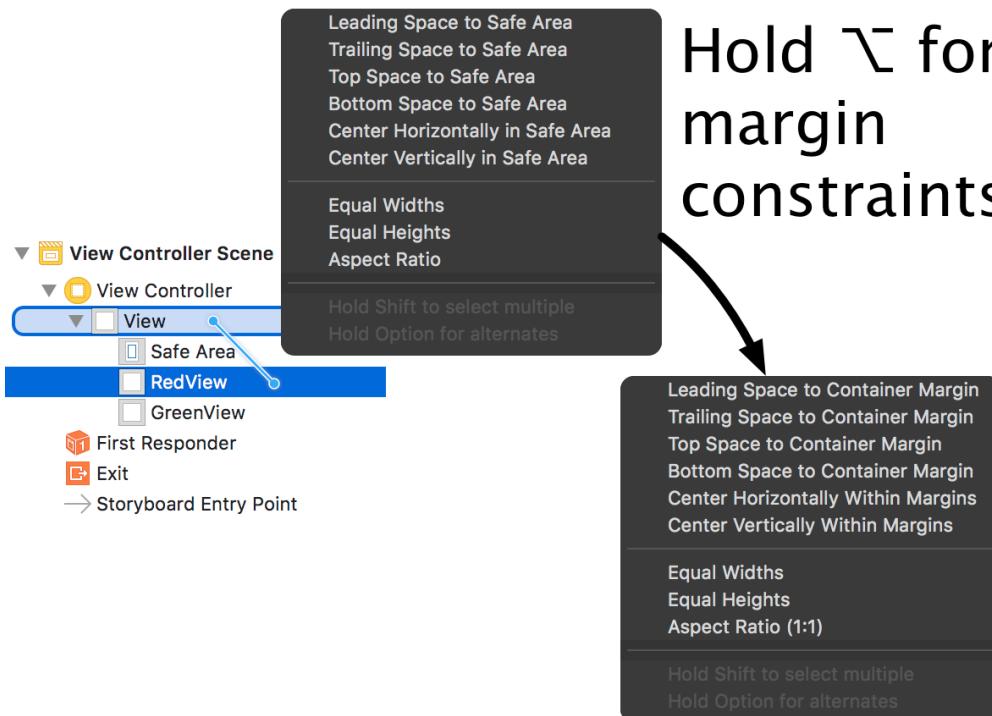


Xcode 10.1 changes control-dragging in the canvas to show constraints for all four directions not just the ones for the direction you drag.

Control-Dragging In The Document Outline

Control-dragging in the canvas is quick, but I often find it easier to use the document outline when you have several views on screen. Click on the first view and then control-drag to the second view. The menu shows you valid constraints between the two views. This might be to the safe area for a root view, horizontal and vertical spacing for sibling views or constraints to the container view edges for a child view.

Hold \uparrow to add more than one constraint



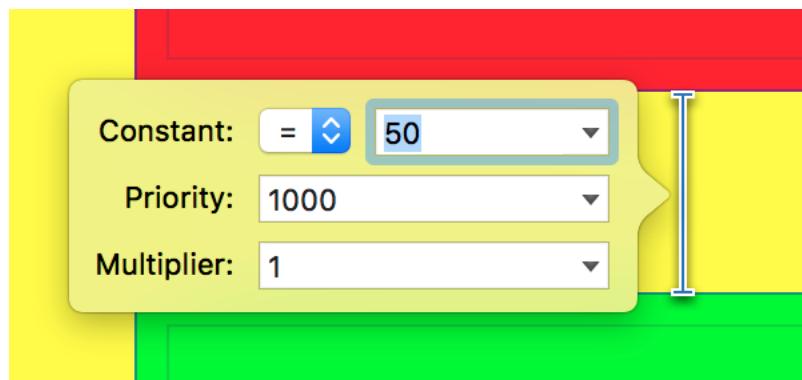
As with the canvas menu, hold the **Option** key to switch to margin-based constraints and hold the **Shift** key to add more than one constraint.

When you create a leading, trailing, top or bottom spacing constraint to the root view you can choose between the safe area or margin of the root view. If you want the constraint to go to the edges of the root view, you need to use the attribute inspector to edit the constraint and switch from the safe area to the superview.

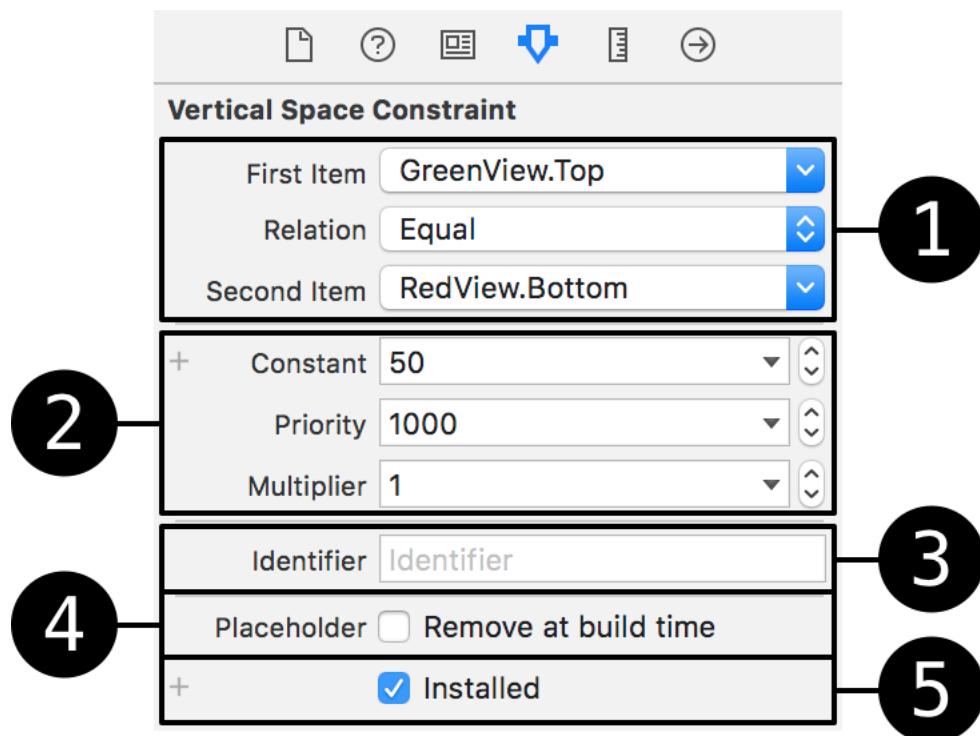
Editing A Constraint

As with all things Interface Builder, there's more than one way to edit a constraint after you have created it.

Double-clicking on a constraint in the canvas shows a popup that allows you to change the constraint relation, the constant value, the priority, and the multiplier:

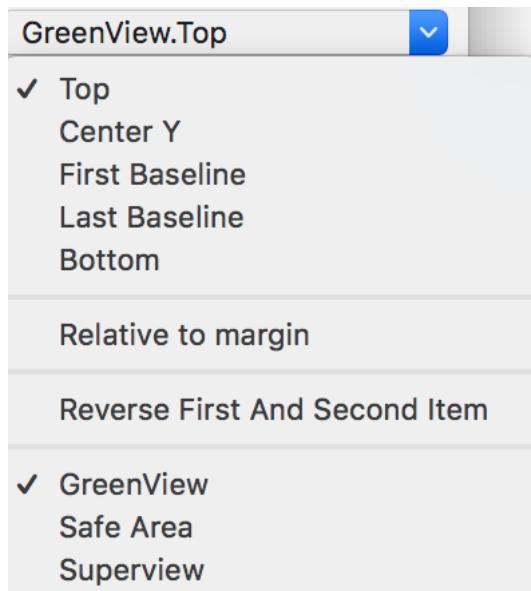


You cannot change the attributes involved in the constraint this way. For that you need to select the constraint in the canvas or document outline and then use the attributes or size inspectors:



Let's take a closer look at what you can do with the inspector:

1. The first section shows you the item(s) the constraint is using and allows you to change the relation. The drop-down menus for the first and second items allow you to change the attribute used for the constraint:



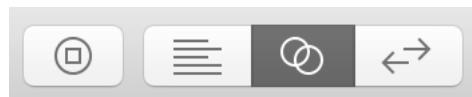
You can switch between using the margin or edge of the item, reverse the items or switch the item to the superview or safe area.

2. Change the constant, priority or multiplier. Use the **[+]** to the left of the constant to add a trait variation. We'll see how to use trait variations when we look at [Size Classes](#).
3. Add an identifier for this constraint that shows up in logs. We'll see how this helps when debugging constraints when we look at [Adding Identifiers To Views And Constraints](#).
4. Interface Builder doesn't include placeholder constraints in your built app. You can use them to avoid Interface Builder warnings for missing constraints you want to add at runtime.
5. Interface Builder installs constraints you create by default. Uncheck this box for constraints you want to activate at runtime or only install for some trait variations. We'll see how to use this when we look at [Size Classes](#).

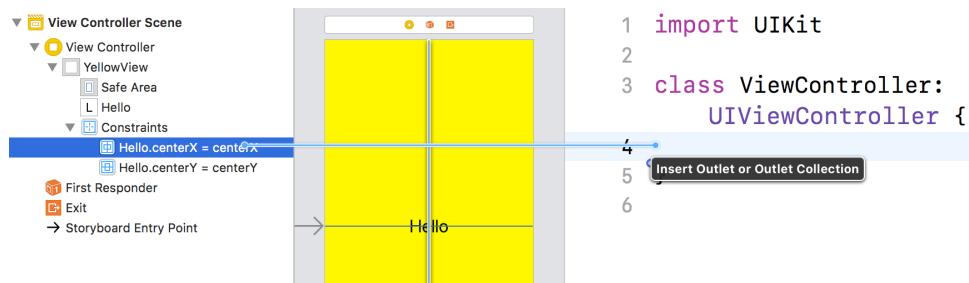
Creating Outlets For Constraints

You may already be creating outlets in a view controller to allow you to access views created in Interface Builder. You can use the same technique to allow you to access constraints:

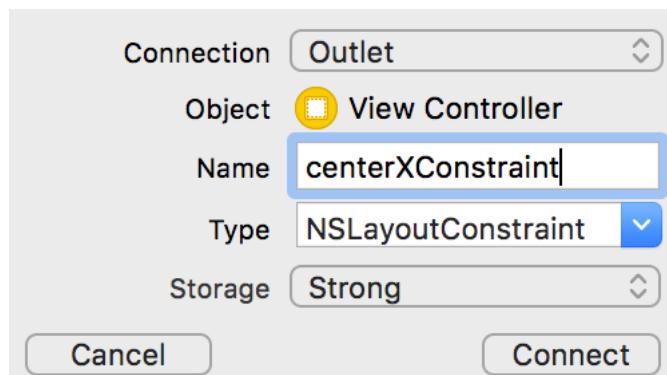
1. Use the assistant editor to show the view controller alongside its view in the Interface Builder canvas:



2. Find the constraint in the document outline on the left and control-drag from the constraint into the view controller on the right:



3. In the pop-up dialog that appears enter a name for the constraint and click "Connect":

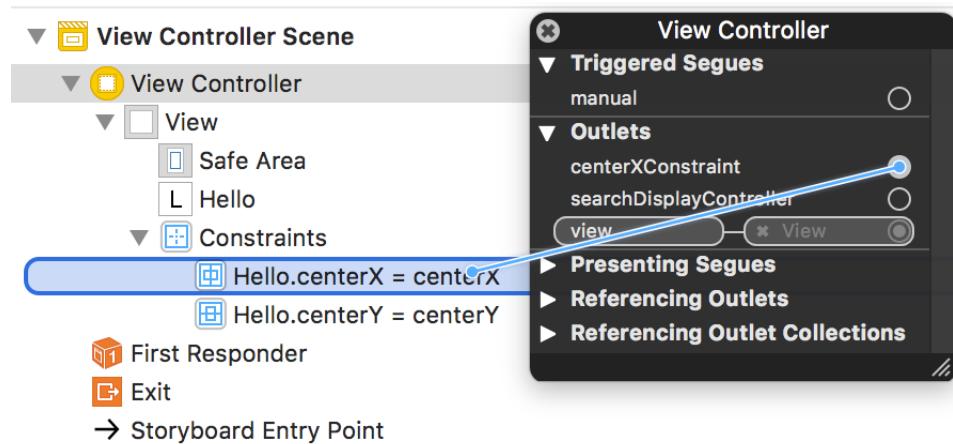


4. You should now have connected the `IBOutlet` property in your view controller to the constraint created in Interface Builder.

```
3 class ViewController: UIViewController {
    @IBOutlet var centerXConstraint: NSLayoutConstraint!
```

Note the solid circle in the margin that shows this is a connected outlet.

5. If you don't want to use the assistant editor, you can also create the property yourself in the view controller. Make sure to mark it with `@IBOutlet` so Interface Builder can find it. To connect it to the constraint right-click on the view controller in the document outline and drag from the outlet to the constraint:



Once created there are limited changes you can make to a constraint in your code. You can activate or deactivate it, change the priority, or change the constant:

```
// Deactivate a constraint
centerXConstraint.isActive = false

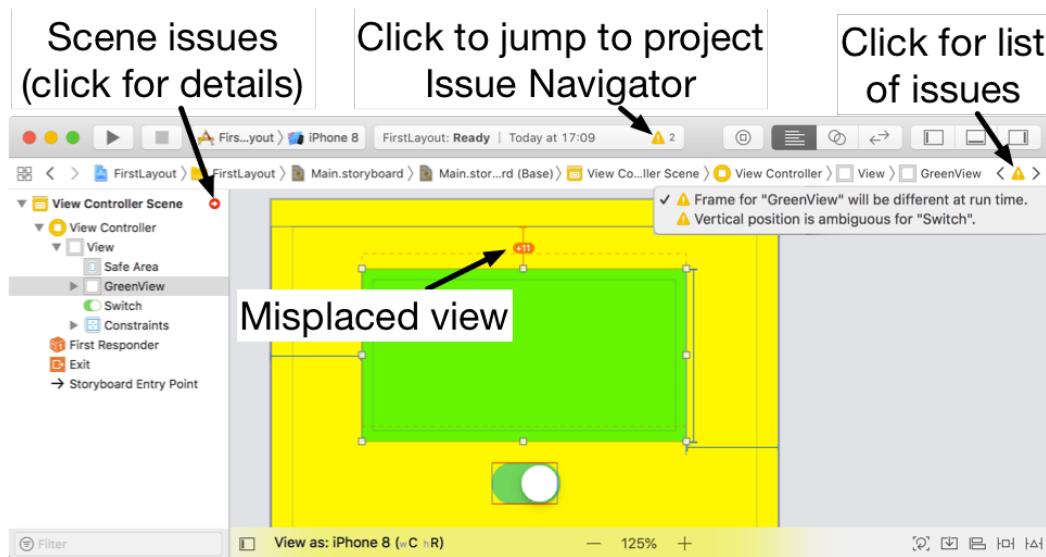
// Change priority
centerXConstraint.priority = .defaultLow

// Change constant value
centerXConstraint.constant = 50.0
```

You cannot change the items or attributes involved in the constraint, the relation or the multiplier.

Viewing Layout Warnings And Errors

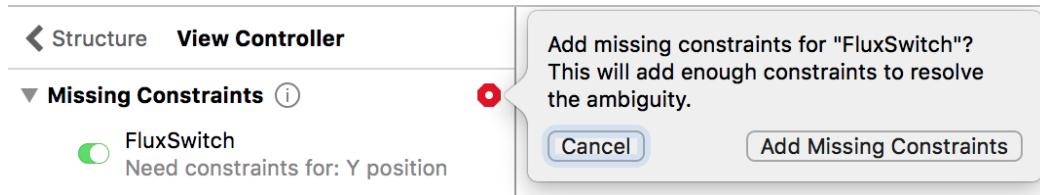
Interface Builder shows many different Auto Layout warnings and errors. These appear as a yellow warning or red error indicator in the Xcode status area, on the Interface Builder canvas and in the document outline.



Clicking on the indicator in the document outline takes you to a more detailed explanation of the issues for a scene:

Missing Constraints

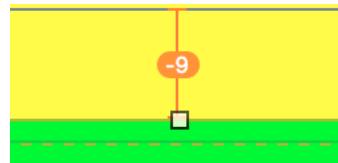
If you have not added enough constraints to fix the size and position of a view Interface Builder shows the “Missing Constraints” error in the document outline. The detailed explanation tells you what you’re missing and offers to add the missing constraints for you:



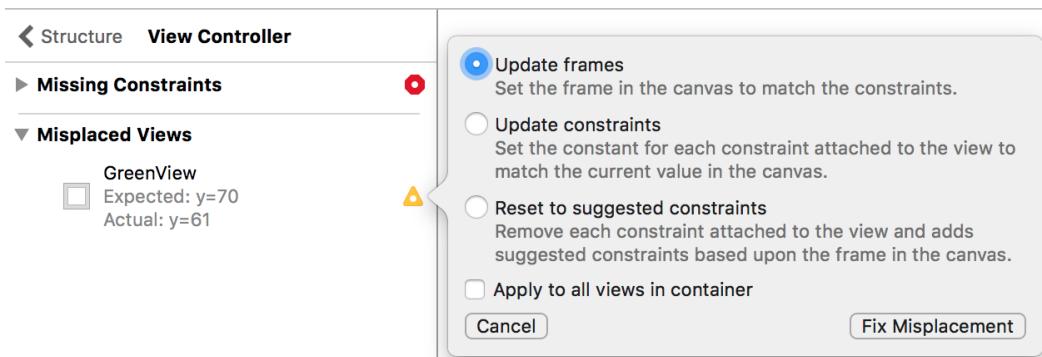
I recommend you figure out for yourself what constraints to add rather than trust Xcode to know what you want.

Misplaced Views

Interface Builder warns you about misplaced views if the frame of a view in the Interface Builder canvas is different from the frame that Auto Layout produces at runtime. This can easily happen when you drag views into the canvas and add constraints without worrying about the final position of the view. The canvas shows the warnings with the offset in a bubble:



The detailed explanation in the document outline shows you the actual and expected values for the view. You can update the frames from the menu of suggested actions or update the constraints to match the actual position of the view.

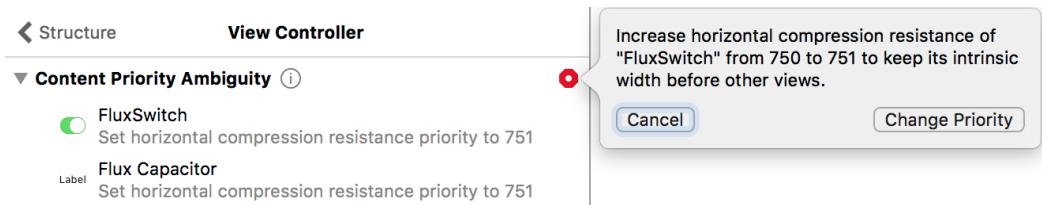


To quickly fix this warning use the update frames button in the toolbar at the bottom of the canvas. This button is enabled when you have a misplaced view or one of its superviews selected.



Content Priority Ambiguity

Auto Layout sometimes needs to stretch or squeeze a view to make it fit. If there are several views involved it chooses based on the relative content hugging or compression resistance priorities of the views. If the views have the same priority Auto Layout doesn't know what to do and creates a content priority ambiguity error:

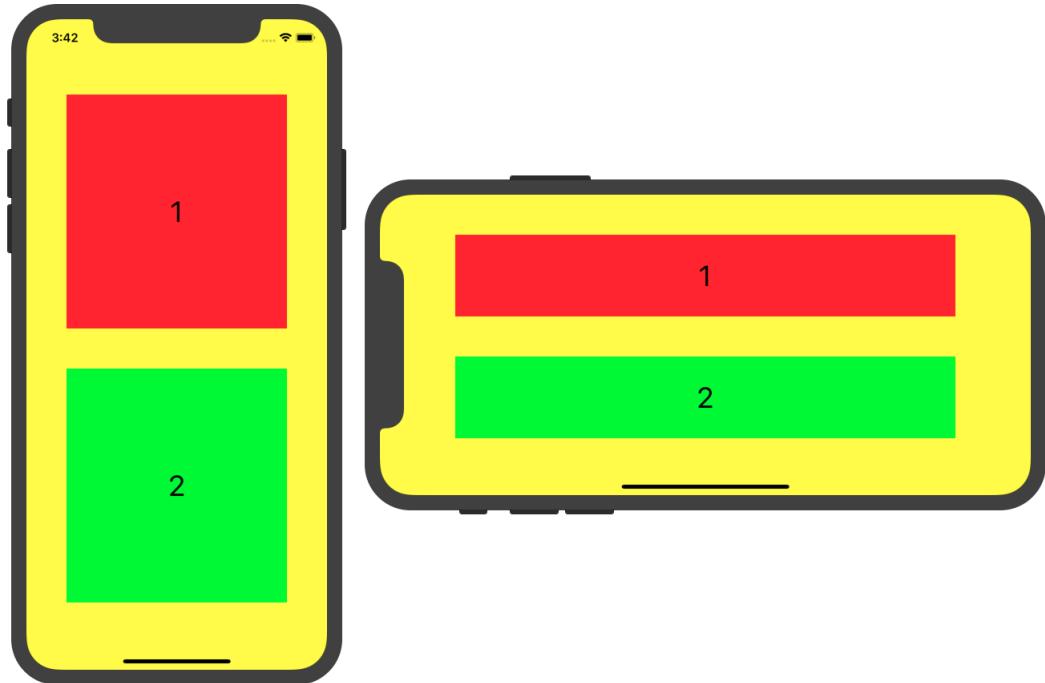


To fix the error, you need to change the priority of one of the views. Don't worry if you don't understand this yet. I explain it when we look at

Content Hugging And Compression Resistance.

Interface Builder Example

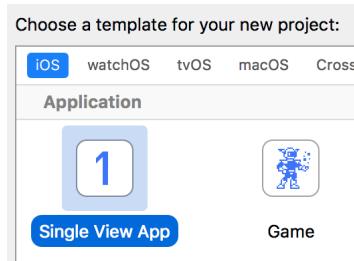
Time for a practical example of using Interface Builder. Here's our layout shown on the iPhone X in both portrait and landscape:



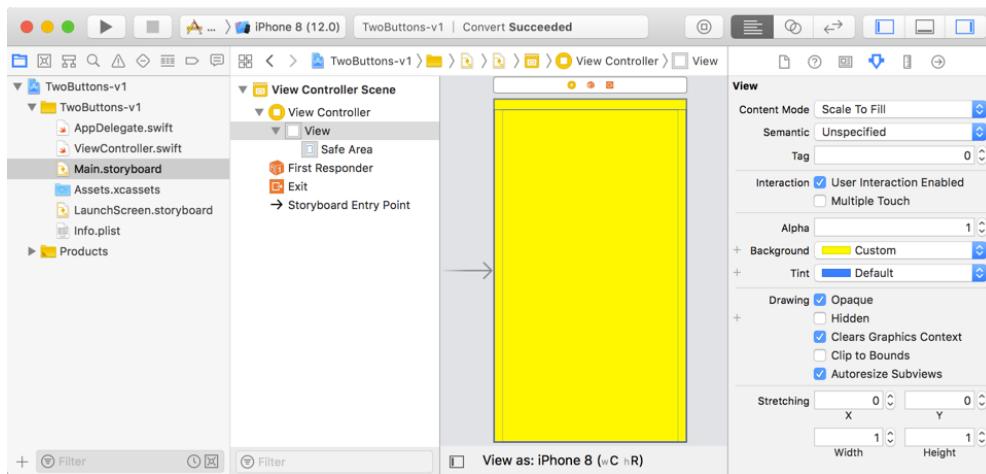
I spaced the red and green views 50 points from the device safe area (keeping them well inside the top camera housing and bottom home indicator areas). The spacing between the two views is also 50 points. The views have the same height, and each has a button in the center.

Getting Started

1. Open Xcode, start a new project (File > New > Project...), and choose the Single View App iOS application template (see sample code: [TwoButtons-v1](#)):



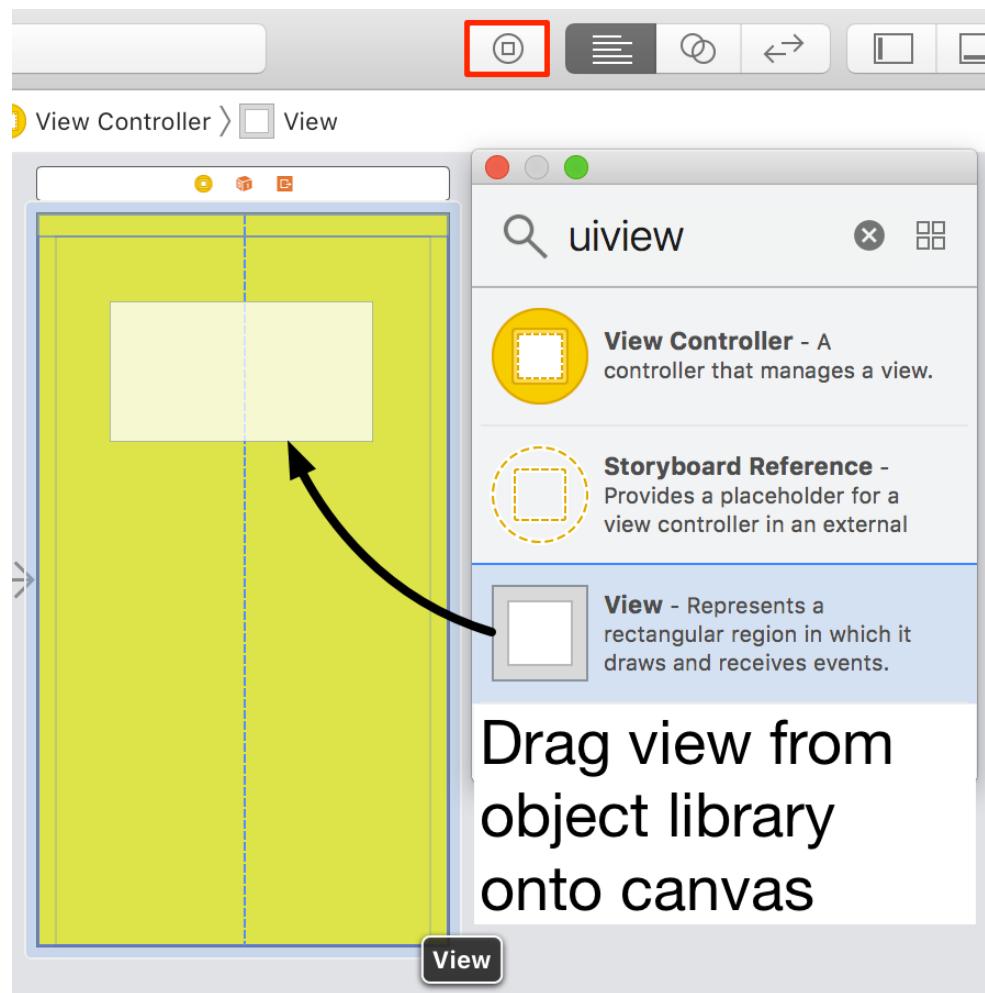
2. Name your project **TwoButtons** and save it.
3. Find the **Main.storyboard** in the Xcode navigator and click on it. The Xcode editor switches to Interface Builder and shows us the single view controller scene.



I changed the color of the root view to yellow to make it easier to see.

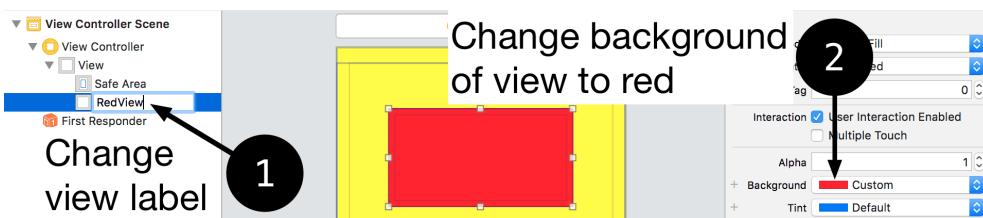
Adding Views

1. Find the **View** object in the Object Library and drag it onto the canvas over the root view. Drop the view somewhere near the top of the yellow root view:



Don't worry too much about where you put the view as Auto Layout works out the exact size and position for us.

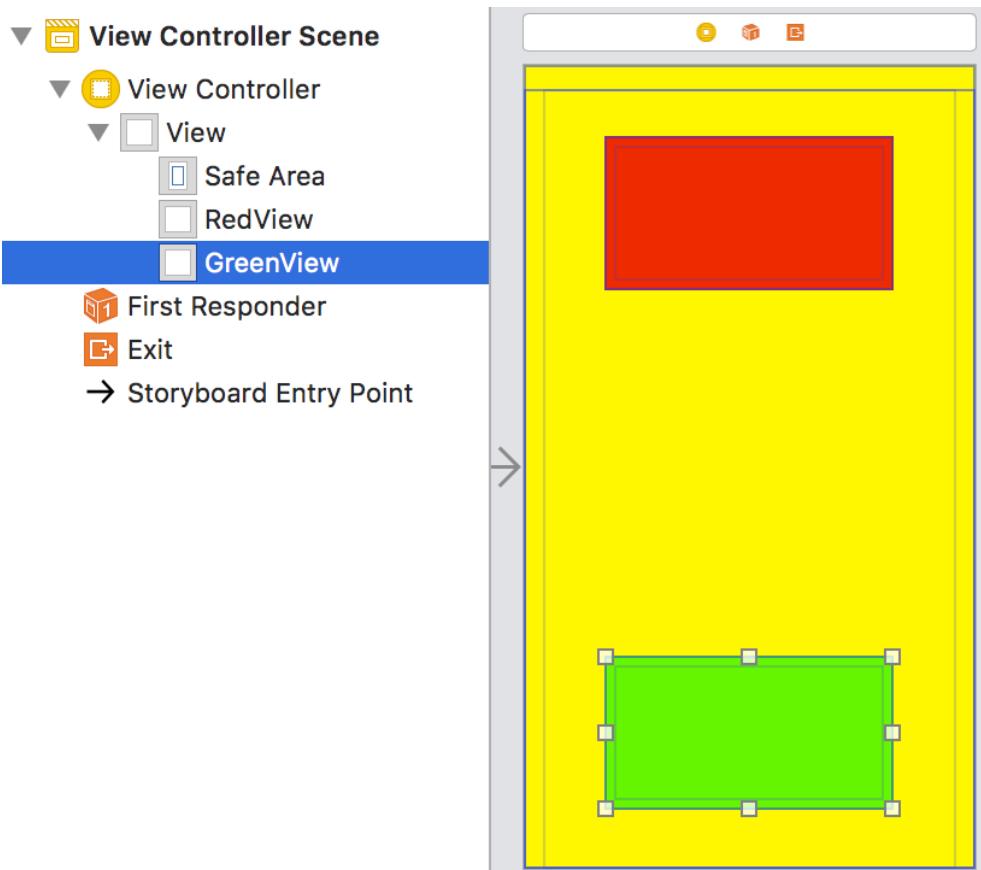
2. Select the new view in the document outline and edit the label. Change the generic "View" label to something more meaningful like "RedView". Then use the attributes inspector to change the background color of the view to red:



Note that changing the label doesn't affect the layout it just makes it easier to pick out the view in the document outline. You can also

change the label for a view using the identity inspector.

3. Follow the same steps to drop another view towards the bottom of the canvas. This time change the label in the document outline to "GreenView" and change the background color of the view to green:

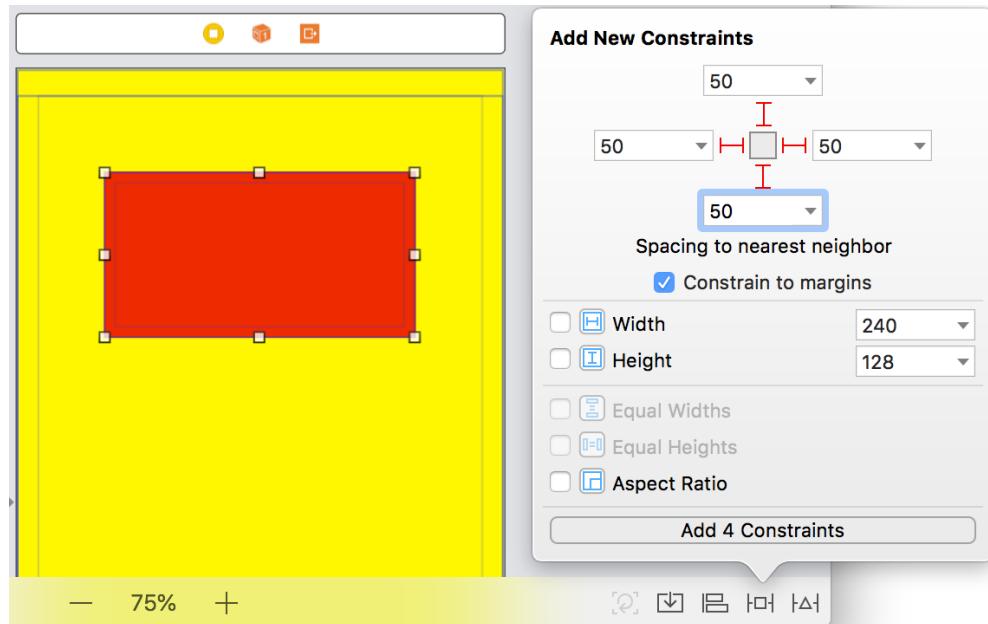


Adding Horizontal And Vertical Spacing

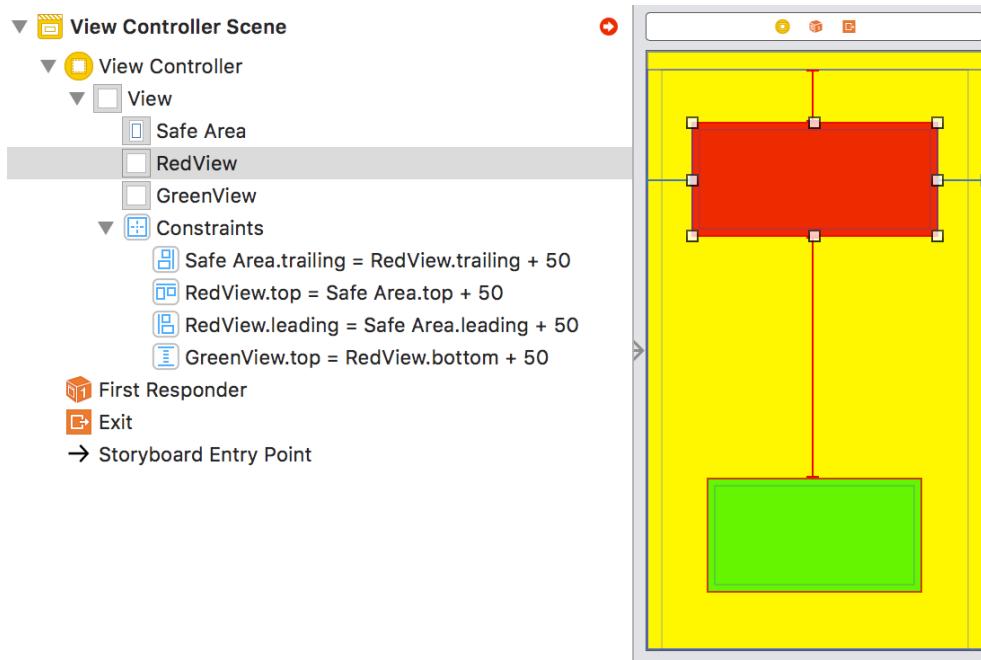
Let's add some constraints for our red view. We want to pin it to the leading, top and trailing edges of the safe area. We also need some vertical spacing between it and the top of the green view. The Add New Constraints tool is a good choice for this:



1. Select the red view in the canvas or document outline and then use the Add New Constraints tool to add the spacing for each direction:

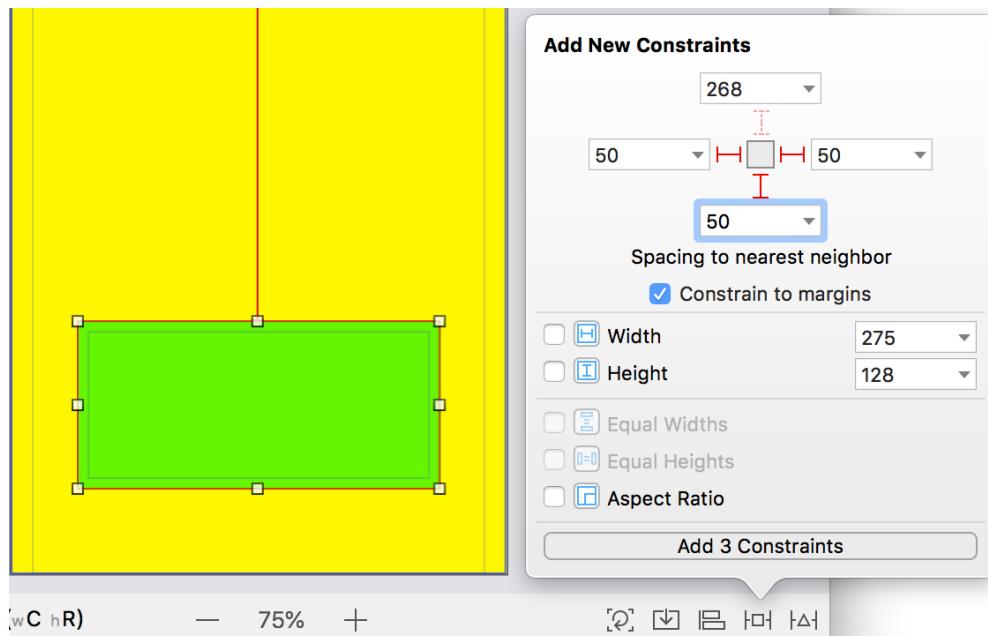


2. Enter 50 for the leading, top, trailing and bottom spacing constraints. The first three of these constraints are with the safe area of the root view. For the bottom constraint, the add tool is smart enough to default to using a constraint to the green view which is the nearest neighbor to the red view in that direction.
3. Click the button at the bottom of the tool labeled Add 4 Constraints. You should see the four newly created constraints in the document outline:

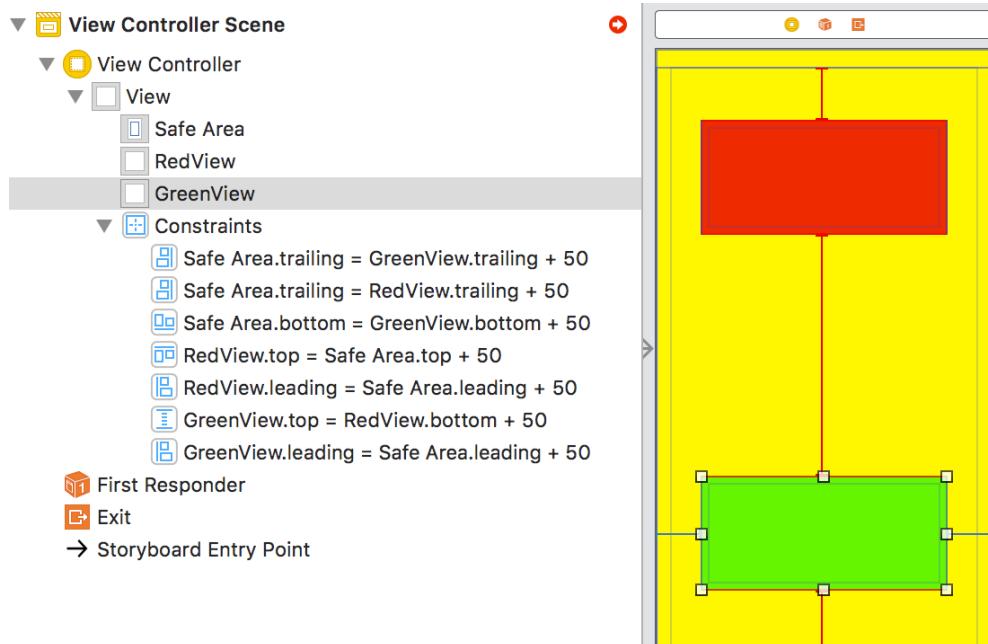


Interface Builder is probably showing you all sorts of errors and warnings at this point. Don't let that worry you. It just means we didn't add enough constraints yet.

4. We can use the same approach to pin the green view to the leading, bottom, and trailing edges. Click on the green view in the canvas or document outline and again use the Add New Constraints tool.
5. This time we only need three constraints as the green view already has a vertical spacing constraint to the red view. Enter 50 for the leading, bottom and trailing space constraints and click Add 3 Constraints:



- After adding those constraints, your layout in the Interface Builder canvas probably looks something like this:

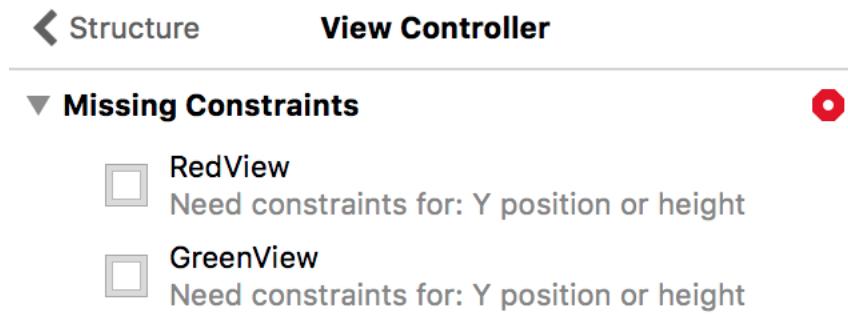


The exact appearance depends on where you put the two views but notice that Interface Builder is showing an error in the document outline. What constraint do you think we are missing? Interface Builder is also showing the vertical constraints in red. A hint that

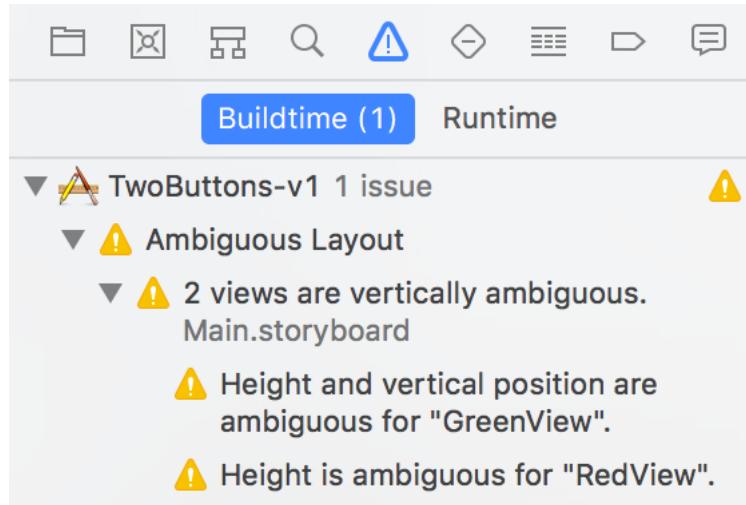
there we are missing a vertical constraint.

Equal Heights

If you're not sure what constraint is missing try clicking on the red error indicator in the document outline to see the details:

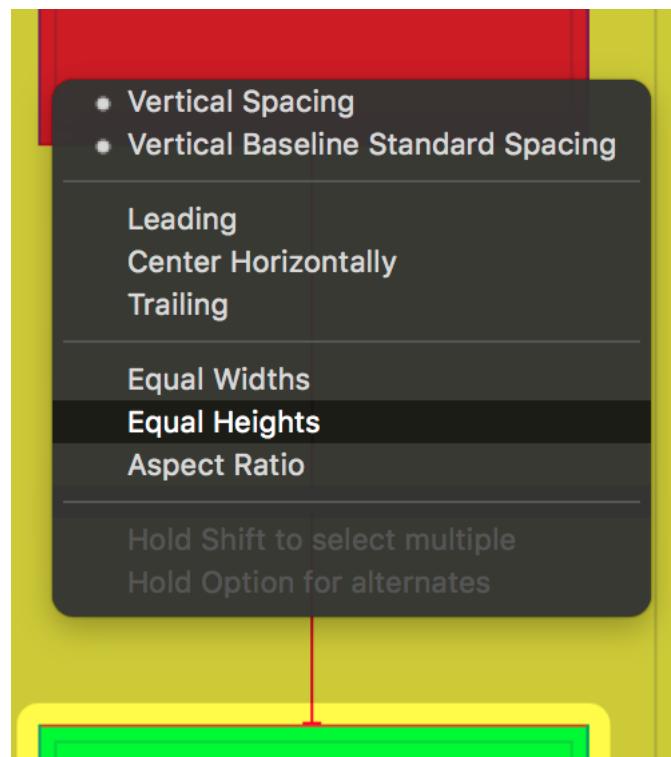


The Xcode issue navigator can also provide some additional clues.



We have not told Auto Layout about the heights of the red and green views. The solution is to add a constraint that gives the red and green views equal height:

1. Control-drag from the red view to the green view (or vice-versa) in the canvas or document outline and choose Equal Heights from the pop-up menu:



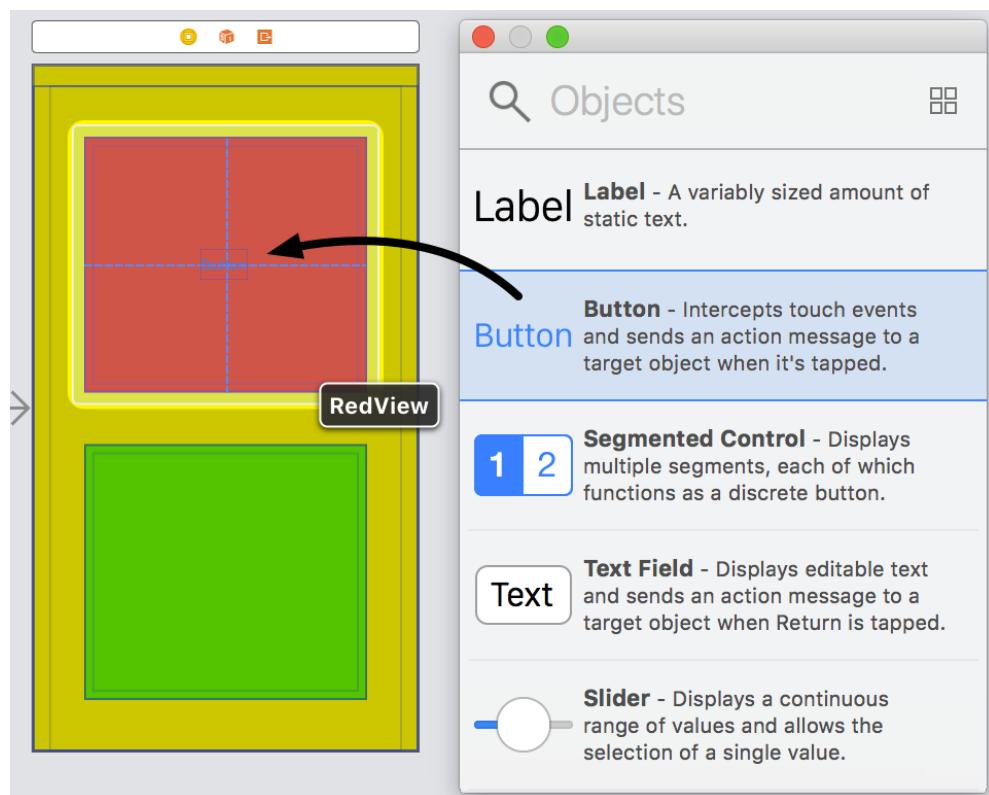
- With the equal heights constraint added the errors disappear and our red and green views are now where we want them:



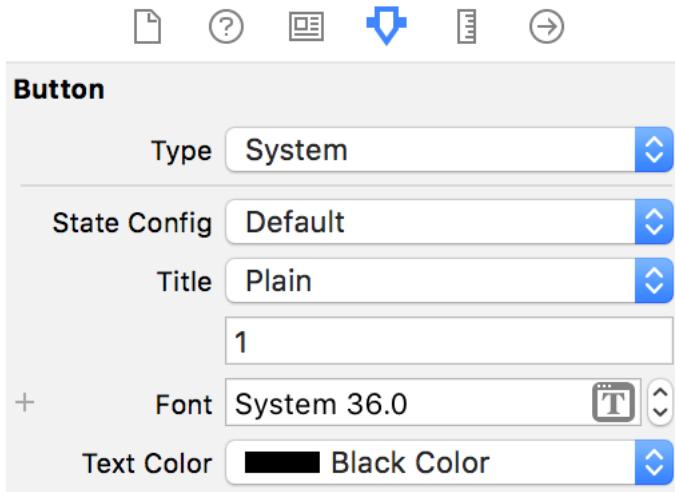
Centering The Buttons

To finish add the buttons to the red and green views:

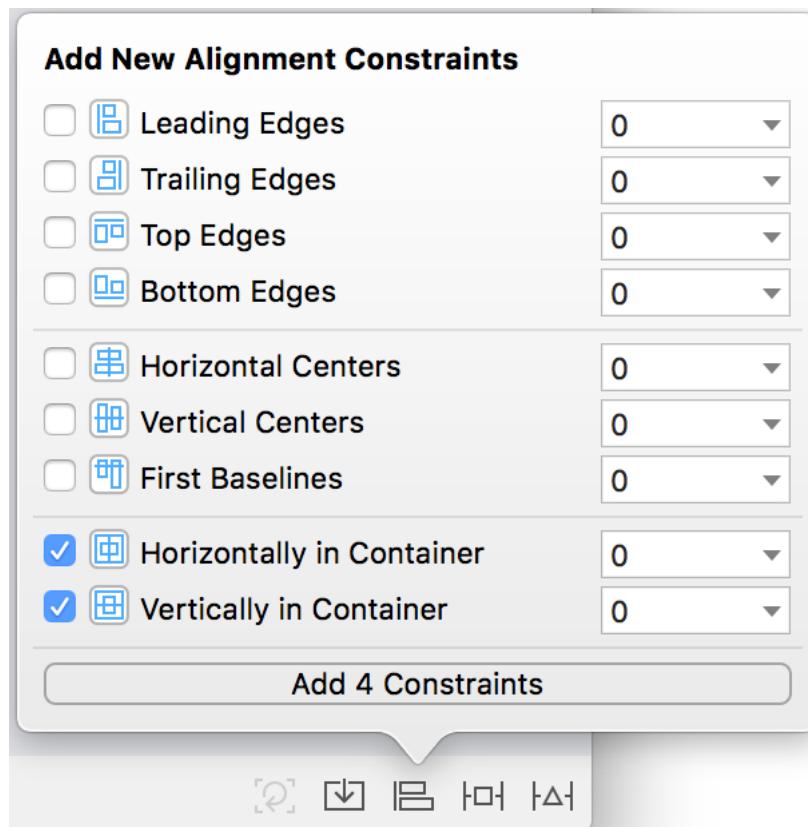
1. Drag two buttons from the object library and drop them in the centers of the red and green views. You can use the guides to help center the buttons but as before don't worry too much about getting it perfect:



2. Use the attributes inspector for each button to change the button text. I also increased the font size to 36pt and changed the text color to make it easier to see:



3. To center the buttons let's use the align tool this time. Select both buttons. I find that easiest to do by command-clicking on each button in the document outline. Then open the align tool in the toolbar at the bottom of the Interface Builder canvas:



Check the Horizontally in Container and Vertically in Container options and then click on Add 4 Constraints. If the button only shows 2 constraints make sure you have both buttons selected.

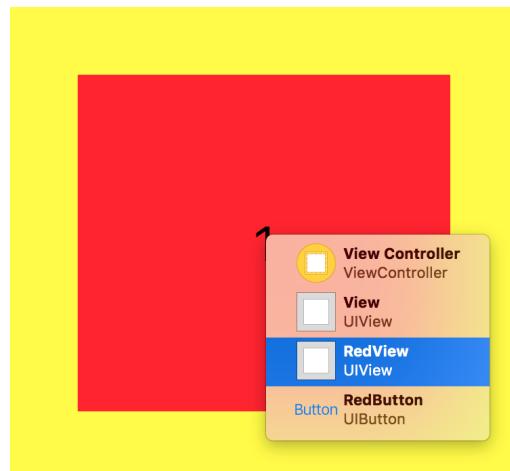
Try It Out

Try running the layout on the simulator for different devices to see how the views respond. Rotate the device to landscape to check that everything is working as expected.

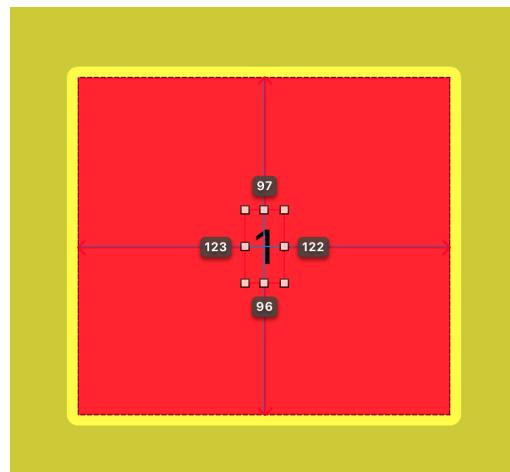
Interface Builder Tips And Tricks

Interface Builder often has many ways to do something. This flexibility is great, but I find it a source of confusion. Here are some tips and tricks to make you an Interface Builder expert. Don't feel like you have to know them all. Use what works for you and ignore the rest.

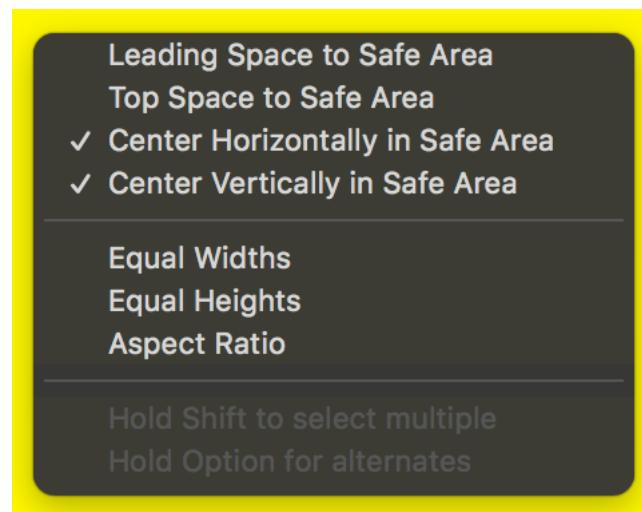
1. To select an object when it's behind a stack of other items hold down the **Ctrl + Shift** keys and then click over the object. Select the item you want from the popup menu showing the full view hierarchy at the point where you clicked.



2. Click on a view in the canvas to select it and then hold down the **Option** key. Move the mouse pointer over other views in the scene to see the distances between the views:



3. To quickly copy an object in the canvas, hold down the Option key and then click on and drag the object.
4. When adjusting the position of a view in the canvas, the arrow keys move the view one point at a time. Hold the Shift key to move by five points at a time.
5. When creating constraints in the canvas or document outline use the Shift key to add multiple constraints:



Use the Option key for alternate constraints. Useful when you want the margins instead of the safe area or a 1:1 aspect ratio:

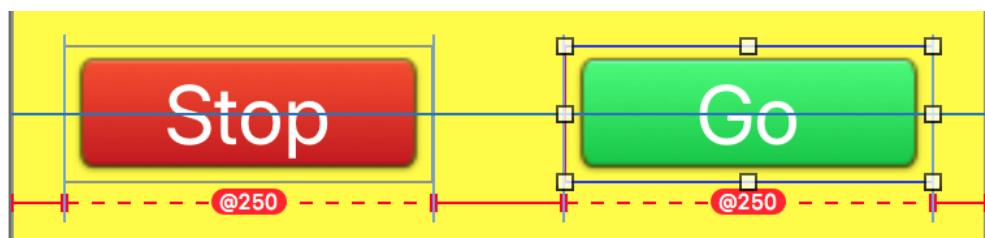


6. There are some configuration options for the canvas in the Xcode Editor menu:

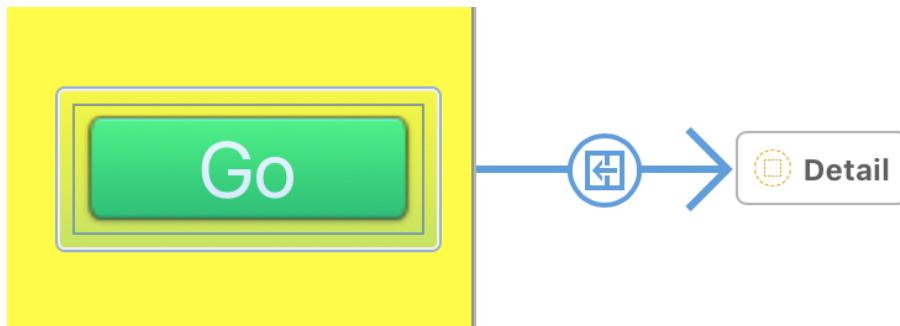
- ✓ Show Constraints
- ✓ Show Intrinsic Size Constraints Contributing To Ambiguity
 - Show Involved Views For Selected Constraints
- ✓ Show Layout Rectangles
 - Show Bounds Rectangles
- ✓ Show Device Bezels
- ✓ Show Placeholder Backgrounds

I like to turn on either Show Layout Rectangles or Show Bounds Rectangles to see the layout guides or bounds of views.

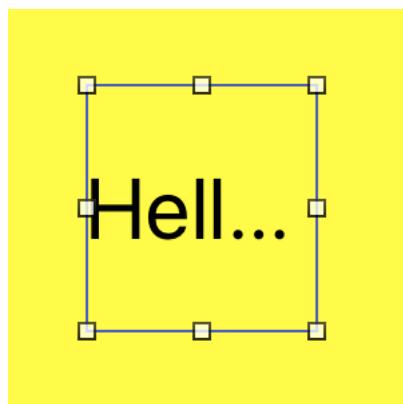
If you're fighting with constraint priorities, we'll see this when we look at [Layout Priorities And Content Size](#), try turning on Show Intrinsic Size Constraints Contributing To Ambiguity. It makes it easier to see which priorities you need to change to fix the problem:



7. Don't let your storyboards get too large. Interface Builder slows down, and if you're collaborating with other developers, it gets harder to avoid conflicts. Use Editor > Refactor To Storyboard to break it into smaller scenes with storyboard references.



8. Don't trust Interface Builder to Reset to Suggested Constraints. It rarely does what you want.
9. Use Cmd + = to resize a label, button, image, etc. to fit the content size. For example, this label is too small and high for the text:

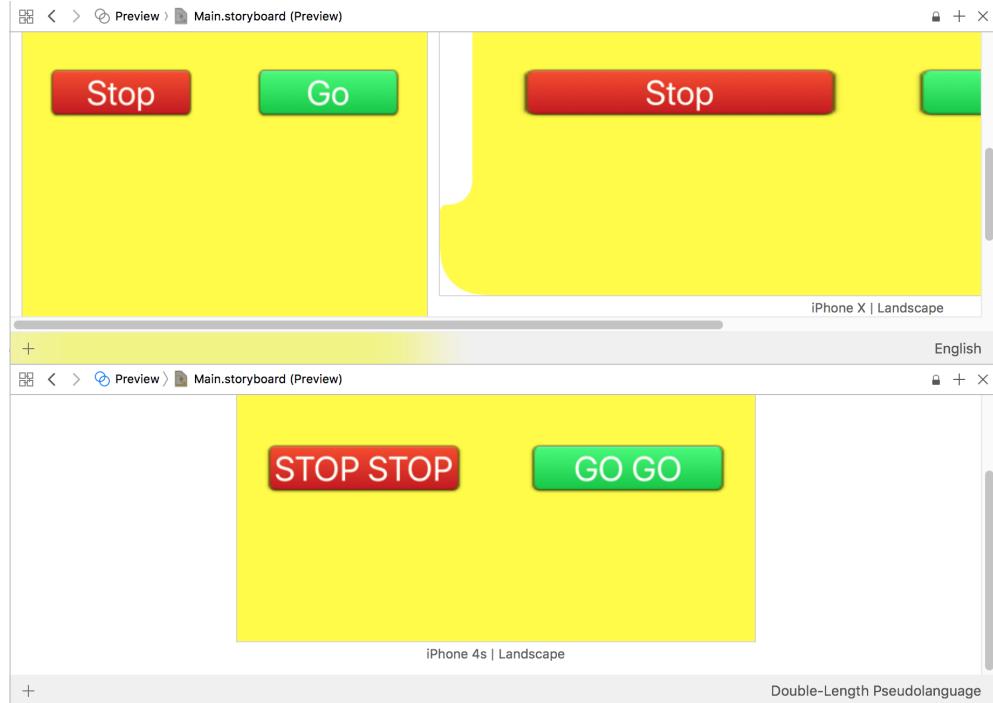


After using Cmd + = to size to fit contents:



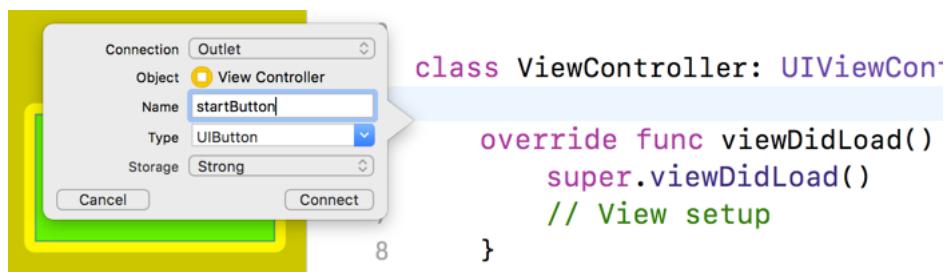
10. Don't forget you can preview your layout on different devices and orientations in Interface Builder with the assistant editor. Previewing is much faster than launching the simulator or running on a

device. Use the **[+]** in the bottom left corner of the assistant editor to add devices.

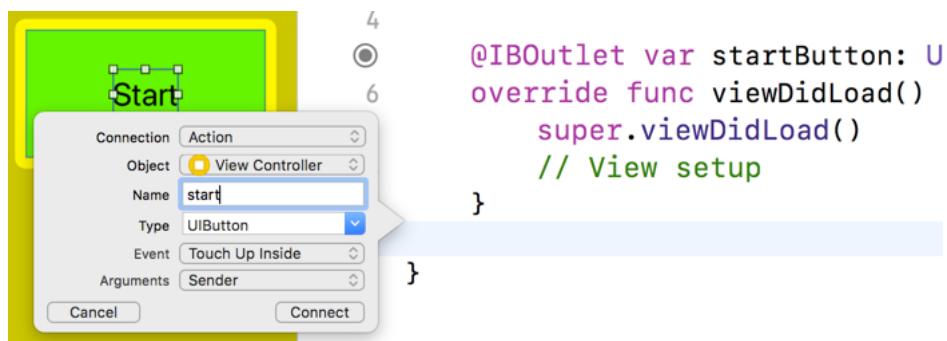


You can add multiple assistant editors with the **[+]** in the top right corner of the assistant editor. I like to use this to preview layouts with different localizations. Change the localization with the menu in the bottom right corner.

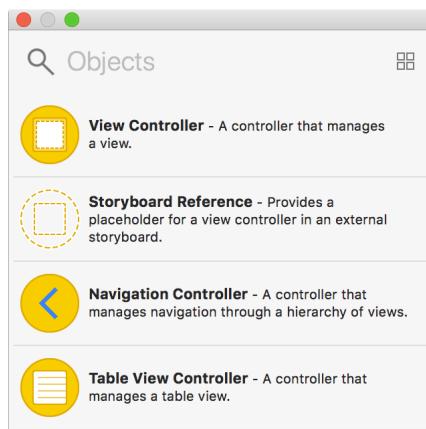
11. Control-dragging from a view in Interface Builder to a source file using the Assistant Editor defaults to creating an **IBOutlet** when you drag above the first method:



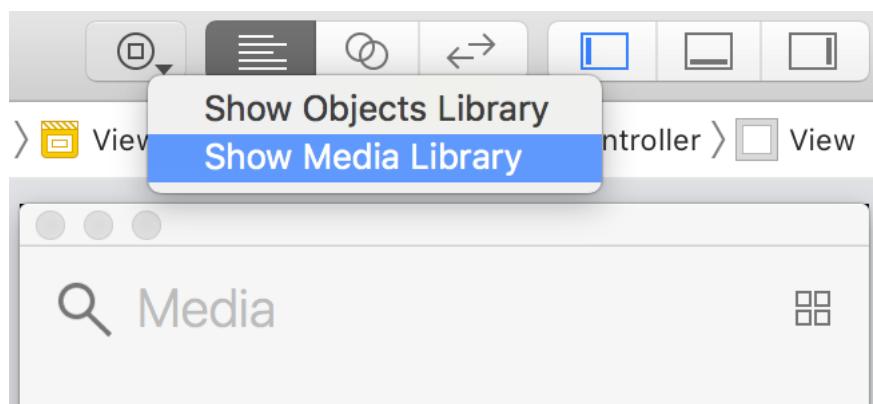
Dragging to below the first method defaults to creating an **IBAction**:



12. The object library automatically closes after you use it. To keep it open hold the Option key when clicking on the Library toolbar button. You can also hold the Option key when dragging an item from the library onto the canvas and it will keep the window open:



A long press on the Library toolbar button allows you to open the media library:



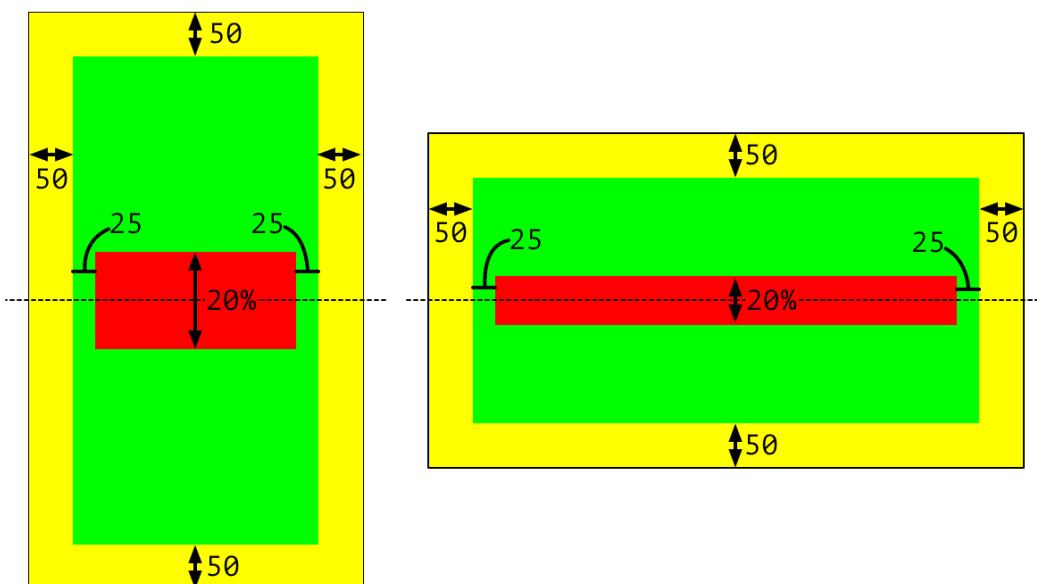
The menu is context-specific. When using the source code editor it shows the snippet editor.

Test Your Knowledge

Practice makes perfect. Try these Interface Builder challenges to test your progress. Experiment with the different ways of creating constraints to find out what works best for you. See the hints and tips if you get stuck.

Challenge 4.1 - Nested View Layout

You may recognize this first layout from back in chapter 2. The only difference is that the red view no longer has a fixed height.



The green view has an external spacing of 50 points and an internal spacing of 25 points to the red view. The red view is vertically centered in the green view and is 20% the height of the green view.

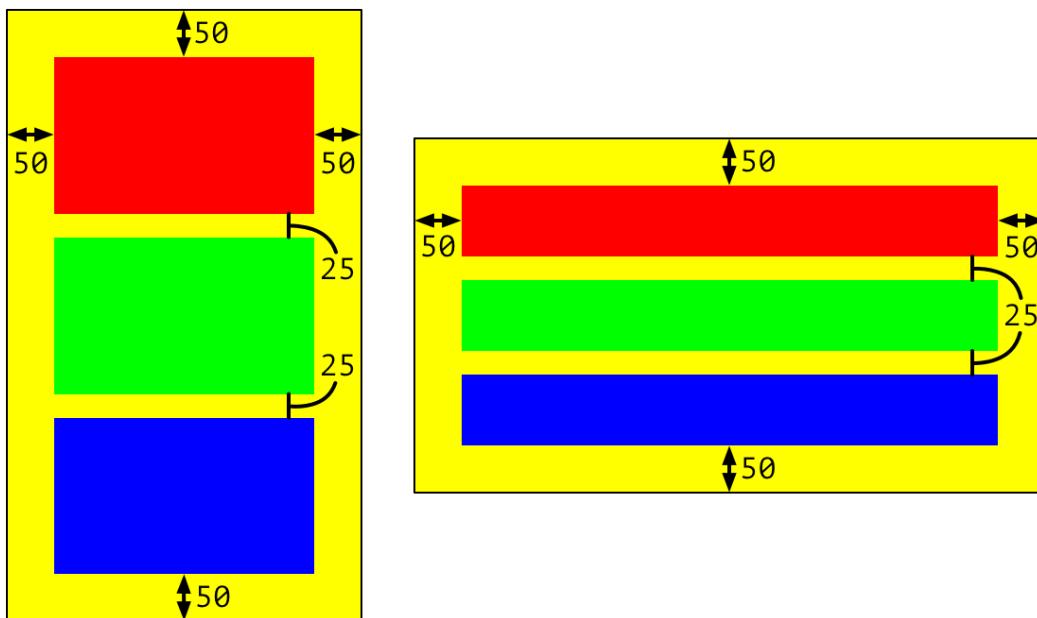
1. Build this layout using Interface Builder and Auto Layout (no code required). Use any of the techniques and tools you have seen in this chapter to create the constraints.
2. The 50 points of external spacing around the green view should be to the safe area of the root view.
3. Test the layout on various devices using the simulator and make sure that the spacing and sizing work in both portrait and landscape.

Hints And Tips

1. You can get into a mess with Interface Builder when the number of views and constraints increases. I find it helps to first roughly size and position the views close to their final layout before adding constraints.
2. With the views roughly positioned work from top to bottom using the Add New Constraints tool to set the spacing around each view.
3. To make the height of one view a percentage of the height of another view takes two steps. You first need to create an equal height constraint between the two views. Then edit the constraint to change the multiplier to the required proportion.
4. You should need no more than 8 constraints.

Challenge 4.2 - Sibling View Layout

This time instead of nesting the views lets make them siblings:



The three views are all spaced 50 points from the edges of the safe area with 25 points of vertical spacing between them. All three views must have the same height.

1. Build this layout using Interface Builder and Auto Layout (no code required).
2. Before you start adding constraints stop and think about how many you think you need? You have three views to layout so how many

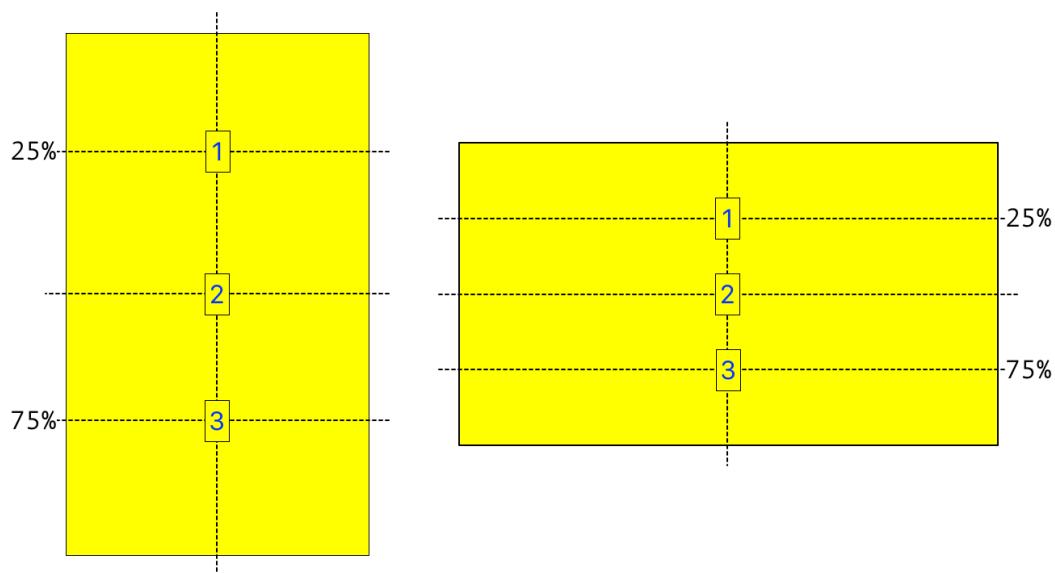
horizontal and vertical constraints do you think you need?

Hints And Tips

1. Don't forget to set the heights of the views.
2. You can do this layout with 12 constraints.

Challenge 4.3 - Proportional Centering

This layout has three vertically aligned buttons. I added some dotted guidelines to make it easier to see the positioning:



The three buttons are all centered horizontally. The top button is one-quarter of the way down the screen. Button "2" is in the middle and button "3" is three-quarters of the way down the screen. I'm using a 32 point system font for the button text.

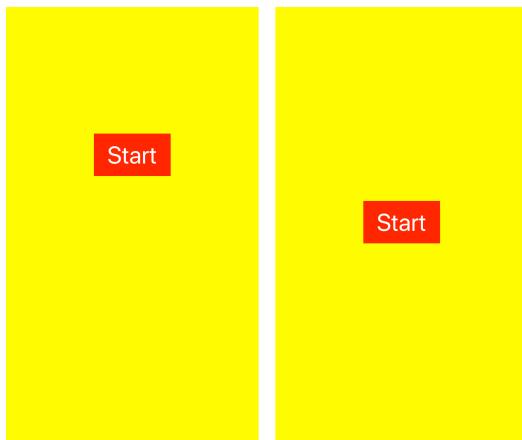
1. Build this layout using Interface Builder and Auto Layout (no code required).
2. Make the buttons centered within the safe area.
3. How many constraints do you think you need this time to constrain the three views? Are you sure? Be careful not to add more constraints than you need.

Hints And Tips

1. To center views, you can use the align tool or control-drag between views in the canvas or document outline. Note that the align tool centers views in the container, not the safe area. To center within the safe area try using the canvas or document outline.
2. Think about the relation of the center of each button to the center of the safe area.
3. The center of button “1” is halfway along the y-axis to the center of the safe area.
4. Create a constraint that centers a button in the superview safe area. Then change the multiplier of the constraint to shift the button position up or down. Button “1” is at 0.5 the center of the safe area. Button “2” is at the center. Button “3” is at 1.5 the center of the safe area.
5. This layout should only need six constraints.

Challenge 4.4 Changing Constraints

Practice creating an outlet for a constraint with this layout. When the view loads, the button is 100 points above the center as shown on the left below. After tapping the button, it should drop to the center as shown on the right:



1. Build this layout using Interface Builder.
2. When the view loads, the button should appear 100 points above the center.
3. In the action method for the button change the layout so that the button drops to the center.

Hints And Tips

1. This layout only needs two constraints to center the button horizontally and vertically.
2. Create an outlet in the view controller that connects to the vertical center constraint in Interface Builder.
3. You can set the initial 100 points offset in Interface Builder or the `viewDidLoad` method of the view controller.
4. Create an action method in the view controller that connects to the touch event of the button.
5. In the action method, change the constant of the center constraint to zero.
6. In a later chapter, we learn how to animate layout changes (see [Animating Constraints](#)).

Chapter 5

Creating Constraints In Code

You don't have to use Interface Builder to create your constraints. You can also do it programmatically. Apple gives you three choices when it comes to creating your constraints in code:

- Use the `NSLayoutConstraint` class.
- Use the Visual Format Language.
- Use Layout Anchors.

There are a large number of third-party libraries that wrap the Apple Auto Layout API in a more concise, readable syntax. I don't recommend learning a third-party Auto Layout library at the same time you learn Auto Layout. Wait until you feel comfortable with Auto Layout before exploring what's available. You may find that a few Swift extensions to shorten the worst of the boilerplate is all you need.

Activating and Deactivating Constraints

Apple introduced the concept of activating and deactivating constraints in iOS 8. Previously you directly added and removed constraints to and from an owning view. The old `addConstraint` and `removeConstraint` methods still exist, but you should not use them.

The big advantage of activation is that you no longer need to think about where to add your constraints (see [Who Owns A Constraint?](#)). Activating and deactivating a constraint takes care of adding and removing it from

the owning view for you.



It's a common mistake to activate a constraint between two subviews before you add them both to the view hierarchy. That causes a runtime error because there's no common superview to own the constraint. Remember to add both views to the same view hierarchy before activating the constraint.

When you create a constraint in code, it's always inactive by default. No view owns inactive constraints, so the layout engine doesn't see them. You activate a constraint by setting the `isActive` property.

```
widthConstraint.isActive = true
```

Setting `isActive` to `true` adds the constraint to the `constraints` array of the nearest common superview of the view(s) involved in the constraint. At that point, the constraint is visible to the layout engine when calculating the layout.

Setting `isActive` to `false` removes the constraint from the `constraints` array of the owning view and the constraint no longer affects the layout.



Removing a view from the view hierarchy also removes any constraints that involve that view or any of its subviews. Hiding a view doesn't remove the constraints so you can position views relative to a hidden view.

If you're creating and activating a group of constraints setting `isActive` for each constraint is a pain. It's easy to miss one when you have lots of constraints. Can you spot the mistake here?

```
redView.widthAnchor.constraint(equalTo:  
    greenView.widthAnchor).isActive = true  
redView.heightAnchor.constraint(equalTo:  
    greenView.heightAnchor) // Missed one  
redView.leadingAnchor.constraint(equalTo:  
    view.leadingAnchor).isActive = true
```

The `NSLayoutConstraint` class has a better way to activate a group of constraints. The convenience method `activate` takes an array of constraints to activate in a batch. This is both more concise and more

efficient than setting `isActive` on each constraint:

```
NSLayoutConstraint.activate([
    redView.widthAnchor.constraint(equalTo:
        greenView.widthAnchor),
    redView.heightAnchor.constraint(equalTo:
        greenView.heightAnchor),
    // other constraints...
])
```

There's a similar method to deactivate a group of constraints:

```
NSLayoutConstraint.deactivate(constraints)
```

Disabling The Autoresizing Mask

We looked at autoresizing masks when first introducing manual layout (see [Autoresizing](#)). A view's autoresizing mask controls how its size and position change when its superview resizes. It may surprise you to know that under the covers the mask is automatically translated into a set of Auto Layout constraints. If we're not careful, these automatically created constraints can conflict with our constraints.

When you create a view in Interface Builder, it defaults to using the autoresizing mask. You may not have noticed this as it disables the autoresizing mask for a view as soon as you add a constraint that involves that view. When you create a view in code, you need to disable the mask yourself if you want to add constraints for that view:

```
let myView = UIView()
myView.translatesAutoresizingMaskIntoConstraints = false
```

Forgetting this step is a frequent source of runtime errors when building your layout in code. You will soon get familiar with seeing this type of error in the Xcode console:

```
Frames[93001:7847043] [LayoutConstraints] Unable to
simultaneously satisfy constraints.
...
(
    "<NSAutoresizingMaskLayoutConstraint:0x60c0000888e0 h=&-&
     v=&-& UIView:0x7f846e3135f0.width == 168    (active)>",
...
)
```

It's a safe bet when you see `NSAutoresizingMaskLayoutConstraint` in the console log that you have forgotten to disable the translation of the autoresizing mask into constraints for one of your views.

Creating Constraints With `NSLayoutConstraint`

Let's start with what I think is the most painful way to create a constraint. Using the `NSLayoutConstraint` initializer:

```
NSLayoutConstraint(item: view1,  
                  attribute: attr1,  
                  relatedBy: relation,  
                  toItem: view2,  
                  attribute: attr2,  
                  multiplier: m,  
                  constant: c)
```

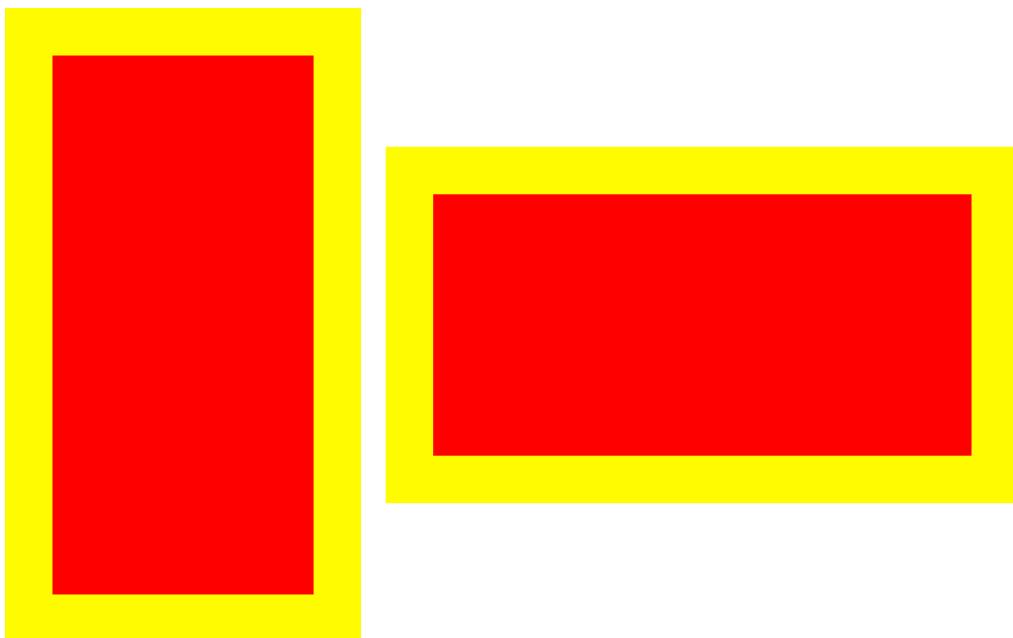
This returns an inactive constraint between two items (`view1` and `view2`). The constraint creates a relation between an attribute of the first view and an attribute of the second view. The relation can be `.equal`, `.lessThanOrEqual` or `.greaterThanOrEqual` so the constraint is one of these relationships:

```
view1.attr1 == view2.attr2 * m + c  
view1.attr1 <= view2.attr2 * m + c  
view1.attr1 >= view2.attr2 * m + c
```

If the constraint is for a single item use `nil` and `notAnAttribute` for the second item and attribute:

```
// redView.width == 150.0  
NSLayoutConstraint(item: redView, attribute: .width,  
                  relatedBy: .equal, toItem: nil, attribute: .notAnAttribute,  
                  multiplier: 1.0, constant: 150.0)
```

To see how this works let's build a layout with a single red subview. I spaced the view 50 points in from the screen edges on all sides (see sample code: [NSLayoutConstraint-v1](#)):



1. Open Xcode, start a new project (File > New > Project...) and either use my manual layout Xcode template or follow the instructions from [Removing The Main Storyboard](#) to set up the project without a storyboard.
2. In the view controller create a private method named `setupView` and call it from `viewDidLoad`:

```
// ViewController.swift
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        setupView()
    }

    private func setupView() {
        // View and constraint setup code
        // goes here
    }
}
```

3. Add a property for the red view:

```
private let redView: UIView = {
    let view = UIView()
    view.translatesAutoresizingMaskIntoConstraints = false
    view.backgroundColor = .red
}
```

```
    return view  
}()
```

Make sure to disable the autoresizing mask constraints so they don't conflict with our constraints.

4. While we are creating properties let's add one for the padding:

```
private let padding: CGFloat = 50.0
```

5. In the `setupView` method we set the background color of the root view and add the red view as a subview:

```
private func setupView() {  
    view.backgroundColor = .yellow  
    view.addSubview(redView)  
}
```

Add the subviews to the view hierarchy before creating the constraints. If the views involved in a constraint don't have a common ancestor in their view hierarchy to own the constraint, you will get a runtime crash.

6. I'm using the `NSLayoutConstraint.activate` convenience method to activate my constraints in a batch. In `setupView`:

```
NSLayoutConstraint.activate([  
    // Create each constraint here  
])
```

7. First the leading and trailing constraints:

```
NSLayoutConstraint.activate([  
    // redView.leadingAnchor == view.leadingAnchor + padding  
    NSLayoutConstraint(item: redView, attribute: .leading,  
        relatedBy: .equal, toItem: view, attribute: .leading,  
        multiplier: 1.0, constant: padding),  
  
    // view.trailingAnchor == redView.trailingAnchor + padding  
    NSLayoutConstraint(item: view, attribute: .trailing,  
        relatedBy: .equal, toItem: orangeView, attribute:  
        .trailing, multiplier: 1.0, constant: padding),
```

8. Next the top and bottom constraints:

```
// redView.top == view.top + padding  
NSLayoutConstraint(item: redView, attribute: .top,  
relatedBy: .equal, toItem: view, attribute: .top,  
multiplier: 1.0, constant: padding),  
  
// view.bottom = redView.bottom + padding  
NSLayoutConstraint(item: view, attribute: .bottom,  
relatedBy: .equal, toItem: redView, attribute: .bottom,  
multiplier: 1.0, constant: padding)  
])
```

We're creating a single view layout, but it still needs a lot of unreadable boilerplate code. There's also no compile-time warning if you get one of the constraints wrong. This constraint builds without error but causes a run-time crash. Can you spot why?

```
NSLayoutConstraint(item: view, attribute: .bottom, relatedBy:  
.equal, toItem: redView, attribute: .trailing, multiplier:  
1.0, constant: padding)
```

Luckily we have better and safer ways to write layout code these days.

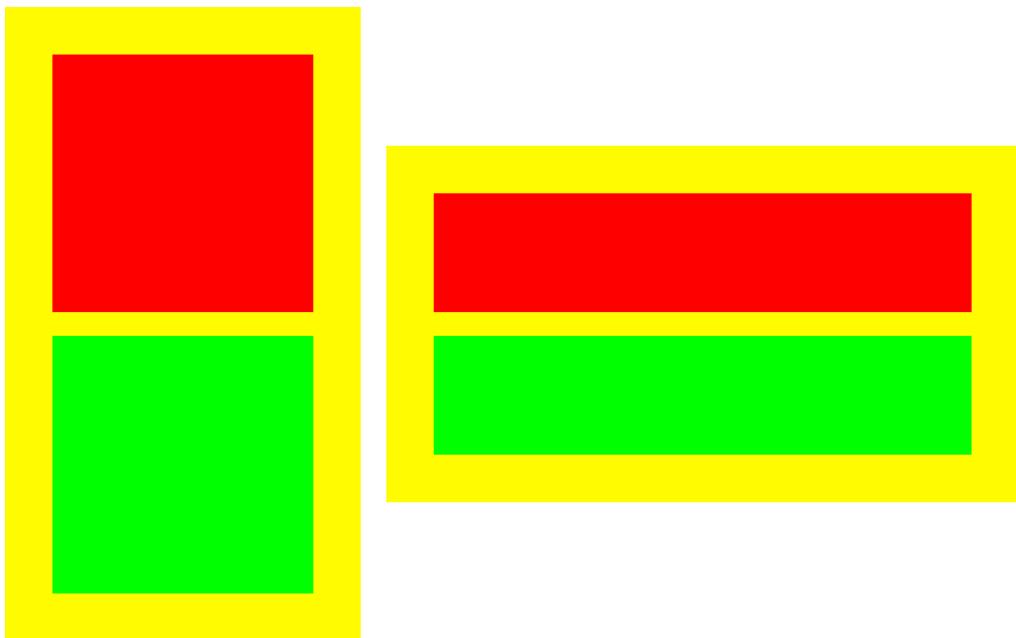
Visual Format Language

The Visual Format Language (VFL) lets you write constraints using an ASCII-art style format string. It allows you to create a collection of constraints more concisely and visually. I don't find it much of an improvement over the plain `NSLayoutConstraint` initializer, but you can decide for yourself.

To create constraints with VFL you use the `NSLayoutConstraint` class method `constraints(withVisualFormat:options:metrics:views:)`. It returns an array of constraints we can then activate:

```
NSLayoutConstraint.constraints(  
withVisualFormat format: String,  
options opts: NSLayoutConstraintOptions = [],  
metrics: [String : Any]?,  
views: [String: Any])
```

The VFL is best explained with an example so let's use it to add a green view to our layout:



There are 25 points of spacing between the views and 50 points of padding to the screen edges (see sample code: [VFL-v1](#)).

1. This time we need a red and a green view so let's use a small private `UIView` extension in our view controller to avoid repeating ourselves:

```
// ViewController.swift
private extension UIView {
    static func makeView(color: UIColor) -> UIView {
        let view = UIView()
        view.translatesAutoresizingMaskIntoConstraints = false
        view.backgroundColor = color
        return view
    }
}
```

2. Now in our view controller we create our two views and add them to the view hierarchy:

```
class ViewController: UIViewController {

    private let redView = UIView.makeView(color: .red)
    private let greenView = UIView.makeView(color: .green)

    override func viewDidLoad() {
        super.viewDidLoad()
        setupView()
    }
}
```

```

    }

    private func setupView() {
        view.backgroundColor = .yellow
        view.addSubview(redView)
        view.addSubview(greenView)

        // Add constraints here
    }
}

```

- To use VFL, you first build a dictionary of the views you want to use in the visual format string. The dictionary keys are the string values used in the format string, and the dictionary values are the views themselves. Our constraints need the red and green views, so we add them to a dictionary with suitable names (in `setupView`):

```

let views = [
    "redView" : redView,
    "greenView" : greenView
]

```

- Add any constants or “magic numbers” for our constraints to a metrics dictionary. We have 50 points of external padding and 25 points of spacing between the views:

```

let metrics = [
    "padding" : 50.0,
    "spacing" : 25.0
]

```

- With the bookkeeping done we can create the horizontal constraints that pin the red and green views to the leading and trailing edges of the superview:

```

let hRedConstraints =
    NSLayoutConstraint.constraints(withVisualFormat:
        "H:|- (padding) - [redView] - (padding) - |", options: [],
        metrics: metrics, views: views)

let hGreenConstraints =
    NSLayoutConstraint.constraints(withVisualFormat:
        "H:|- (padding) - [greenView] - (padding) - |", options: [],
        metrics: metrics, views: views)

```

Take a look at the format string: "H| - (padding) - [redView] -

(padding) - | ". The views are always in square brackets. The vertical bars (|) are the edges of the superview and the - is a spacing constraint. The constant for the spacing is in parentheses and comes from the metrics dictionary.

6. For the vertical constraints you prefix the format string with a V:. So to position the red view above the green view with spacing:

```
let vConstraints =  
    NSLayoutConstraint.constraints(withVisualFormat:  
        "V:|- (padding)- [redView(==greenView)]- (spacing)-  
        [greenView]- (padding)- |", options: [], metrics: metrics,  
        views: views)
```

This is a more complicated format string, but the only new feature is to set the height of the red view. The format `redView(==greenView)` makes the height of the red view equal to the green view.

7. Finally with all the constraints created we activate them in one go:

```
let constraints = hRedConstraints + hGreenConstraints +  
    vConstraints  
NSLayoutConstraint.activate(constraints)
```

I'm not a big fan of the VFL for creating constraints. It has some limitations, and Apple has not updated it in recent years. It's worth being aware of as the syntax shows up in the Xcode debug console but I don't recommend you use it for creating constraints.

Layout Anchors

A frequent complaint with Auto Layout is how verbose and unreadable the syntax is for programmatically creating constraints. Apple made a significant improvement with the introduction of layout anchors in iOS 9. From the class documentation:

The `NSLayoutAnchor` class is a factory class for creating `NSLayoutConstraint` objects using a fluent API. Use these constraints to programmatically define your layout using Auto Layout.

A `UIView` has layout anchors for each of the [Constraint Attributes](#) we saw back in chapter 3. Each layout anchor is a subclass of `NSLayoutAnchor`

with methods to directly create constraints to other layout anchors of the same type.

You don't use the `NSLayoutAnchor` class directly. Instead, you use one of its subclasses depending on whether you want to create a horizontal, vertical or size-based constraint:

Horizontal Constraints

Use layout anchors of type `NSLayoutXAxisAnchor` to create horizontal constraints:

- `centerXAnchor`
- `leadingAnchor` and `trailingAnchor`
- `leftAnchor` and `rightAnchor`

For example, to create a constraint that center aligns two views:

```
redView.centerXAnchor.constraint(equalTo:  
    greenView.centerXAnchor)
```

Note how you start with an anchor on one view and create the constraint to an anchor on another view.



Prefer the `leadingAnchor` and `trailingAnchor` over the `leftAnchor` and `rightAnchor`. The leading and trailing anchors are aware of Right-To-Left (RTL) languages and flip the interface when necessary.

Vertical Constraints

Use layout anchors of type `NSLayoutYAxisAnchor` to create vertical constraints:

- `centerYAnchor`
- `bottomAnchor` and `topAnchor`
- `firstBaselineAnchor` and `lastBaselineAnchor`

For example, to create a constraint that spaces the top of `greenView` 25 points below the bottom of `redView`:

```
greenView.topAnchor.constraint(equalTo: redView.bottomAnchor,  
    constant: 25)
```

Size Based Constraints

Use layout anchors of type `NSLayoutDimension` to create size-based constraints:

- `heightAnchor` and `widthAnchor`

For example, to create a constraint that fixes the width of a view to 50 points:

```
redView.widthAnchor.constraint(equalToConstant: 50.0)
```

To make the height of `redView` twice the height of `greenView`:

```
redView.heightAnchor.constraint(equalTo:  
    greenView.heightAnchor, multiplier: 2.0)
```

Creating Constraints With Layout Anchors

There's a full set of methods for creating constraints with layout anchors. Some example usage:

```
// anchor (==, >=, <=) otherAnchor  
anchor.constraint(equalTo:otherAnchor)  
anchor.constraint(greaterThanOrEqualTo:otherAnchor)  
anchor.constraint(lessThanOrEqualTo:otherAnchor)  
  
// anchor == otherAnchor + constant  
anchor.constraint(equalTo:otherAnchor constant:8.0)  
  
// dimensionAnchor == otherDimensionAnchor * multiplier  
widthAnchor.constraint(equalTo:otherWidthAnchor,  
    multiplier:2.0)  
  
// dimensionAnchor == otherDimensionAnchor * multiplier +  
// constant  
widthAnchor.constraint(equalTo:otherWidthAnchor,  
    multiplier:1.0, constant:20.0)
```

If you're using at least iOS 11 there are some extra convenience methods that use the standard system spacing instead of a constant. For horizontal constraints there are three methods for standard system spacing after an anchor:

```
constraint(equalToSystemSpacingAfter anchor:multiplier:)
```

```
constraint(greaterThanOrEqualToSystemSpacingAfter  
    anchor:multiplier:)  
constraint(lessThanOrEqualToSystemSpacingAfter  
    anchor:multiplier:)
```

The `multiplier` applies to the system spacing. For example, to horizontally space the `greenView` at twice the system spacing after the `redView`:

```
greenView.leadingAnchor.constraint(equalToSystemSpacingAfter:  
    redView.trailingAnchor, multiplier: 2.0)
```

A similar set of methods cover the vertical constraints creating standard spacing below an anchor. For example, a `blueView` at the standard spacing below the `redView`:

```
blueView.topAnchor.constraint(equalToSystemSpacingBelow:  
    redView.bottomAnchor, multiplier: 1.0)
```

You can only create constraints between anchors of the same type. The compiler checks this to prevent you from creating meaningless constraints. For example, creating a constraint between a leading anchor and a bottom anchor is an error:

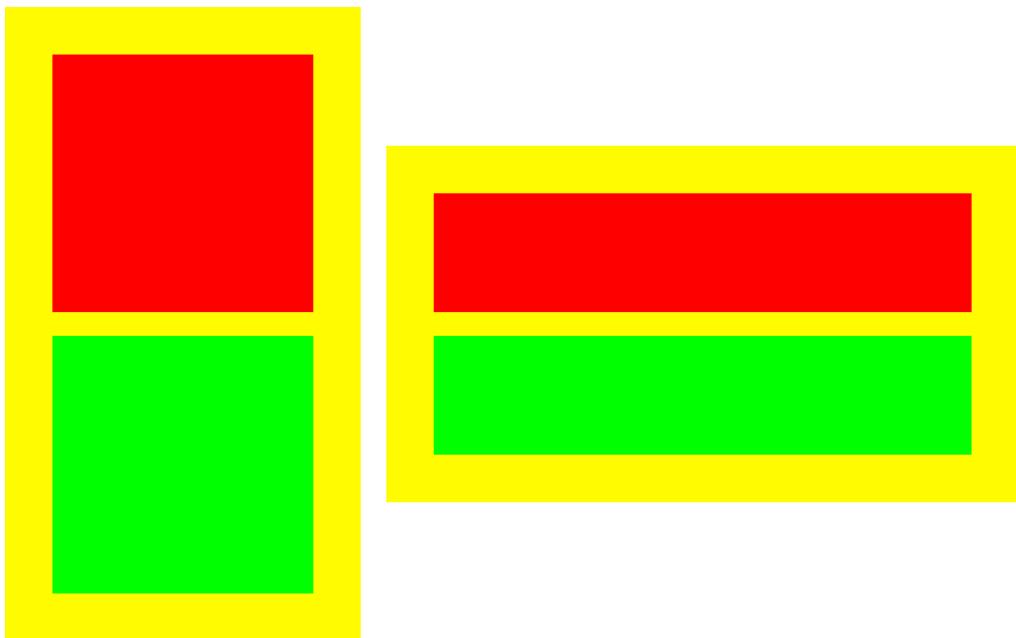
```
redView.leadingAnchor.constraint(equalTo:  
    view.bottomAnchor),
```

 Cannot convert value of type 'NSLayoutYAxisAnchor' to expected argument type
'NSLayoutAnchor<NSLayoutXAxisAnchor>' 

This is not foolproof. You can create invalid constraints with layout anchors. For example, between a `.leftAnchor` and `.leadingAnchor` which are both of type `NSLayoutXAxisAnchor`. Still, it's a big improvement over using VFL or `NSLayoutConstraint`.

A Layout Anchor Example

To complete our comparison of the three different ways to code a constraint let's recreate the last example with layout anchors (see sample code: [Anchors-v1](#)).



1. I'll skip the setup and assume we already have our view controller with the views created as in the last example:

```
class ViewController: UIViewController {

    private let padding: CGFloat = 50.0
    private let spacing: CGFloat = 25.0

    private let redView: UIView.makeView(color: .red)
    private let greenView: UIView.makeView(color: .green)

    override func viewDidLoad() {
        super.viewDidLoad()
        setupView()
    }

    private func setupView() {
        view.backgroundColor = .yellow
        view.addSubview(redView)
        view.addSubview(greenView)

        // Add constraints here
    }
}
```

2. Let's start with the leading anchors for the red and green views. These are both equal to the leading anchor of the superview plus the padding:

```
NSLayoutConstraint.activate([
    redView.leadingAnchor.constraint(equalTo:
        view.leadingAnchor, constant: padding),
    greenView.leadingAnchor.constraint(equalTo:
        view.leadingAnchor, constant: padding),
```

3. The trailing constraints are similar:

```
view.trailingAnchor.constraint(equalTo:
    redView.trailingAnchor, constant: padding),
view.trailingAnchor.constraint(equalTo:
    greenView.trailingAnchor, constant: padding),
```

4. The vertical spacing constraints:

```
redView.topAnchor.constraint(equalTo: view.topAnchor,
    constant: padding),
greenView.topAnchor.constraint(equalTo:
    redView.bottomAnchor, constant: spacing),
view.bottomAnchor.constraint(equalTo:
    greenView.bottomAnchor, constant: padding),
```

5. Finally using height anchors, we make the red view height match the green view height:

```
redView.heightAnchor.constraint(equalTo:
    greenView.heightAnchor)
])
```

Which Should You Use?

If you're creating constraints in code, and don't need to support iOS 8, my strong recommendation is to use layout anchors.

Using the `NSLayoutConstraint` initializer is the most verbose and least readable way to create constraints. There's no type safety, so it's easy to make mistakes that are difficult to spot in a long list of constraints.

The Visual Format Language is more concise and readable, but the need to set up view and metrics dictionaries is a pain. It doesn't handle some everyday situations, and Apple has not extended it to support new concepts like layout margins and the safe area.

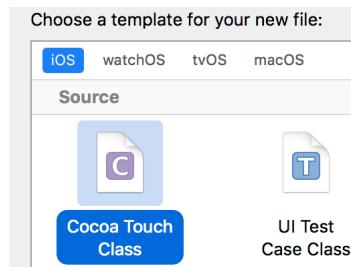
Layout anchors are type safe which catches most cut and paste style mistakes when you're writing constraints. This is both faster for the

developer and avoids a source of runtime crashes. I find them the most readable of the choices, and they work well with layout margins and the safe area which we look at in the next chapter.

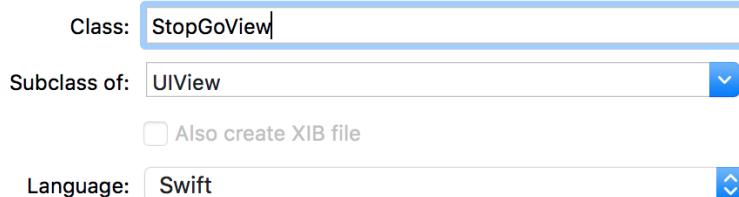
Constraints In A Custom View

How should you create your constraints when using a custom view? A good starting point is to create them from the initializer. Let's revisit the last example and move the setup and constraints for the red and green views to a custom subview (see sample code: [Anchors-v2](#)).

1. Starting with a new Xcode project with no storyboard add a new file (File > New > File...). Choose the iOS Cocoa Touch Class file template:



2. Name the class `StopGoView`, make sure it's a subclass of `UIView` and save it the project folder.



3. We can use the custom view template from chapter 2. This time we don't need to override `layoutSubviews`:

```
import UIKit

class StopGoView: UIView {

    override init(frame: CGRect) {
        super.init(frame: frame)
        setupView()
    }
}
```

```
required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
    setupView()
}

private func setupView() {
    // Build views
    // Add constraints
}
```

4. We can move most of the setup view code from the view controller to the custom view with some small changes. First to create the two views and add them to the view hierarchy:

```
private let padding: CGFloat = 50.0
private let spacing: CGFloat = 25.0

private let redView = UIView.makeView(color: .red)
private let greenView = UIView.makeView(color: .green)

private func setupView() {
    addSubview(redView)
    addSubview(greenView)
}
```

5. Then to constrain the two views to the edges of the superview using the padding and internal spacing:

```
private func setupView() {
    addSubview(redView)
    addSubview(greenView)

    NSLayoutConstraint.activate([
        redView.leadingAnchor.constraint(equalTo:
leadingAnchor, constant: padding),
        greenView.leadingAnchor.constraint(equalTo:
leadingAnchor, constant: padding),

        trailingAnchor.constraint(equalTo:
redView.trailingAnchor, constant: padding),
        trailingAnchor.constraint(equalTo:
greenView.trailingAnchor, constant: padding),
```

```
    redView.topAnchor.constraint(equalTo: topAnchor,
constant: padding),
    greenView.topAnchor.constraint(equalTo:
redView.bottomAnchor, constant: spacing),
    bottomAnchor.constraint(equalTo:
greenView.bottomAnchor, constant: padding),

    redView.heightAnchor.constraint(equalTo:
greenView.heightAnchor)
])
}
```

6. This custom view works fine when used in a storyboard or with programmatic layouts. To use it in our view controller we add a property for it and configure it as usual:

```
class ViewController: UIViewController {

private let stopGoView: StopGoView = {
    let view = StopGoView()
    view.translatesAutoresizingMaskIntoConstraints = false
    view.backgroundColor = .yellow
    return view
}()
```

7. Finally, add it as a subview and pin it to the edges of the superview with four constraints:

```
override func viewDidLoad() {
    super.viewDidLoad()
    setupView()
}

private func setupView() {
    view.backgroundColor = .yellow
    view.addSubview(stopGoView)

    NSLayoutConstraint.activate([
        stopGoView.leadingAnchor.constraint(equalTo:
view.leadingAnchor),
        stopGoView.topAnchor.constraint(equalTo:
view.topAnchor),
        stopGoView.trailingAnchor.constraint(equalTo:
view.trailingAnchor),
        stopGoView.bottomAnchor.constraint(equalTo:
view.bottomAnchor)
```

```
    ])
}
```

Key Points To Remember

We have covered a lot in this chapter. Here are the key points to remember when creating your constraints programmatically:

- Add your views to the view hierarchy before activating their constraints.
- Activate and deactivate your constraints. Don't use the old add and remove constraint methods.
- Use the `NSLayoutConstraint` class methods to activate and deactivate your constraints in batches. It's faster and more readable than setting `isActive` on each constraint:

```
NSLayoutConstraint.activate([
    // constraints
])
```

- Using `removeFromSuperview` to remove a view from the view hierarchy also removes any constraints involving that view or any of its subviews.
- Hiding a view doesn't deactivate its constraints.
- Don't forget to disable the translation of the autoresizing mask into constraints when you create a view in code:

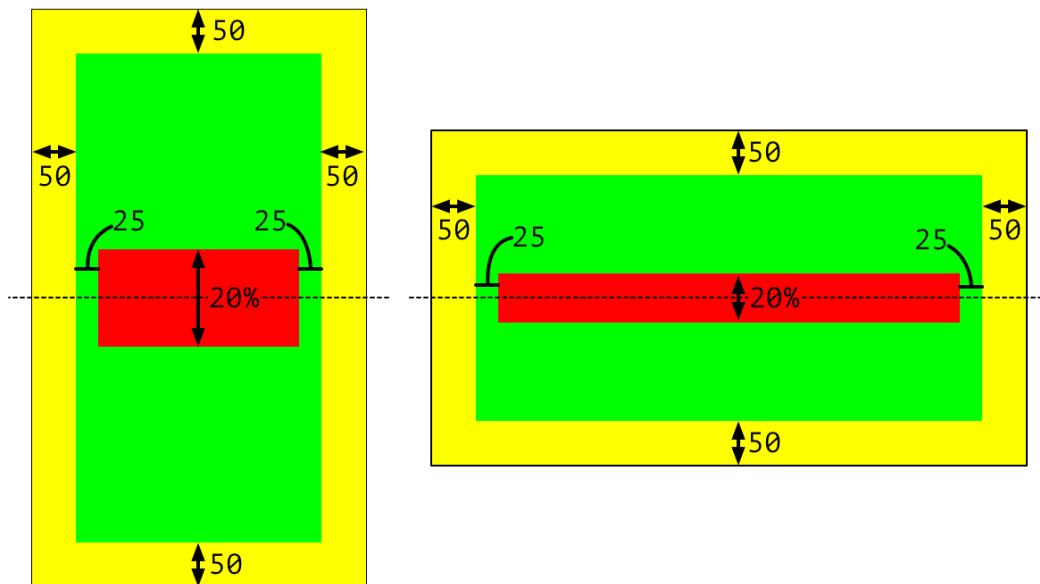
```
translatesAutoresizingMaskIntoConstraints = false
```

- Unless you still need to support iOS 8 use layout anchors.
- We created our constraints in `viewDidLoad` in the view controller or in the initializer for custom views. These are good starting choices for the layouts we have seen so far.

Test Your Knowledge

Challenge 5.1 Nested View Layout

In challenge 4.1 we built a nested layout using Interface Builder. Let's see if we can re-create it building our layout and constraints in code. Here's a reminder of the layout:



The green view has an external spacing of 50 points and an internal spacing of 25 points to the red view. The red view is vertically centered in the green view and is 20% the height of the green view.

1. Build this layout without using Interface Builder. Create the views, build the view hierarchy and add constraints in code in the view controller.
2. Since I didn't explain how to do it yet, create your constraints with the edges of the superview rather than to the safe area. We can improve this layout with the safe area and margins in the next chapter.

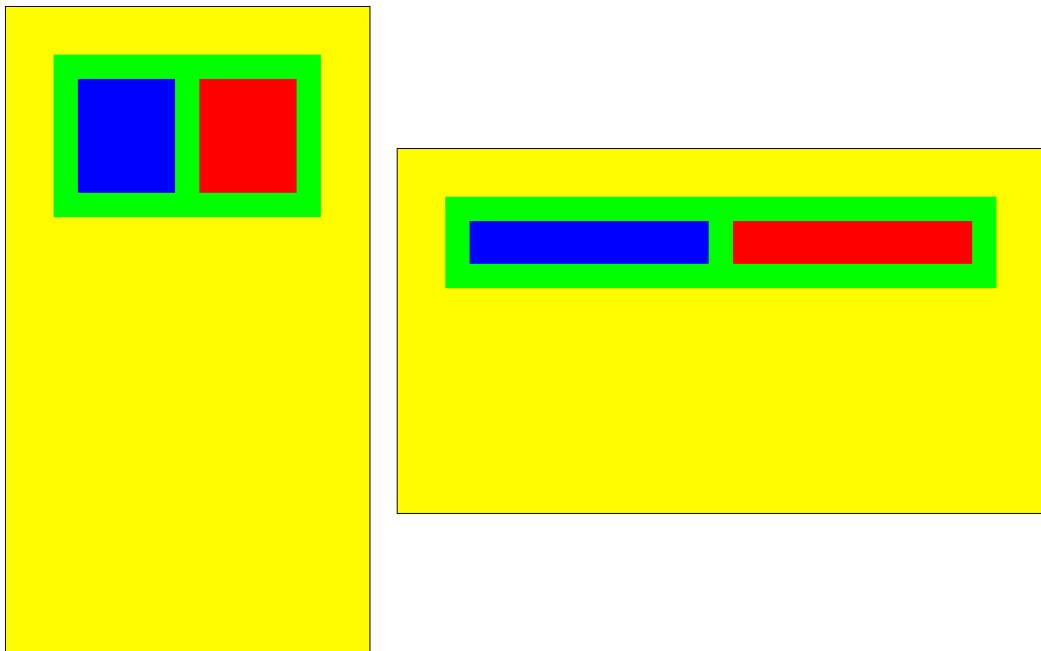
Hints And Tips

1. I recommend you use layout anchors to create your constraints.
2. Remember to add subviews to their parent views before activating constraints.
3. Use `NSLayoutConstraint.activate` to activate your constraints in a single batch.

4. As always don't forget to disable the automatic translation of the autore sizing mask into constraints.

Challenge 5.2 The Tile View

Let's revisit the blue and red tile custom view we built with a manual frame based layout back in chapter 2:



The green container view for the tiles has 50 points of spacing to leading, top and trailing edges of the screen and a height that's 25% of the yellow superview height.

The blue and red tiles have 25 points of padding on all sides and have equal width and height.

1. Build this layout without using Interface Builder.
2. Minimize the layout code in the view controller by creating a custom subclass of `UIView` for the green view and its contents.
3. As in the last challenge create your constraints with the edges of views until we learn how to use the safe area and margins.

Hints And Tips

1. If you need a reminder of how to create a custom view refer back to [Constraints In A Custom View](#).

2. You don't need to implement `layoutSubviews` in the custom view.
3. If you're still struggling to get started here's a template for a custom view you can use for the tile view:

```
// TileView.swift
import UIKit

class TileView: UIView {

    override init(frame: CGRect) {
        super.init(frame: frame)
        setupView()
    }

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
        setupView()
    }

    private func setupView() {
    }
}
```

4. Create the blue and red views, add them to the tile view and then create their constraints.
5. Think about how many constraints you need in your tile view. Don't add more than you need.
6. The view controller should only need constraints to pin the leading, top and trailing edges and set the height of the tile view.
7. Use a property in the view controller to store the tile view.
8. Don't forget to disable the translation of the autoresizing mask and set the background color of the tile view.
9. Your tile view should only need eight constraints (four horizontal and four vertical). It's easy to get misled by the fact that the blue and red views have both equal width and equal height. You don't need to add an equal height constraint as the top, and bottom constraints already fix the height of the views. You do need an equal width constraint (make sure you can see why).

Chapter 6

Safe Areas And Layout Margins

You often need to mark out areas of a layout. Maybe you want to define areas to show content, create margins inside a view, group views or mark out space between views.

In the early days of Auto Layout, this led to using extra spacing or dummy views and lots of padding constants in your spacing constraints. The dummy views were there only to mark out an area between or to contain other views but still created an overhead as part of the view hierarchy.

Apple added `UILayoutGuide` in iOS 9 as a lightweight way to define a rectangular area. It has a layout frame (`CGRect`), layout anchors that you can create constraints with and an owning view but it's not part of the view hierarchy.

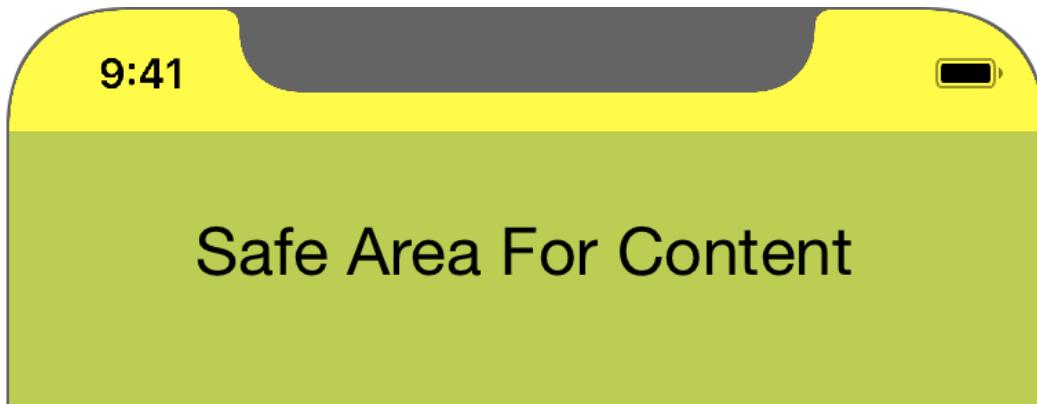
Apple uses layout guides to define default safe areas and margins, but we can also create our own. After reading this chapter, you should be able to:

- Use the safe area layout guide introduced in iOS 11 to create layouts that protect your content from the status, navigation, and toolbars, and the home screen indicator and rounded device corners found on newer devices like the iPhone X.
- Fall back to the top and bottom layout guides when you need to deploy back to iOS 10 and iOS 9.
- Use margins to create extra spacing around your views without littering your constraints with constants.
- Create layout guides to replace spacer views for layouts needing

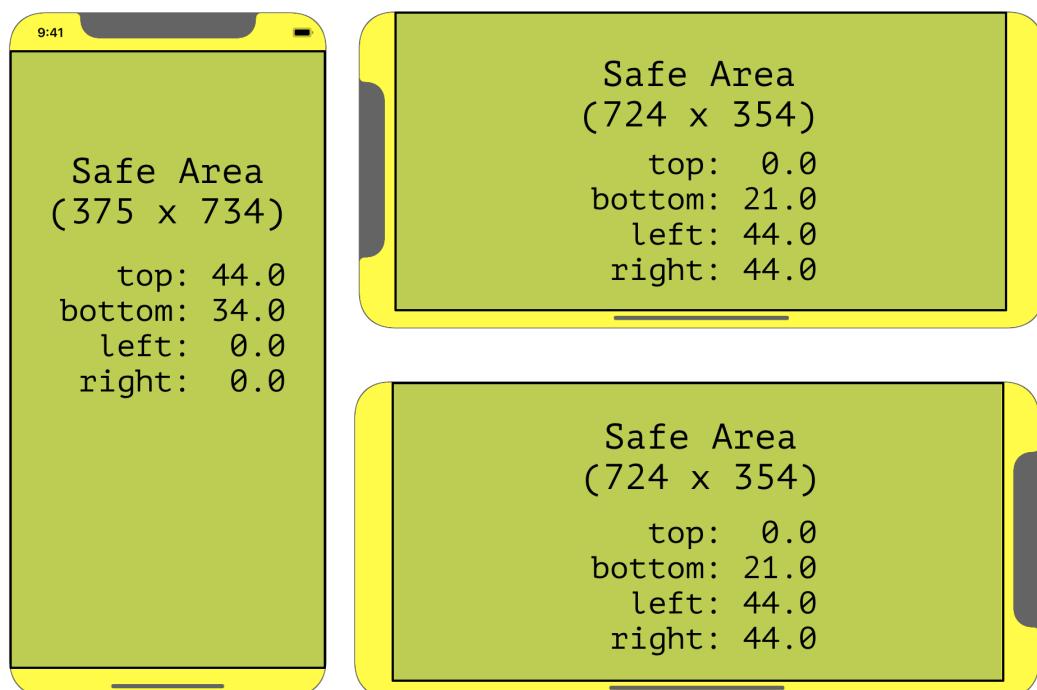
constraints on the spacing between views.

Safe Area Layout Guide

Apple added safe area layout guides in iOS 11 to define a rectangle safe for you to show content. The status, navigation, and tab bars never cover the safe area.

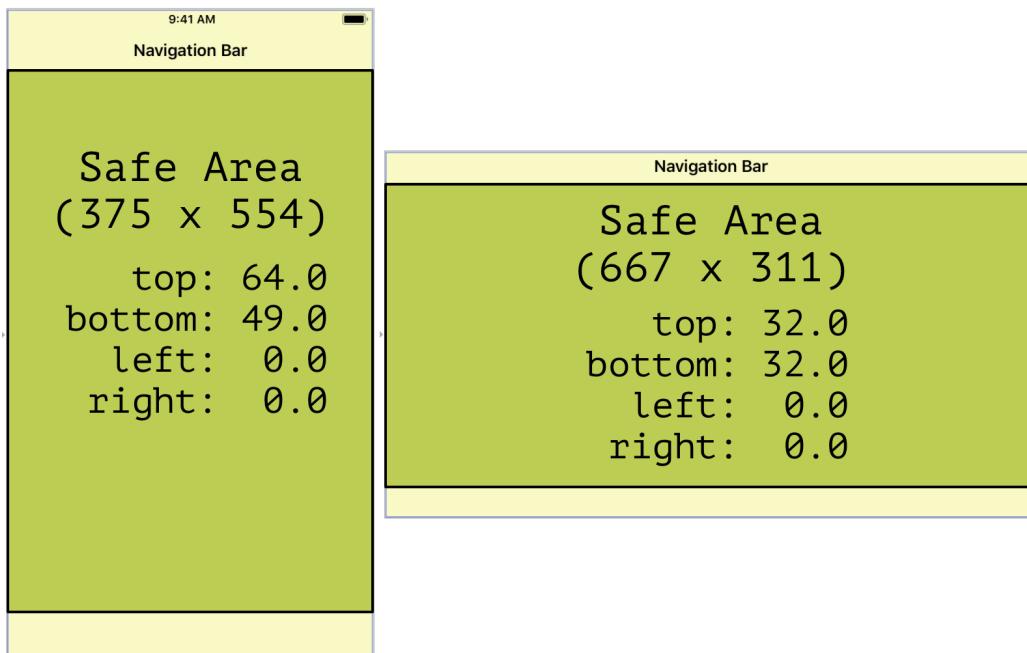


Safe areas become even more important with device designs like the iPhone X. The rounded corners, top central housing, and bottom home indicator can clip, hide or overlay views that would be fine on other iPhone models. Here's how the safe area is inset from the superview on an iPhone X (and XS) in portrait and landscape with just a status bar:



In portrait, the top inset of 44 points keeps your content away from the center housing and the rounded corners. The bottom inset of 34 points allows for the home control indicator. Note in landscape that the left and right insets are both 44 points regardless of the side the “notch” is on.

The safe area insets change to allow for navigation bars and tab bars. For comparison here's an iPhone 8 with a top navigation bar and bottom tab bar. There's no notch to worry about, so the left and right insets are always zero.



The safe area layout guide is a property of the view. It's a `UILayoutGuide` with a `layoutFrame` and a set of layout anchors that mark out the safe area of the view. In general, you want to keep your content in the safe area.

When using Auto Layout, you create your constraints to the layout anchors of the guide. For example, to pin `redView` to the leading and trailing edges of the safe area of a view:

```
let guide = view.safeAreaLayoutGuide
redView.leadingAnchor.constraint(equalTo: guide.leadingAnchor)
redView.trailingAnchor.constraint(equalTo:
    guide.trailingAnchor)
```

The `safeAreaInsets` property of `UIView` gives you the amount the safe area is inset from the edge of a view. If a view is entirely inside the safe area, it has zero insets.

```
// UIEdgeInsets  
let safeInsets = view.safeAreaInsets
```

A view must be loaded and onscreen for its safe area to be set (so don't rely on the insets in `viewDidLoad`). You cannot change the safe area layout guide or the safe area insets. To increase the size of the safe area for a view controller, use the `additionalSafeAreaInsets` property. For example, to increase the top inset to allow for a custom toolbar:

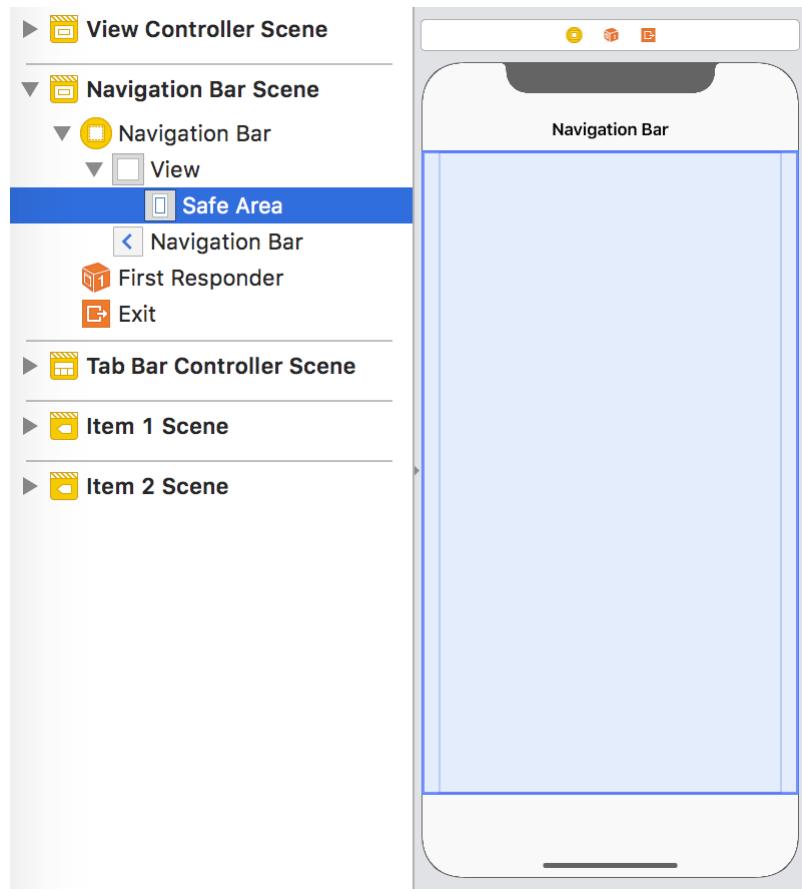
```
additionalSafeAreaInsets = UIEdgeInsets(top: toolbarHeight,  
left: 0, bottom: 0, right: 0)
```

If your view controller needs to know when the safe area changes use the view controller method `viewSafeAreaInsetsDidChange()` or in a custom view use `safeAreaInsetsDidChange()`.

Using The Safe Area With Interface Builder

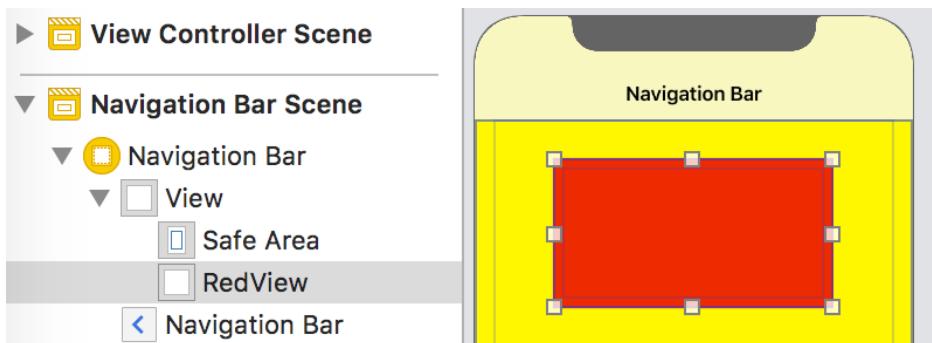
Let's look at an example of using Interface Builder to create constraints with the safe area layout guide of a view (see sample code: [SafeArea-v1](#)):

1. My project has a view controller embedded in both a navigation and a tab bar controller. Notice how the safe area layout guide shows up in the document outline below the root view:



Selecting the safe area layout guide in the outline highlights the safe area for that view in the canvas taking into account both the device and any bars added by parent views.

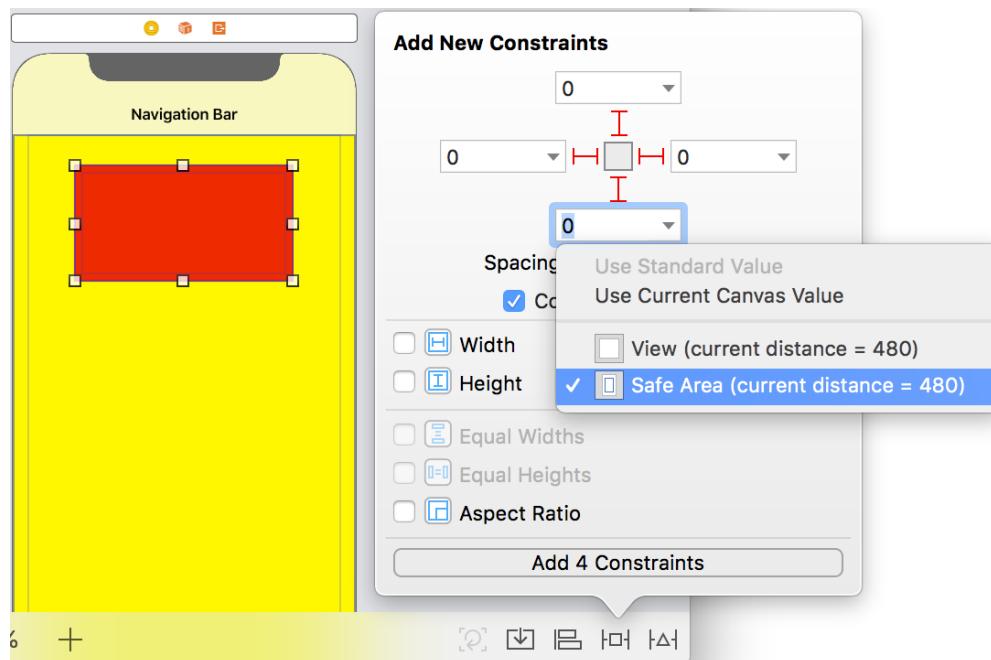
2. All views have a safe area layout guide property but by default Interface Builder only shows it for the view controller's root view. See what happens when I add a red subview to the root view:



Notice that Interface Builder is not showing the safe area layout guide for the red view. Most of the time this makes sense. The safe

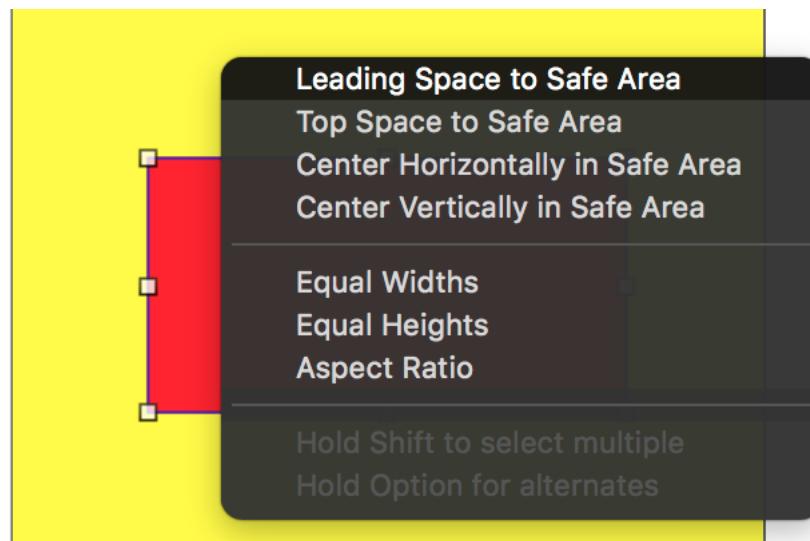
area layout guide for a view that's not covered or clipped is the same size as the view with zero insets on all sides. We'll see how to make Interface Builder show the guide for subviews in the next example.

3. There are several ways to create constraints between a view and the safe area layout guide. For example, with the red subview selected use the Add New Constraint tool to create horizontal and vertical spacing constraints. By default, if Interface Builder is showing the safe area layout guide for a view, it uses it in the constraint.

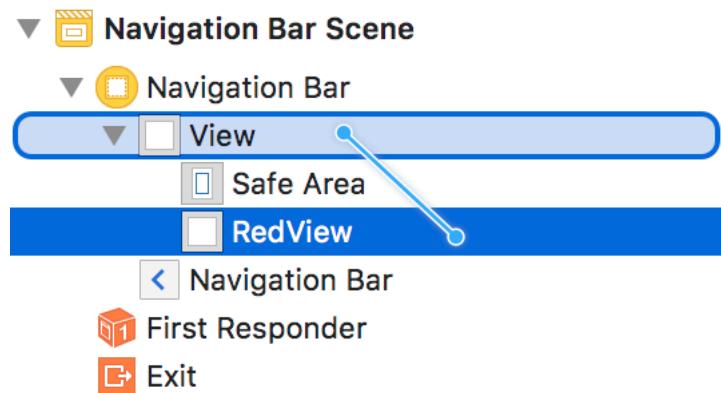


You can switch between the safe area and the superview (to create constraints to the edges or the margin of the root view) with the drop-down menu in the spacing value.

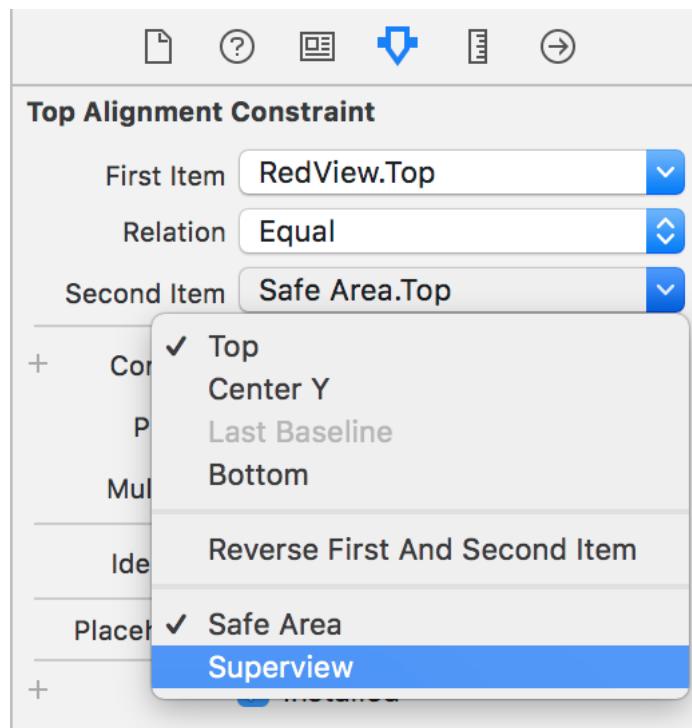
4. Alternatively, control-drag in the canvas or document outline between a subview and its superview. The constraints menu defaults to using the safe area layout guide if Interface Builder is showing the guide for that view (use the Option key to switch between the safe area and margin):



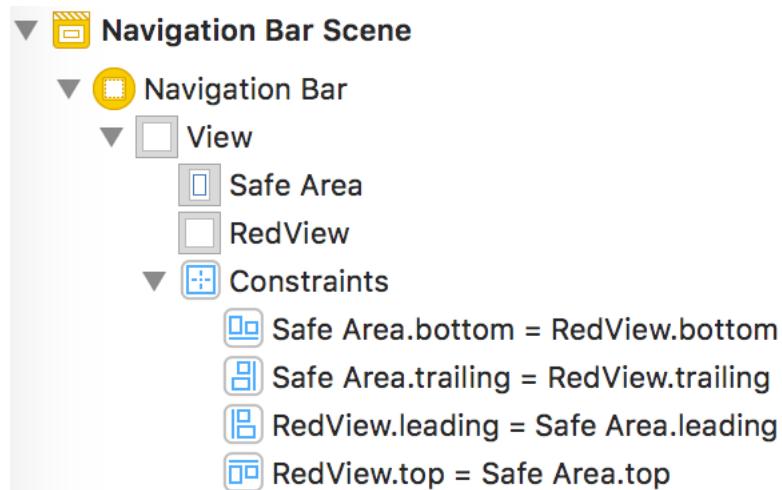
If you're using the document outline drag to or from the view not the safe area layout guide icon under the view:



5. Finally, you can edit a safe area layout guide constraint with the attributes inspector and switch the constraint item between the safe area and the superview:



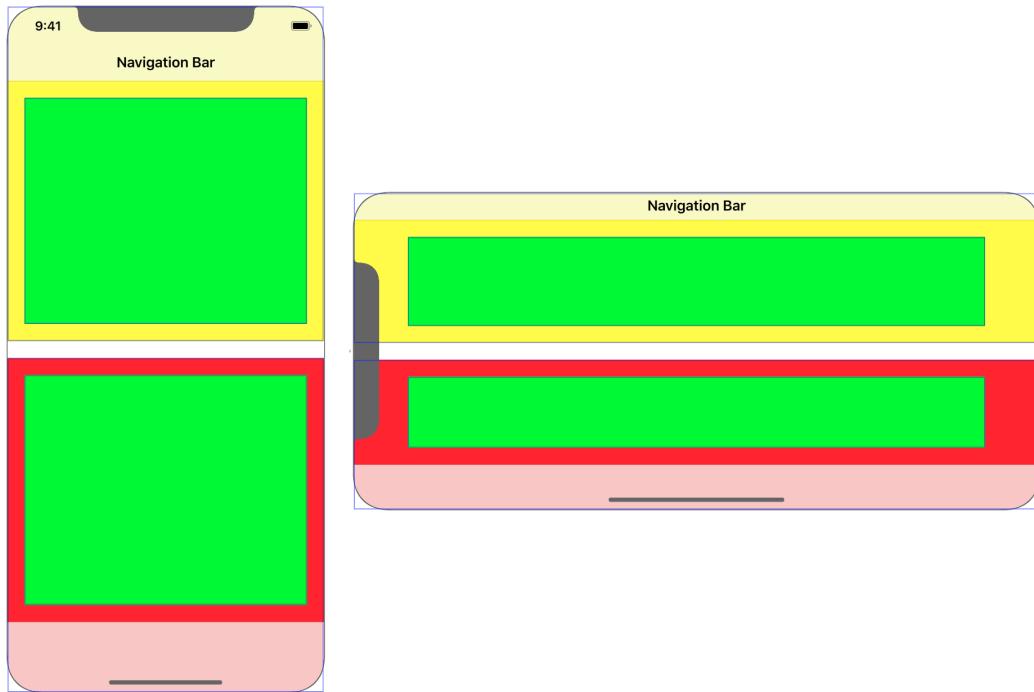
6. Check your constraints in the document outline to see if they are using the safe area:



Sometimes you want to use the safe area layout guide for more than just the root view. Let's see an example of how and why you might do that (see sample code: [safeArea-v2](#)).

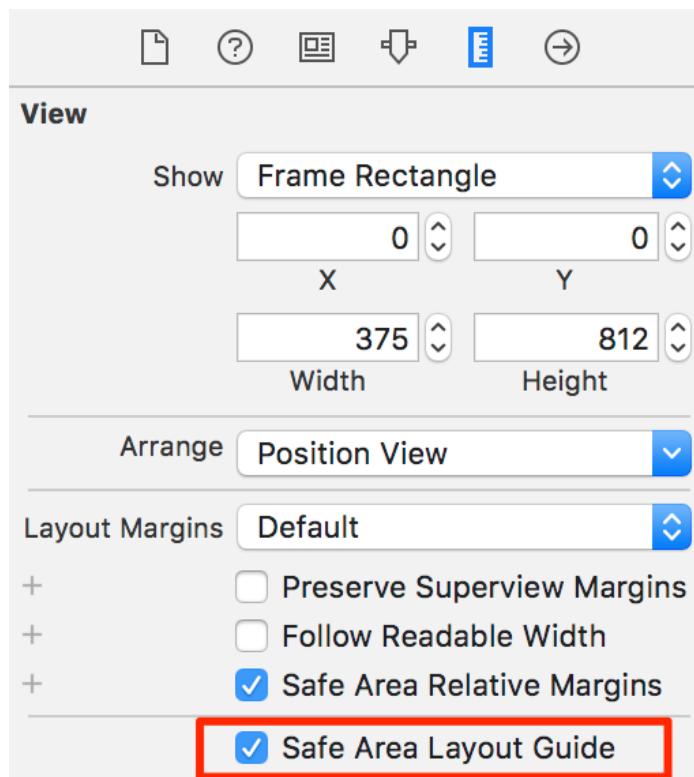
This layout has a yellow and red view that are both pinned to the edges of the view controller's root view with some vertical spacing in the middle. I embedded the view controller in a navigation controller which I then embedded in a tab bar controller. The navigation bar covers the top of

the yellow view, and the tab bar covers the bottom of the red view:

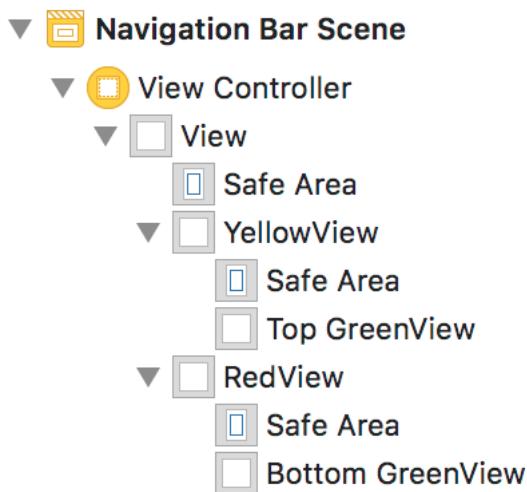


The two green subviews represent my content. I don't want either to be covered by the navigation or tab bars. To make sure they stay within the safe area I need to create constraints with the safe area layout guides of the yellow and red views:

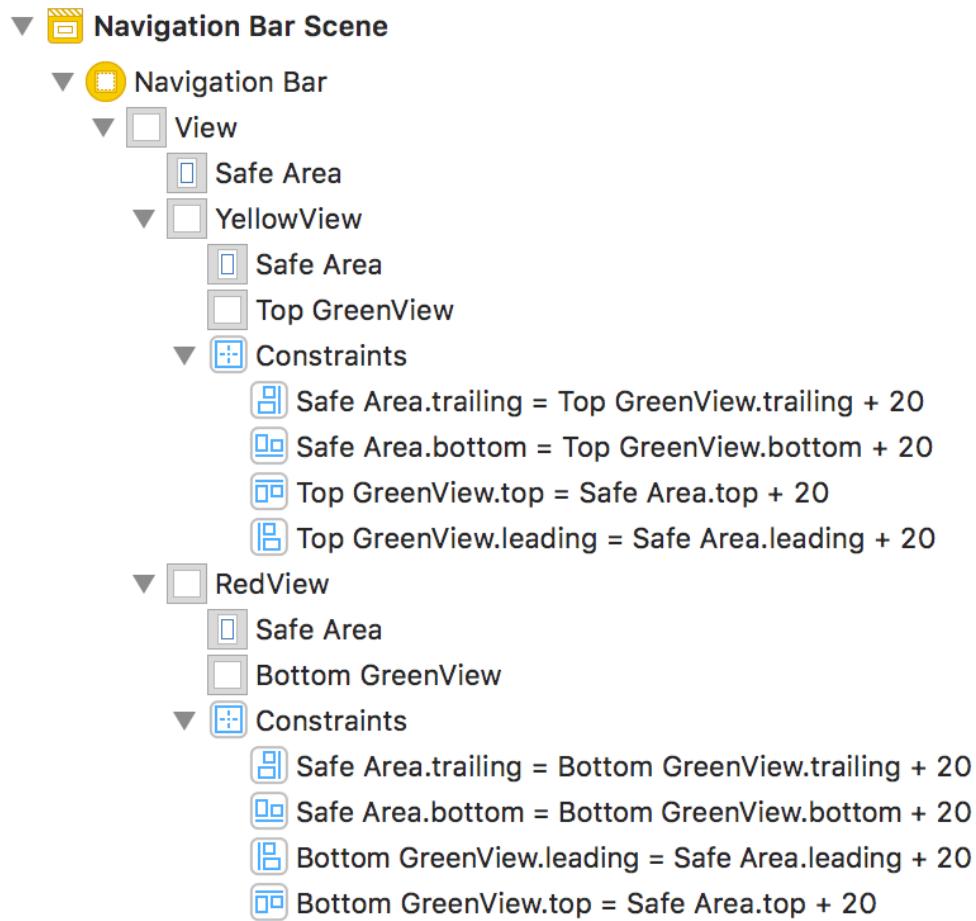
1. We first need to make Interface Builder show the safe area layout guides for the yellow and red container views. Select each view and use the Size Inspector to enable the safe area layout guide:



2. Interface Builder should now be showing the safe area layout guides for both views in the document outline:

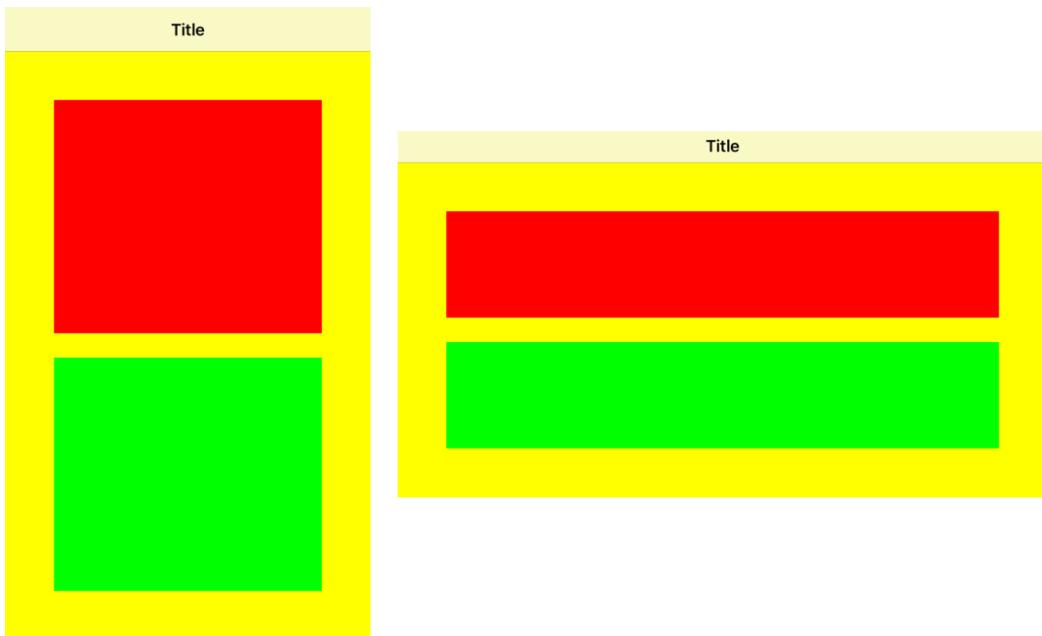


3. Now when you add constraints for the two green content views you will have the option to use the safe area layout guides of the yellow and red container views. Your constraints should end up looking like this:



Using The Safe Area In Code

To see an example of using the safe area layout guides in code let's fix our layout anchors project from the last chapter to use the safe area. The red and green views have 50 points of external padding that should now be to the safe area, not the screen edge. I also embedded the view controller in a navigation controller (see sample code: [Anchors-v3](#)):



1. To recap our view controller follows the usual pattern of creating the views and adding them to the view hierarchy in `setupView` called from `viewDidLoad`:

```
class ViewController: UIViewController {

    private let externalPadding: CGFloat = 50.0
    private let internalSpacing: CGFloat = 25.0

    private let redView = UIView.makeView(color: .red)
    private let greenView = UIView.makeView(color: .green)

    override func viewDidLoad() {
        super.viewDidLoad()
        setupView()
    }

    private func setupView() {
        view.backgroundColor = .yellow
        view.addSubview(redView)
        view.addSubview(greenView)

        // Add constraints here
    }
}
```

2. Then in the `setupView` method of the view controller, we create our constraints. The safe area layout guide is a property of the view:

```
let guide = view.safeAreaLayoutGuide
```

3. Instead of constraining our view to the layout anchors of the root view we use the layout anchors of the safe area layout guide. For example, this is the leading constraint:

```
redView.leadingAnchor.constraint(equalTo:  
    guide.leadingAnchor, constant: externalPadding),
```

4. The full set of constraints for the layout after converting to use the safe area layout guide:

```
let guide = view.safeAreaLayoutGuide  
NSLayoutConstraint.activate([  
    redView.leadingAnchor.constraint(equalTo:  
        guide.leadingAnchor, constant: externalPadding),  
    greenView.leadingAnchor.constraint(equalTo:  
        guide.leadingAnchor, constant: externalPadding),  
  
    guide.trailingAnchor.constraint(equalTo:  
        redView.trailingAnchor, constant: externalPadding),  
    guide.trailingAnchor.constraint(equalTo:  
        greenView.trailingAnchor, constant: externalPadding),  
  
    redView.topAnchor.constraint(equalTo: guide.topAnchor,  
        constant: externalPadding),  
    greenView.topAnchor.constraint(equalTo:  
        redView.bottomAnchor, constant: internalSpacing),  
  
    guide.bottomAnchor.constraint(equalTo:  
        greenView.bottomAnchor, constant: externalPadding),  
  
    redView.heightAnchor.constraint(equalTo:  
        greenView.heightAnchor)  
])
```

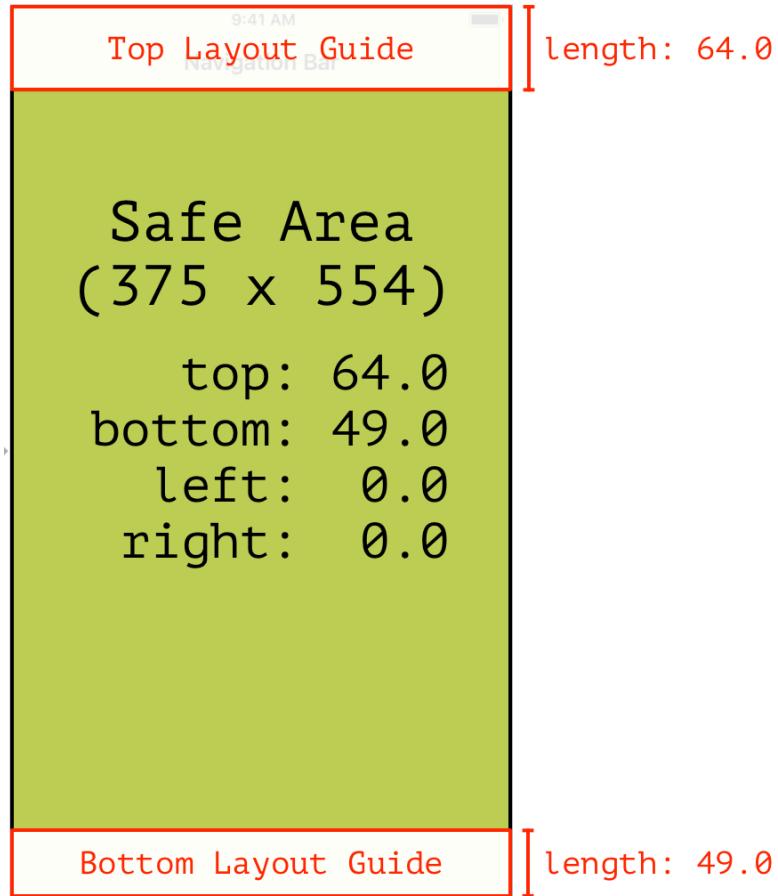
5. We'll see how to remove the padding constants from each constraint when we look at layout margins.

Top And Bottom Layout Guides

Apple introduced `topLayoutGuide` and `bottomLayoutGuide` as properties of `UIViewController` in iOS 7. Like the safe area, they have layout anchors you can use in your constraints to keep your content away from the status bar and other container views like the navigation and tab bar.

Unlike the safe area layout guide, they are not of type `UILayoutGuide`. Instead, they implement the `UILayoutSupport` protocol which only promises to have a `topAnchor`, `bottomAnchor`, `heightAnchor` and a `length`.

Here's how they compare to the safe area for an iPhone 8 with a top navigation bar and bottom tab bar:



To use these guides you typically constrain your content view to the bottom anchor of the top layout guide and the top anchor of the bottom layout guide:

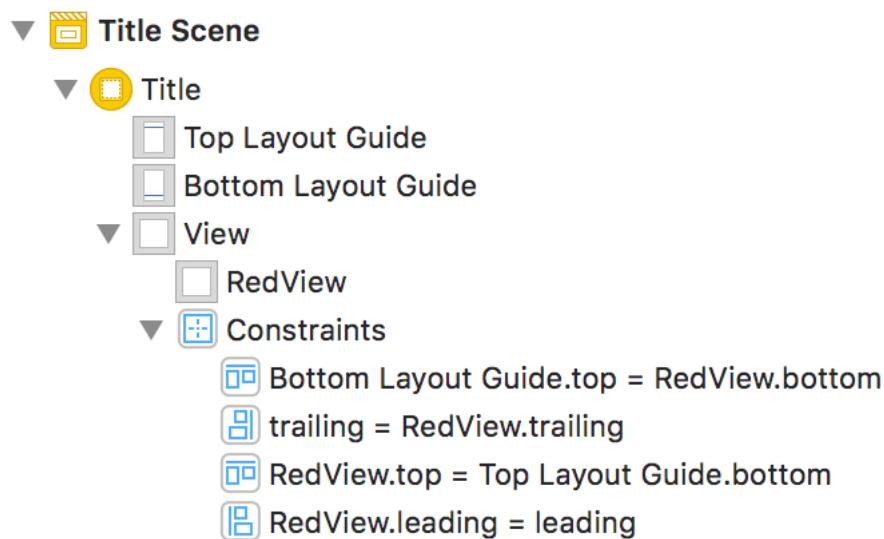
```
redView.topAnchor.constraint(equalTo:  
    topLayoutGuide.bottomAnchor)  
redView.bottomAnchor.constraint(equalTo:  
    bottomLayoutGuide.topAnchor)
```

When Apple introduced safe area layout guides in iOS 11, they deprecated the top and bottom layout guides. You should switch to using the safe area to support new devices like the iPhone X and only fall back to the top and bottom layout guides when you need to support older versions of

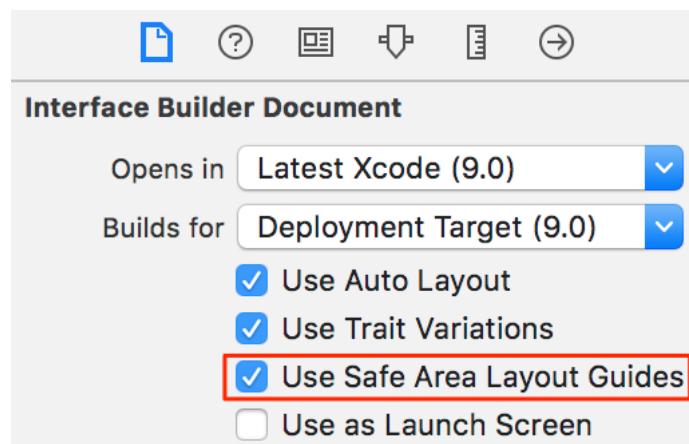
iOS.

Migrating Storyboards

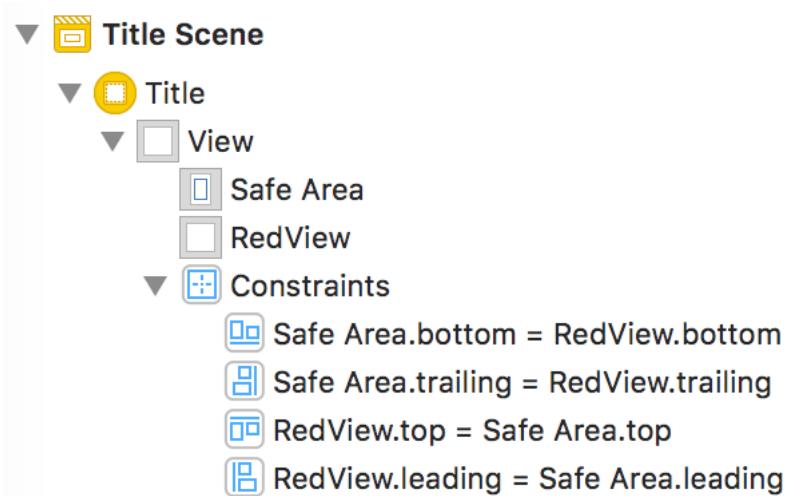
The good news if you use safe area layout guides in Interface Builder is that they are backward compatible to iOS 9. If you have an older project that's using top and bottom layout guides you can switch to safe area layout guides and your app still works back to iOS 9. For example, here's an older project where I pinned a red view to the top and bottom layout guides and leading and trailing edges:



You can switch this layout to using safe area layout guides in the Interface Builder Document settings using the File Inspector (don't confuse this with the view setting in the Size Inspector):



Interface Builder automatically replaces the top and bottom layout guides with a safe area layout guide and updates the constraints:



Check your constraints after converting to safe area layout guides. Interface Builder converts any leading and trailing constraints to using the safe area which might not be what you want.

Checking For Safe Area Availability

The bad news if you're building your layout in code is that you need to handle this fallback yourself by checking for availability. Let's change the deployment target of our last project to iOS 9:



We are using the safe area layout guide which is not available before iOS 11. The following line of code will no longer compile:

```
let guide = view.safeAreaLayoutGuide
```

To fix this, we can test for the availability of iOS 11 before using the safe area layout guide. Otherwise, we fall back to creating constraints with the top and bottom layout guides. Our constraints look something like this (see sample code: [Anchors-v4](#)):

```
if #available(iOS 11, *) {
    let guide = view.safeAreaLayoutGuide
    // Create constraints using safe area
} else {
    // fallback to top and bottom layout guides
    // and leading and trailing edges.
}
```

Having to write your constraints twice like this is painful. I prefer to move the fallback code to an extension on `UIViewController`:

```
// UIViewController+SafeAnchor.swift
import UIKit
extension UIViewController {
    var safeTopAnchor: NSLayoutYAxisAnchor {
        if #available(iOS 11, *) {
            return view.safeAreaLayoutGuide.topAnchor
        } else {
            return topLayoutGuide.bottomAnchor
        }
    }
    var safeBottomAnchor: NSLayoutYAxisAnchor {
        if #available(iOS 11, *) {
            return view.safeAreaLayoutGuide.bottomAnchor
        } else {
            return bottomLayoutGuide.topAnchor
        }
    }
    var safeLeadingAnchor: NSLayoutXAxisAnchor {
        if #available(iOS 11, *) {
            return view.safeAreaLayoutGuide.leadingAnchor
        } else {
            return view.leadingAnchor
        }
    }
    var safeTrailingAnchor: NSLayoutXAxisAnchor {
        if #available(iOS 11, *) {
            return view.safeAreaLayoutGuide.trailingAnchor
        } else {
            return view.trailingAnchor
        }
    }
}
```

The computed properties are anchors that use the safe area layout guide for iOS 11 or fallback to the top and bottom layout guides and leading and trailing anchors of the root view. Using this to write our backward

compatible constraints:

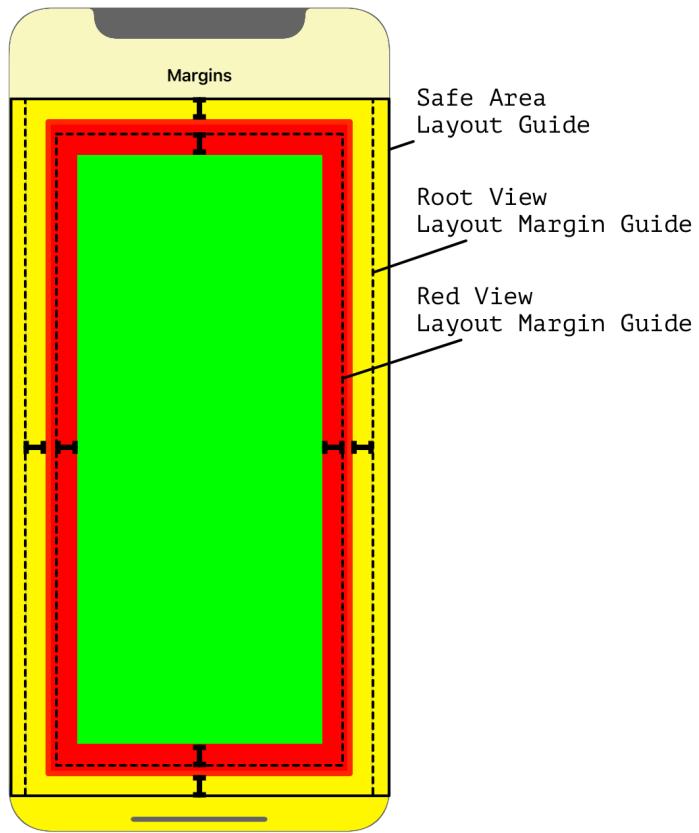
```
NSLayoutConstraint.activate([
    redView.topAnchor.constraint(equalTo: safeTopAnchor,
        constant: externalPadding),
    safeBottomAnchor.constraint(equalTo: greenView.bottomAnchor,
        constant: externalPadding),
    ...
])
```

Layout Margins

All views have a margin layout guide which you can use when you want extra spacing or padding insets from the edge of the view to any content subviews. The `layoutMarginsGuide` property of a view is a layout guide with the usual set of layout anchors for creating constraints with the margin of the view.

Creating Margin Constraints In Interface Builder

To see how margins work let's build a nested view that uses the default margins to create some padding between views. The green view is a subview of the red view which is a subview of the yellow root view:

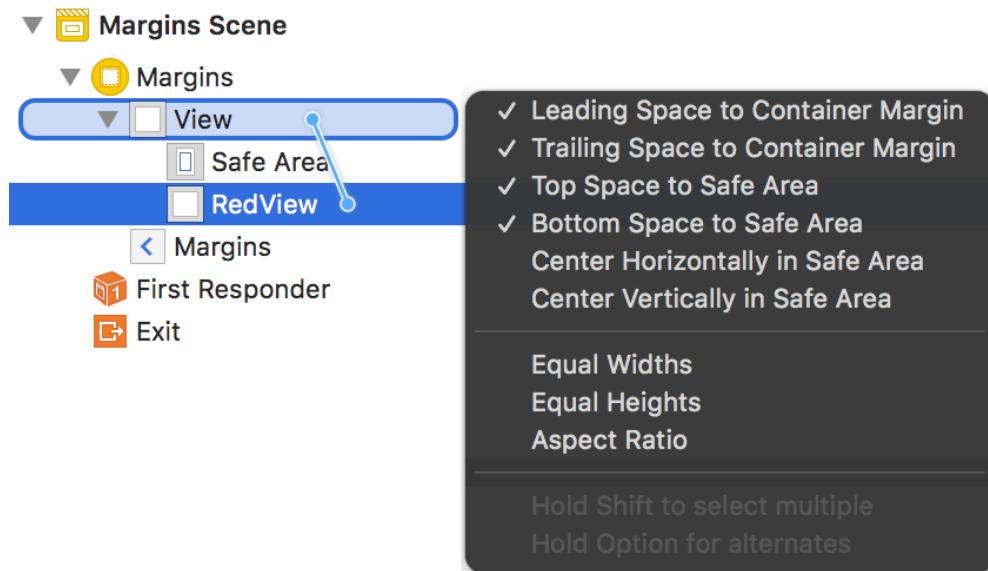


The red view is constrained to the leading and trailing margins of the root view and the top and bottom of the safe area. The green view is constrained to the margins of the red view (see sample code: [Margins-v1](#)).

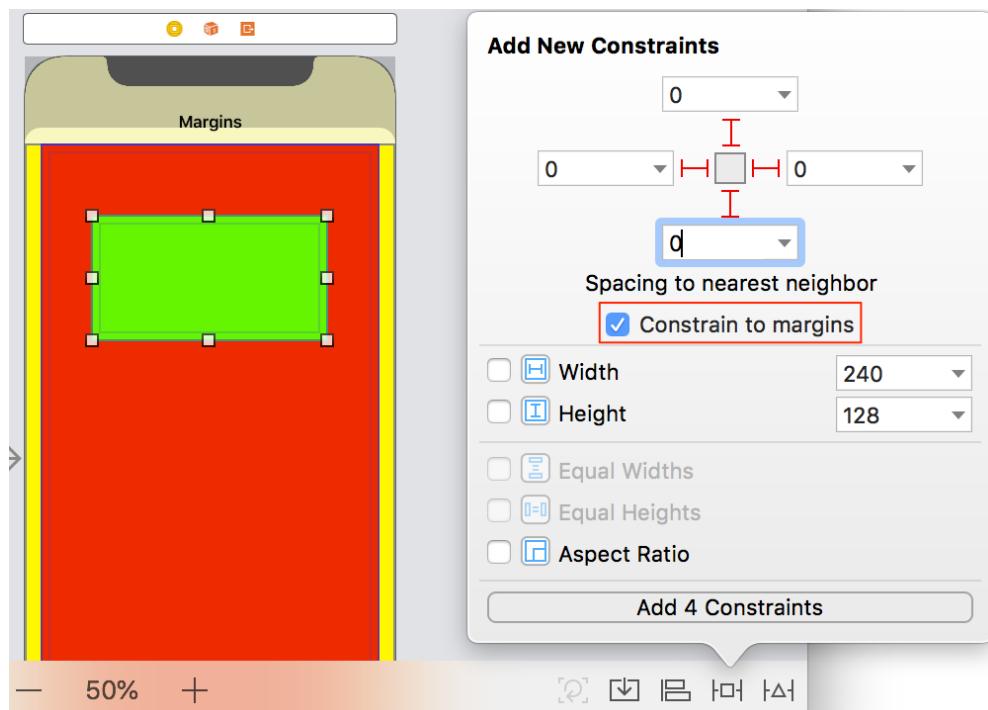
1. Start with the Single View App Xcode template and embed a view controller in a navigation controller. Then drag a plain view onto the Interface Builder canvas and change its background color to red.
2. Use the blue dotted guidelines in the canvas to size and position the red view to fit between the side margins and the top and bottom of the safe area:



3. Control-drag between the red view and the root view in the document outline and while holding the Shift key first select the top and bottom space constraints to the safe area. Then hold the Option key and select the leading and trailing space constraints to the container margin:

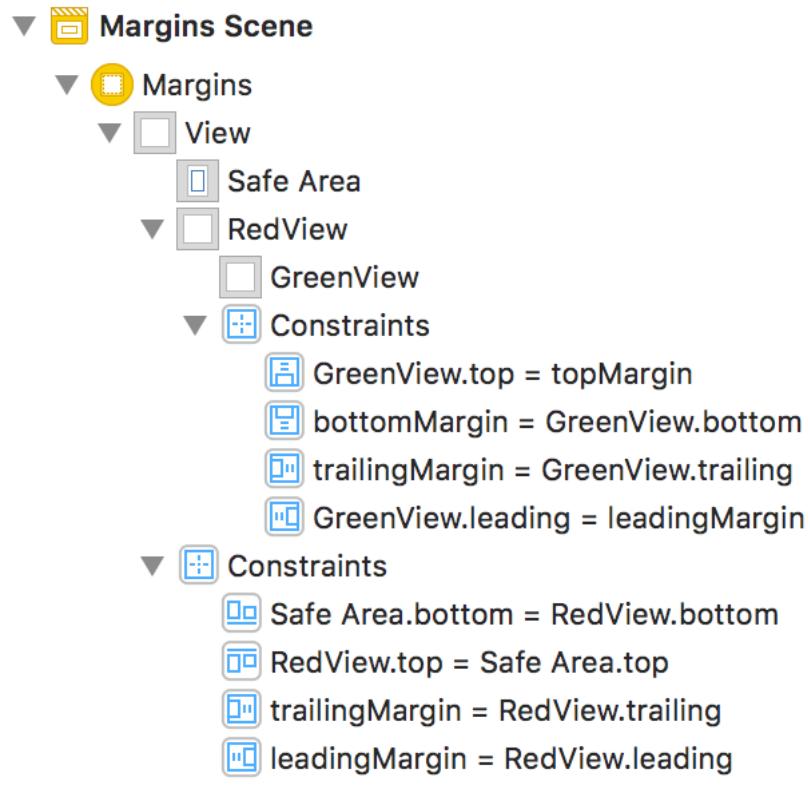


- Add a green subview to the red view. This time use the Add New Constraints tool to pin the green view to the margins of the red view with zero spacing. Make sure you have “Constrain to margins” selected:

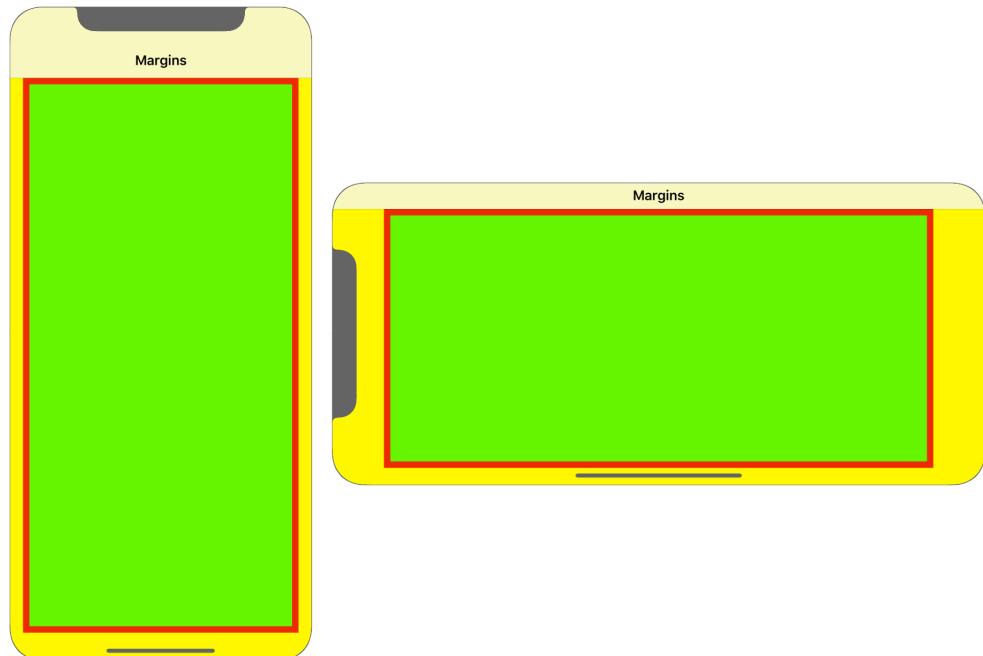


- It's easy to make a mistake so check your constraints in the document outline and fix any constraints with extra spacing constants or that

use the wrong item:



6. Here's how it looks on an iPhone X in portrait and landscape:



Safe Area Relative Margins (iOS 11)

Starting with iOS 11 a view's margins are, by default, automatically increased if necessary to keep them inside the safe area. You can see this effect if we constrain a view to the margins of a view controller's root view. In this case with a top navigation bar:



It may not be obvious, but the top of the red view is below the navigation bar even though I constrained it to the top margin of the yellow root view.

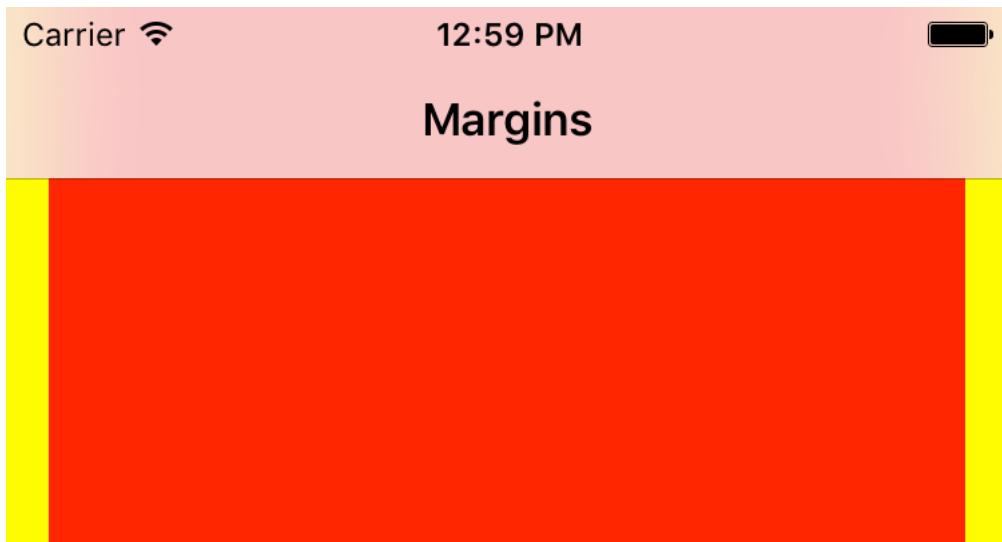
```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    print(view.layoutMargins)
}

// iPhone X
UIEdgeInsets(top: 88.0, left: 16.0, bottom: 34.0, right: 16.0)
```



I'm printing the margins in `viewDidAppear`. The margins are only valid once the view controller has loaded the root view and it's onscreen.

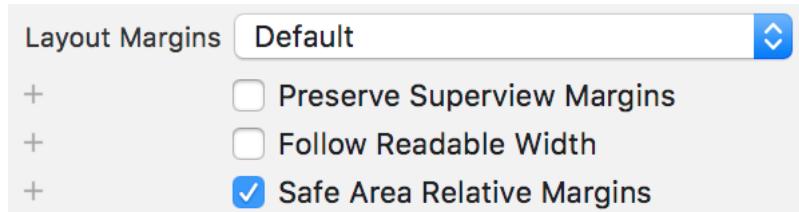
The 88 point top inset and 34 point bottom inset match the safe area layout guide insets. Unfortunately, it's not backward compatible. Before iOS 11 the margins didn't take into account the top and bottom layout guides. Here's how the same layout would look on an iPhone 7 running iOS 10:



The top of the red view has disappeared behind the status and navigation bars. The root view top margin now has a zero inset:

```
// iPhone 7 (iOS 10)
UIEdgeInsets(top: 0.0, left: 16.0, bottom: 0.0, right: 16.0)
```

You can disable safe area relative margins for a view with the Size Inspector in Interface Builder. The default setting is enabled:



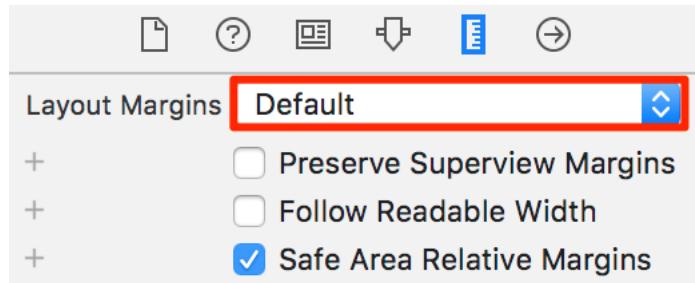
In code use the `insetsLayoutMarginsFromSafeArea` property of the view:

```
if #available(iOS 11, *) {
    // default is true
    view.insetsLayoutMarginsFromSafeArea = false
}
```

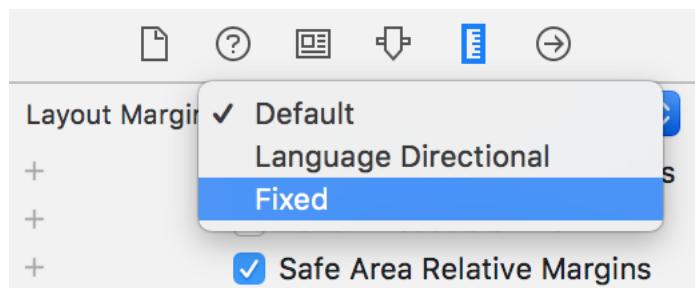
If your minimum deployment target is iOS 11 use the layout margins to inset your content inside the safe area. If deploying back to iOS 10 or earlier use the top and bottom layout guides and add any padding as a constant to your constraints.

Changing The Size Of Margins

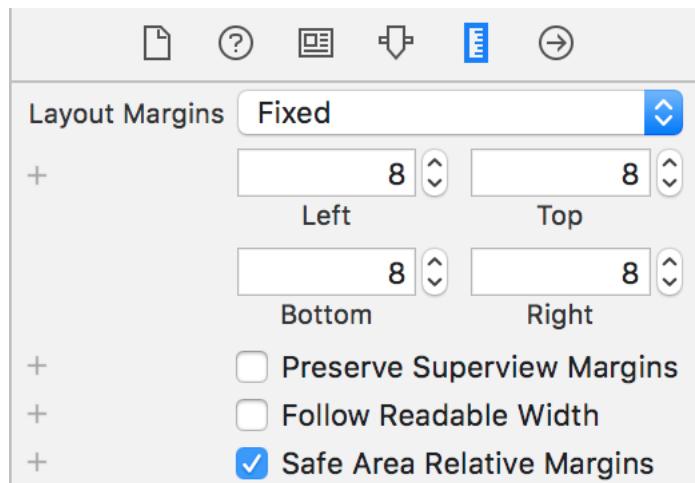
A new UIView has default layout margins that are inset by 8 points on all sides. (A view controller's root view is different as we'll see shortly). You can change the default margin for a view using the Size inspector in Interface Builder:



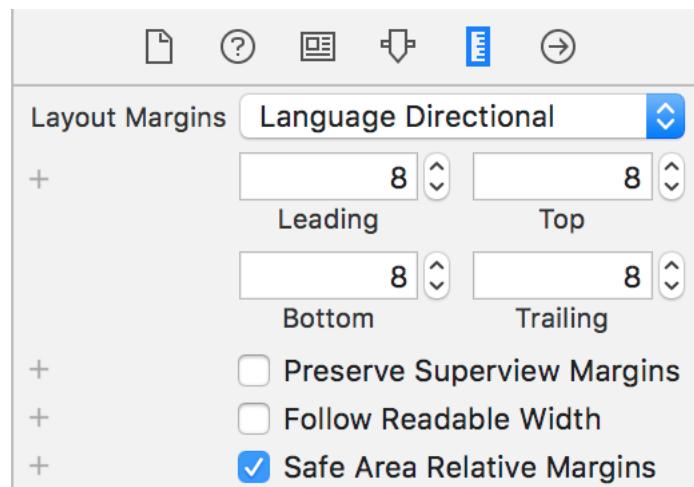
You have two choices depending on whether you need to deploy to targets earlier than iOS 11 - "Fixed" or "Language Directional":



If you need to support iOS 10 or earlier use the "Fixed" layout margins to set left, top, right and bottom insets (UIEdgeInsets):



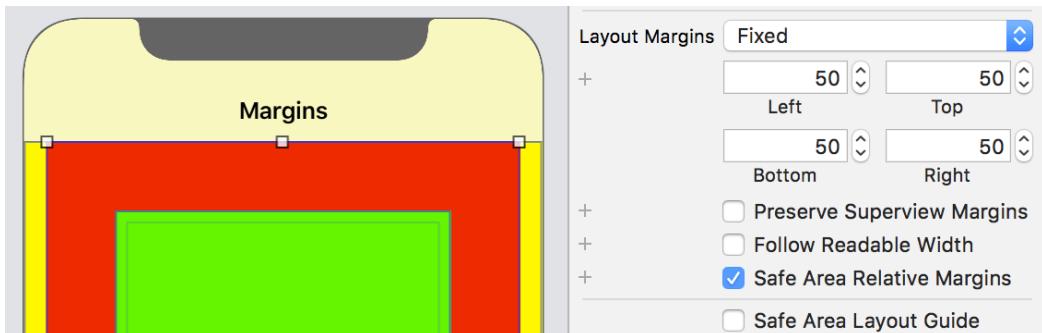
If your minimum deployment target is iOS 11 use the "Language Directional" layout margins which replace the left and right insets with leading and trailing insets (NSDirectionalEdgeInsets):



The system applies the margins based on the interface language direction. The leading margin is on the left for left-to-right languages and on the right for right-to-left languages.

To see an example with Interface Builder let's change our nested view example to increase the margins of the red view from the default 8 points to 50 points (see sample code: [Margins-v2](#)):

I set the deployment target for this project to iOS 9, so I have to use the fixed margins:



Xcode warns you if you use the directional layout margins before iOS 11. See [Using Margins In Programmatic Layouts](#) for an example of how to set the directional margins when available.

Changing The Root View Margins (iOS 11)

Unlike other views, the system manages the margins of a view controller's root view. By default, it enforces minimum left and right margins of either 16 or 20 points depending on the view width. The top and bottom margins are by default zero.

Before iOS 11 you could not change the margins of the root view. If you try to change the root view margins on iOS 10 and earlier, it has no effect. Starting in iOS 11 you can change the layout margins of the root view and control whether the system enforces a minimum margin.

The `systemMinimumLayoutMargins` property of the view controller gives the minimum system margins:

```
// iPhone X (iOS 11) portrait
print(systemMinimumLayoutMargins)
NSDirectionalEdgeInsets(top: 0.0, leading: 16.0, bottom: 0.0,
    trailing: 16.0)
```

Set `viewRespectsSystemMinimumLayoutMargins` to `false` in the view controller if you want a margin less than the system minimum. There doesn't appear to be a way to do this with Interface Builder:

```
if #available(iOS 11, *) {
    viewRespectsSystemMinimumLayoutMargins = false
    view.directionalLayoutMargins = NSDirectionalEdgeInsets(top:
        0.0, leading: 8.0, bottom: 0.0, trailing: 8.0)
}
```

The safe area margins may also increase the final layout margins if the view is using safe area relative margins. See [Safe Area Relative Margins \(iOS 11\)](#).

Using Margins In Programmatic Layouts

I've mostly used Interface Builder to explain layout margins so let's repeat our nested view example with constraints created in code. Let's also make this deployable back to iOS 9 (see sample code: [Margins-v3](#)):

1. Start a new Xcode project without a storyboard (see [Xcode Project And File Templates](#)). Add a new file (File > New > File...). Choose the iOS Cocoa Touch Class file template, name the class `NestedView` and save the file.
2. Follow our usual template for a custom view class:

```
// NestedView.swift
import UIKit

class NestedView: UIView {

    override init(frame: CGRect) {
```

```

    super.init(frame: frame)
    setupView()
}

required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
    setupView()
}

private func setupView() {
    // Build views
    // Add constraints
}
}

```

3. Add properties for the nested view and its color. I made the nested view a private property, but the user of the class can set its color. It defaults to green:

```

var nestedColor: UIColor = .green {
    didSet {
        nestedView.backgroundColor = nestedColor
    }
}

private lazy var nestedView: UIView = {
    let view = UIView()
    view.translatesAutoresizingMaskIntoConstraints = false
    view.backgroundColor = nestedColor
    return view
}()

```

4. In `setupView()` add the nested view to the superview and activate the four constraints to pin the nested subview to the margins of the superview using `layoutMarginsGuide`:

```

private func setupView() {
    addSubview(nestedView)
    NSLayoutConstraint.activate([
        nestedView.leadingAnchor.constraint(equalTo:
            layoutMarginsGuide.leadingAnchor),
        nestedView.topAnchor.constraint(equalTo:
            layoutMarginsGuide.topAnchor),
        nestedView.trailingAnchor.constraint(equalTo:
            layoutMarginsGuide.trailingAnchor),
        nestedView.bottomAnchor.constraint(equalTo:
            layoutMarginsGuide.bottomAnchor),
    ])
}

```

```
        nestedView.bottomAnchor.constraint(equalTo:  
layoutMarginsGuide.bottomAnchor)  
    ])  
}
```

5. The view controller takes care of creating the nested view, setting colors and margins and then pinning the nested view to the superview. First, define the margin, create the nested view and give it a red background:

```
class ViewController: UIViewController {  
  
    private let margin: CGFloat = 50.0  
  
    private let nestedView: NestedView = {  
        let view = NestedView()  
        view.translatesAutoresizingMaskIntoConstraints = false  
        view.backgroundColor = .red  
        return view  
    }()
```

6. Then in the `setupView()` method of the view controller add the nested view to the root view, set the margins and activate four constraints pinning the view to the top and bottom safe areas and leading and trailing margins of the root view:

```
private func setupView() {  
    view.backgroundColor = .yellow  
    view.addSubview(nestedView)  
    changeNestedMargins(inset: margin)  
  
    NSLayoutConstraint.activate([  
        nestedView.topAnchor.constraint(equalTo:  
safeTopAnchor),  
        nestedView.bottomAnchor.constraint(equalTo:  
safeBottomAnchor),  
        nestedView.leadingAnchor.constraint(equalTo:  
view.layoutMarginsGuide.leadingAnchor),  
        nestedView.trailingAnchor.constraint(equalTo:  
view.layoutMarginsGuide.trailingAnchor)  
    ])  
}
```

Note to make this work back to iOS 9 I'm using the same `UIViewController` extension we saw when looking at [Top And Bottom Layout Guides](#) to allow my top and bottom safe area anchors to fallback to

the top and bottom layout guides.

- When changing the margin of the red nested view, set the leading and trailing insets of the `directionalLayoutMargins` when available from iOS 11 onwards or fallback to setting left and right insets with `layoutMargins`:

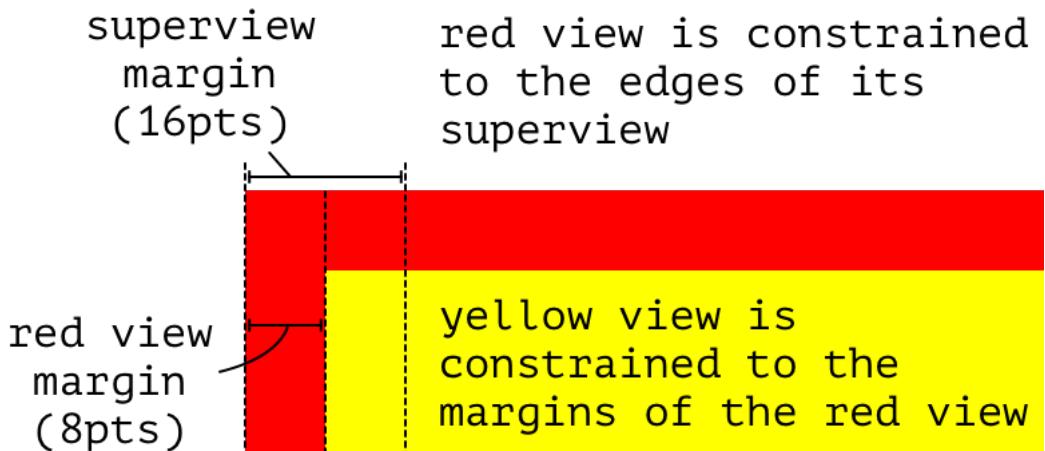
```
private func changeNestedMargins(inset: CGFloat) {
    if #available(iOS 11, *) {
        nestedView.directionalLayoutMargins =
        NSDirectionalEdgeInsets(top: inset, leading: inset,
        bottom: inset, trailing: inset)
    } else {
        nestedView.layoutMargins = UIEdgeInsets(top: inset,
        left: inset, bottom: inset, right: inset)
    }
}
```

PreservingSuperview Layout Margins

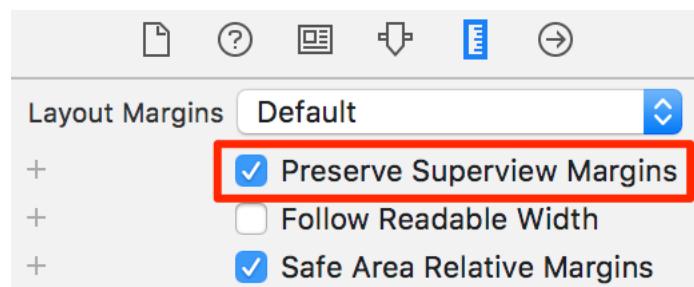
Here's a layout where I constrained a red container view to the edges of a view controller's root view. The yellow subview is constrained to the leading and trailing margins of the red view:



The red container view has default margins of 8 points which puts the yellow content view inside the 16 point default margin of the root view:



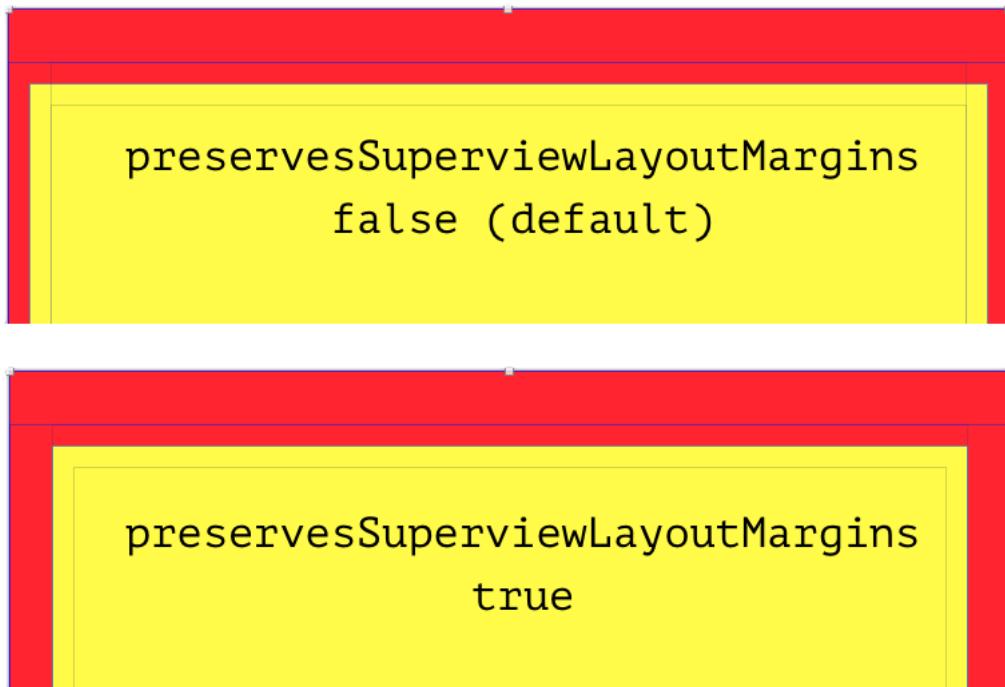
You can make the red view margin respect the larger margin of its superview by setting the `preservesSuperviewLayoutMargins` property for the red view. The default is `false`. You can set the property with Interface Builder using the Size inspector:



To set the property in code:

```
redView.preservesSuperviewLayoutMargins = true
```

The result is to increase the margin of the red view to keep the yellow content view inside the 16 point margin of the superview:



It takes a specific set of circumstances for this to be necessary:

- the container view is constrained close to the edges of its superview placing it inside the margins of the superview.
- the content view is constrained to the margins of the container view.
- the container view margins are smaller than its superview margins placing the content view inside the superview margins.

If the content view has constraints to the edges of the container view, the property has no effect.

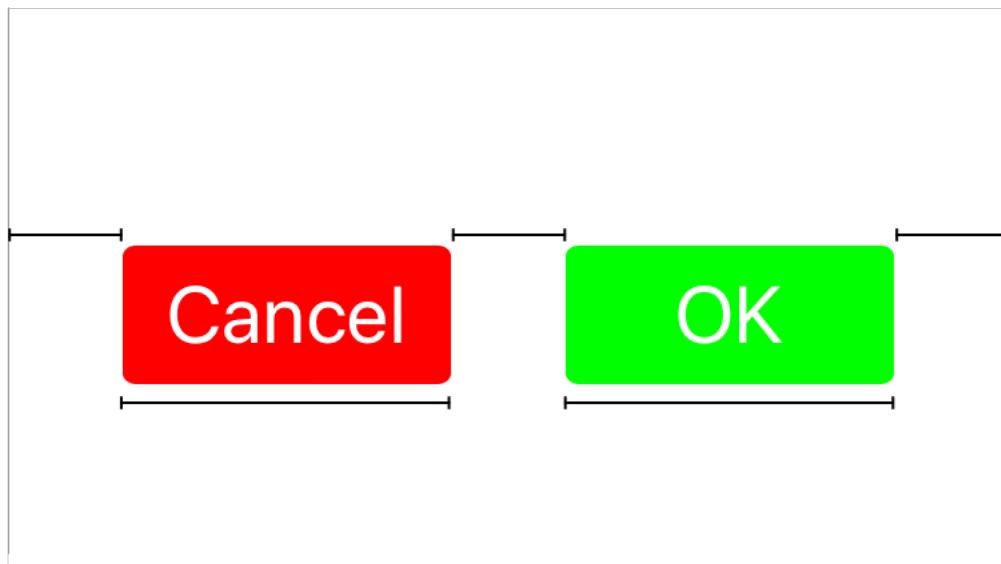
Layout Guides

The use of hidden spacer or dummy views is a well-used layout technique to control the spacing between views or layout groups of views. The disadvantage is that spacer views are still real views in the view hierarchy consuming memory and able to respond to events. The introduction of the `UILayoutGuide` class in iOS 9 allows us to do the same job without the overhead.

Unfortunately, you cannot create layout guides in Interface Builder. So before we look at using layout guides let's see how we might create a layout with Interface Builder that uses dummy spacer views.

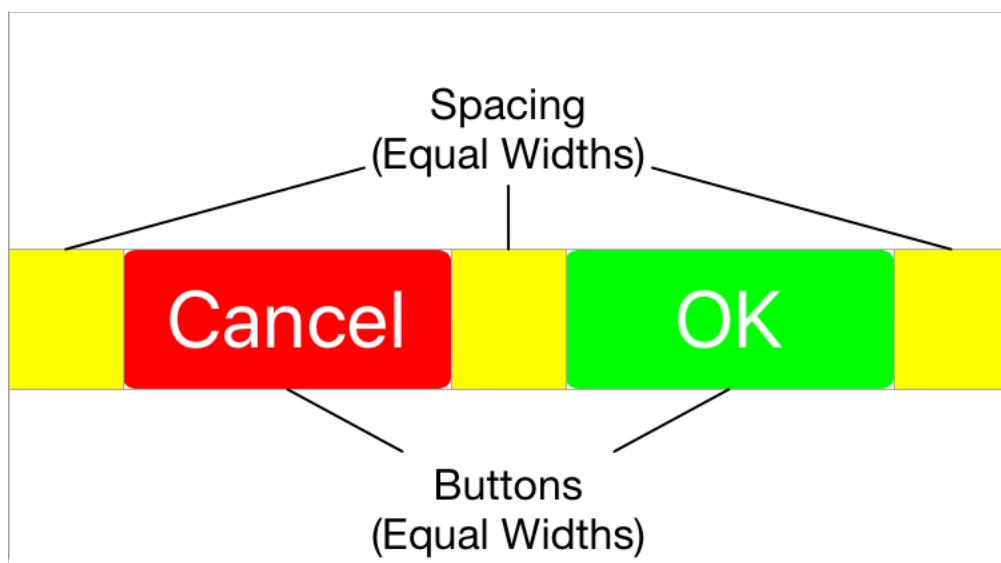
Equal Spacing With Interface Builder

This layout has two buttons of equal width with the spacing evenly distributed on each side of the buttons.

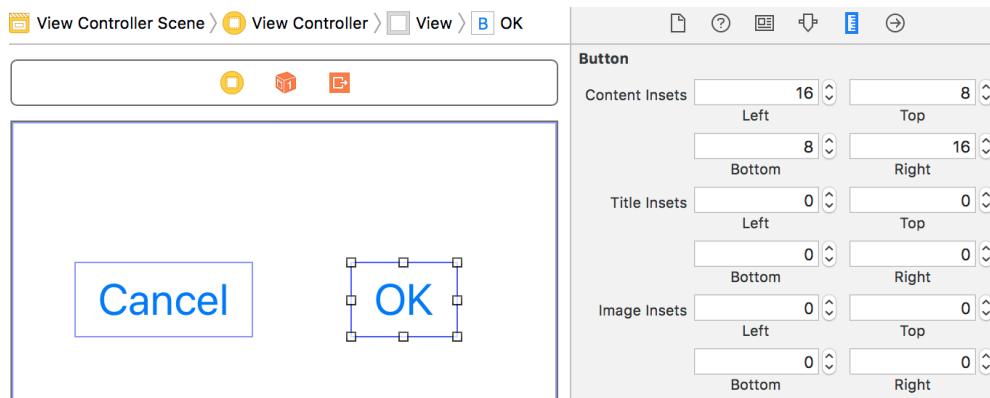


This type of layout is often confused with an equal centering layout. For example, fixing the centers of the buttons at some percentage of the center along the x-axis. The difference is that centering doesn't maintain equal spacing when the superview width changes.

You cannot create constraints on space. Instead, we add dummy spacer views where we want the space, shown in yellow below (see sample code: [EqualSpacing-v1](#)):

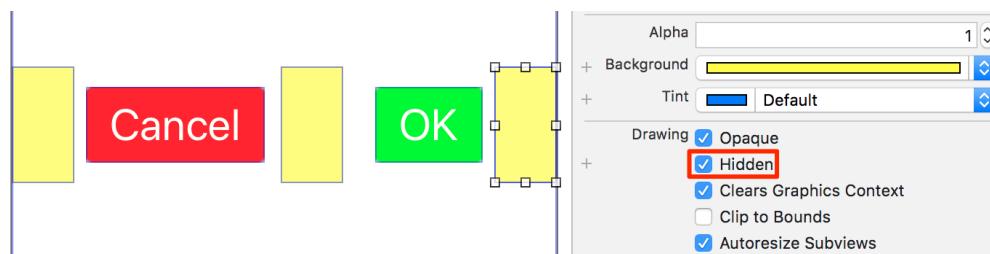


1. Create a new Xcode project using the Single View App iOS application template.
2. Drag two buttons into the Interface Builder canvas. Change the button text to “Cancel” and “OK” and increase the font size to 30 points. I also added some content insets to increase the internal padding (16 points left and right, 8 points top and bottom):

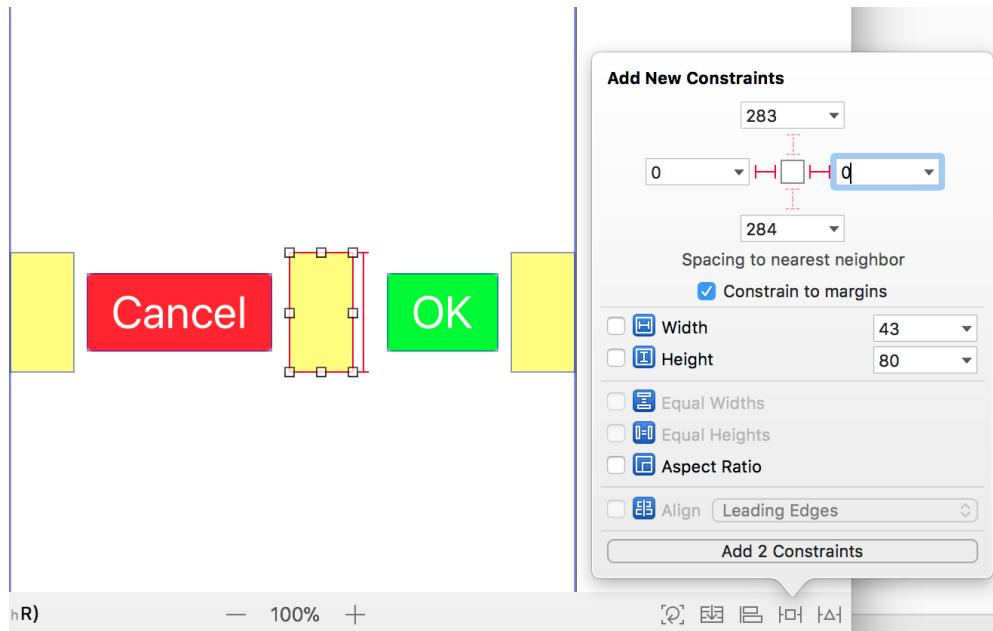


If label width is truncating the text, remember you can use **Cmd + =** to size the button to fit its contents. The OK button is smaller than the Cancel button at this point.

3. To make the two buttons stand out, I’m using red and green tinted background images. The template image I used is in the sample code.
4. Add three plain views to the canvas to act as the spacer views. Position and size them roughly between the buttons and make them hidden. I gave them a yellow background to make them easier to see in the canvas.

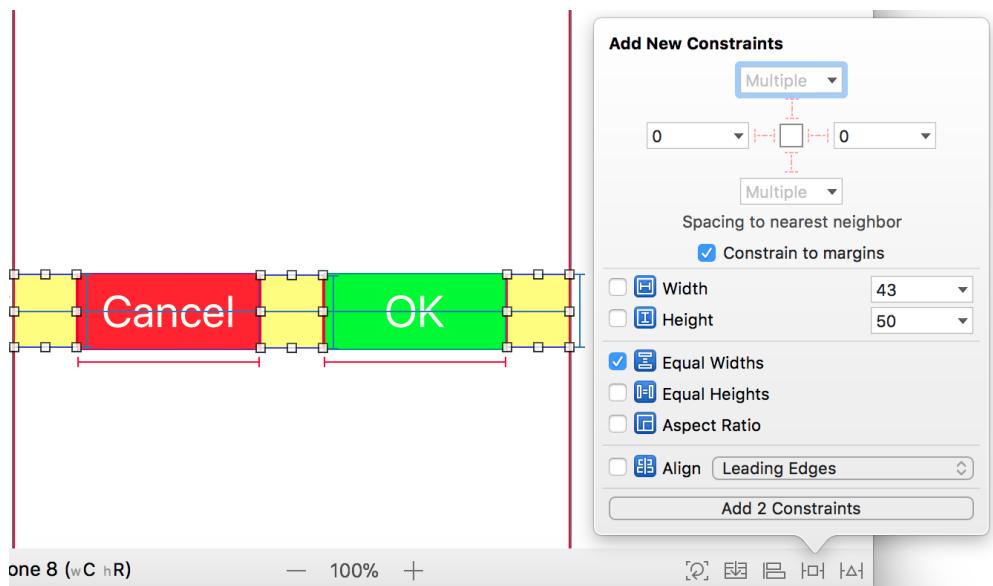


5. Add the leading and trailing constraints first. Click on the first spacer view and add a leading and trailing constraint with zero spacing. I find the **Add New Constraints** tool works well for this.



Repeat for the other two spacer views.

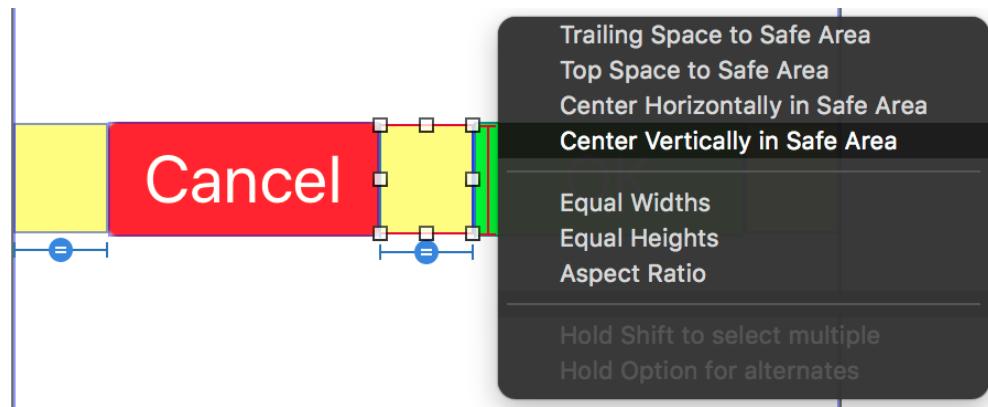
- With the three spacer views selected use the Add New Constraints tool to add two equal width constraints.



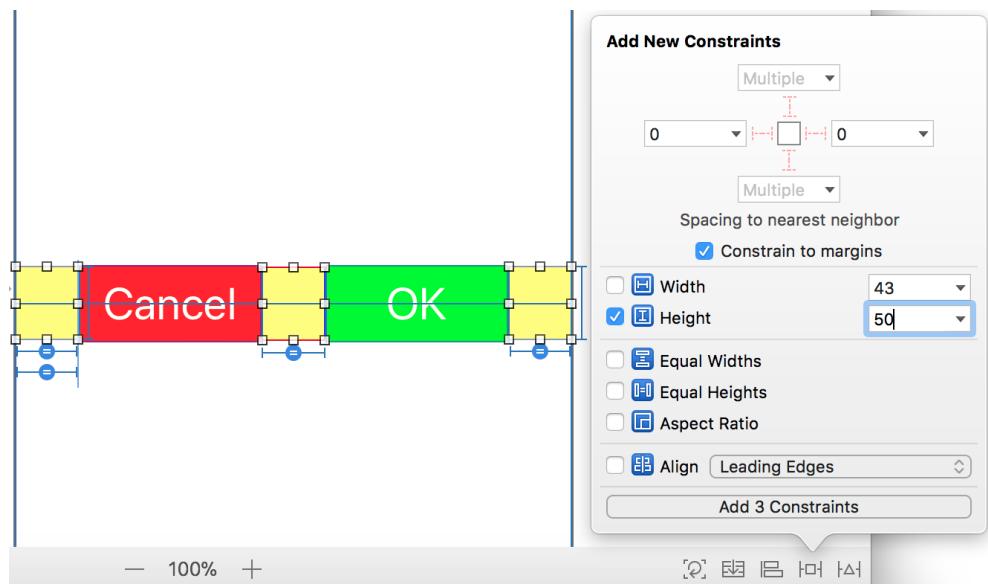
Add an equal width constraint for the two buttons in the same way.

- Add constraints to center each of the five views vertically within the safe area. For each view, control-drag vertically in the canvas

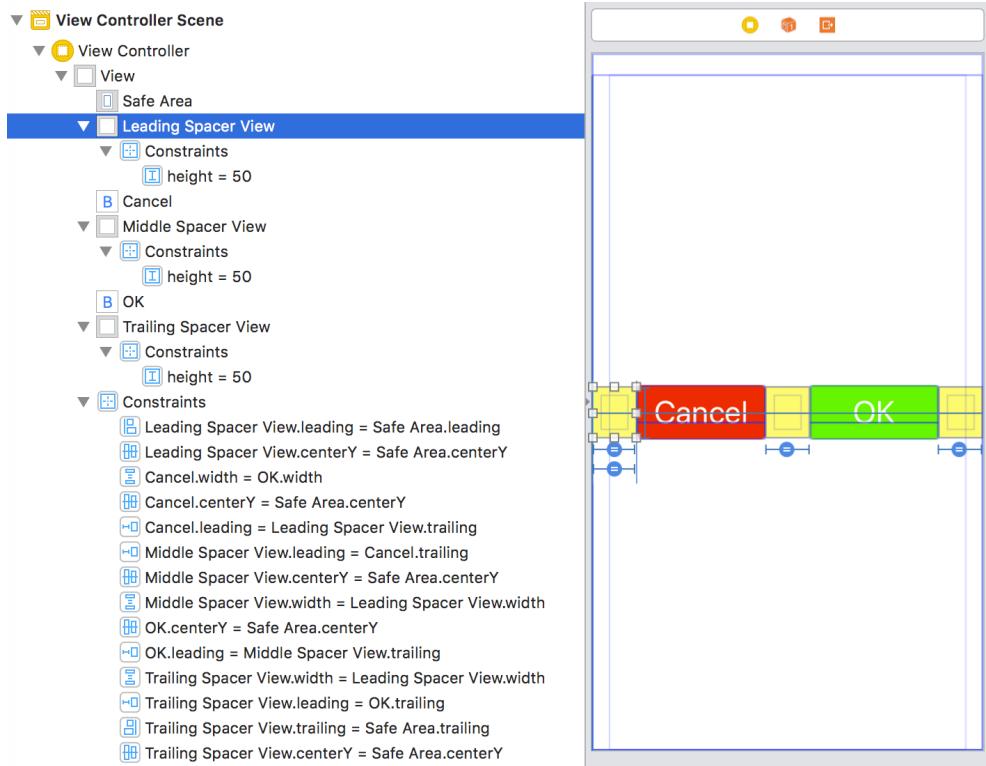
and select the Center Vertically in Safe Area option from the menu.



- Finally, the spacer views need a height. These views are not intended to be visible, so it doesn't matter what height we use. You can make them zero height, but I used 50 points so you can still see the spacer views in Interface Builder.



- That took some work. A total of 17 constraints:



Equal Spacing With Layout Guides

Let's recreate the equal spacing layout with layout guides. We need to build this in code as Interface Builder doesn't support creating layout guides (see sample code: [EqualSpacing-v2](#)):

1. Create a new Xcode project and save it. I'll skip the details here but assume I made both buttons properties of my view controller:

```
class ViewController: UIViewController {
    private lazy var cancelButton: UIButton = { }
    private lazy var okButton: UIButton = { }
```

2. We follow the usual pattern to set up our views. First add the buttons to the view hierarchy:

```
override func viewDidLoad() {
    super.viewDidLoad()
    setupView()
}

private func setupView() {
    view.addSubview(cancelButton)
```

```
    view.addSubview(okButton)
}
```

3. Before we add the constraints, create the three layout guides. In `setupView`:

```
let leadingGuide = UILayoutGuide()
let middleGuide = UILayoutGuide()
let trailingGuide = UILayoutGuide()
```

4. Remember that layout guides are not part of the view hierarchy. We don't add them to the view hierarchy with `addSubview` but they do need an owning view. Use the `addLayoutGuide` method to add the guides to the root view:

```
view.addLayoutGuide(leadingGuide)
view.addLayoutGuide(middleGuide)
view.addLayoutGuide(trailingGuide)
```

5. With the views and layout guides created we can add our constraints. A layout guide has layout anchors just like a regular view. So working from left-to-right starting with the leading layout guide which controls the space between the leading edge and the cancel button:

```
NSLayoutConstraint.activate([
    view.leadingAnchor.constraint(equalTo:
        leadingGuide.leadingAnchor),
    leadingGuide.trailingAnchor.constraint(equalTo:
        cancelButton.leadingAnchor),
```

6. Add constraints for the middle guide to set the spacing between the two buttons:

```
    cancelButton.trailingAnchor.constraint(equalTo:
        middleGuide.leadingAnchor),
    middleGuide.trailingAnchor.constraint(equalTo:
        okButton.leadingAnchor),
```

7. Then the trailing layout guide for the space between the OK button and the trailing edge:

```
    okButton.trailingAnchor.constraint(equalTo:
        trailingGuide.leadingAnchor),
```

```
trailingGuide.trailingAnchor.constraint(equalTo:  
    view.trailingAnchor),
```

8. The two buttons need to be of equal width:

```
cancelButton.widthAnchor.constraint(equalTo:  
    okButton.widthAnchor),
```

9. We also want our three layout guides to have equal width:

```
leadingGuide.widthAnchor.constraint(equalTo:  
    middleGuide.widthAnchor),  
    leadingGuide.widthAnchor.constraint(equalTo:  
    trailingGuide.widthAnchor),
```

10. Finally, we set the vertical position of the buttons centering them within the safe area guide:

```
cancelButton.centerYAnchor.constraint(equalTo:  
    view.safeAreaLayoutGuide.centerYAnchor),  
    okButton.centerYAnchor.constraint(equalTo:  
    view.safeAreaLayoutGuide.centerYAnchor),
```

Note that unlike the spacer views in Interface Builder we don't care about fixing the height and vertical position of the layout guides as neither impacts our layout.

11. Build and run. Here's how it looks in landscape on an iPhone 8:



We have the same layout but without the overhead of having the spacer views in the view hierarchy.

Key Points To Remember

If you're only supporting iOS 11 or later:

- Create constraints to the safe area layout guide of the superview for content subviews that you don't want to be covered by bars or clipped by the rounded corners of an iPhone X style device.
- If you want some extra padding inside the safe area create your constraints to the margins of the superview. (Margins allow for the safe area by default in iOS 11).
- Use the directional layout margins for right-to-left language support.
- You can change the margins of the root view.

If you need to support iOS 9 or iOS 10:

- If you create your layout in Interface Builder you can continue to constrain your content to the safe area layout guide. Interface Builder takes care of making this backward compatible for iOS 9 and iOS 10.
- If you create your constraints in code you need to take care of falling back to using the top and bottom layout guides and leading and trailing edges when the safe area layout guide is not available.
- Remember that margins don't take into account the top and bottom layout guides in iOS 9 and iOS 10. If you constrain your content to a margin, a parent view may cover your content.
- You cannot change the margins of the root view.

In both cases:

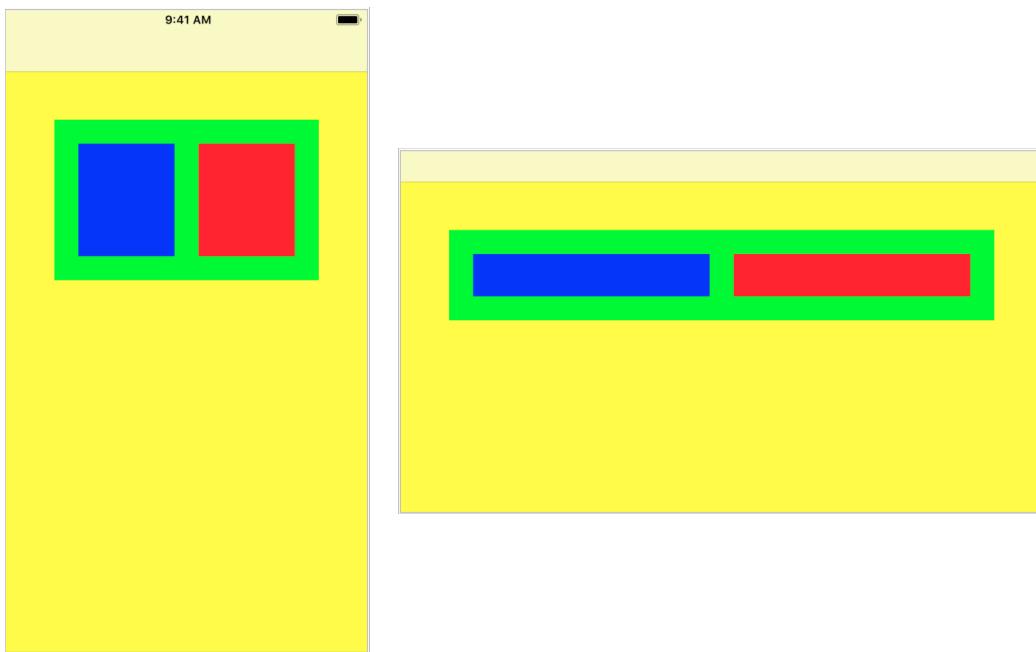
- Create constraints to the edges of the superview for background subviews where you don't care if something clips or covers the view. For example, a background image that you want to fill the screen.

Test Your Knowledge

Practice using margins and the safe area with these challenges.

Challenge 6.1 Margins In Interface Builder

Let's revisit the tile view from earlier chapters and use margins for the external spacing constraints. Here's the layout, embedded in a navigation controller:



The green container view is 50 points inside the safe area on the leading, top and trailing sides and is 25% the height of the yellow root view. The blue and red subviews have 25 points of padding on all sides inside the green container view and have equal width.

1. Build this layout with Interface Builder and Auto Layout (no code needed).
2. You can require at least iOS 11 and take advantage of any features introduced with iOS 11.
3. Use margins in your constraints where possible. You should only need one constraint with a non-zero constant value.

Hints And Tips

1. Avoid using non-zero constant values in your constraints. Where possible, create a constraint to the margin and adjust the margin insets to give you the spacing.
2. To create constraints to the margins of the root view, I prefer to control-drag in the document outline between the subview and root view. Remember to hold down the Option key to switch between the safe area and container margin.
3. To create constraints to the margins of a non-root view that doesn't have the safe area layout guide enabled I like to use the Add New Constraint tool but experiment to find what works for you.

4. When you create a constraint be careful not to include extra spacing by mistake or constrain to an edge or safe area instead of the margin. Check your constraints in the document outline and edit any constraints that have additional constant values or that use the wrong item.
5. Margin constraints are relative to the safe area by default since iOS 11. Constraining a subview to the margin of its superview keeps the subview inside the safe area (this is not the case before iOS 11).
6. When adjusting the margins of the views use the language directional layout margins (needs at least iOS 11).
7. The horizontal spacing constraint between the blue and red views is the only constraint that needs a non-zero constant value.

Challenge 6.2 Backwards Compatibility With Interface Builder

I simplified the last challenge by requiring at least iOS 11. You have to work a little harder when you need to support earlier releases of iOS.

1. Create the layout from [Challenge 6.1](#) but this time set the deployment target for the project to be iOS 10.
2. As before create the layout in Interface Builder using Auto Layout (no code needed).
3. Your project should build without warnings and run on iOS 10 and up to iOS 12.
4. Test the layout using the simulator on both iOS 10 and iOS 12 and make sure the 50 points of spacing to the safe area works in both cases. Pay attention to the spacing below the navigation bar and at the leading and trailing edges of the root view.

Hints And Tips

1. The language directional layout margins are only available from iOS 11. Xcode warns you if you use them in a project that targets iOS 10. Use the fixed layout margins instead.
2. You cannot change the margins of the root view on iOS 10.
3. Margins are not relative to the safe area on iOS 10.
4. Replace the green tile view constraint to the top margin of the root view with a constraint to the safe area layout. Use a constant value

to get the 50 point spacing.

5. Replace the green tile view constraints to the leading and trailing margins of the root view with constraints to the safe area. Use a constant value to the 50 point spacing.
6. Interface Builder takes care of making the safe area backward compatible to iOS 10.

Challenge 6.3 Programmatic Margins

Finally, practice using the safe area and margins when creating your layouts with code.

1. Create the tile view layout from [Challenge 6.1](#) without using Interface Builder.
2. Prefer creating constraints to margins over using constant values for spacing.
3. You can require iOS 11 or iOS 12 to make it easier.

Hints And Tips

1. Use a custom `UIView` subclass for the green container view and its two subviews.
2. The view controller needs a property for the custom subview. Don't forget about `translatesAutoresizingMaskIntoConstraints`.
3. Set the `directionalLayoutMargins` for the root view and the custom subview in the view controller.
4. The view controller only needs to create four constraints to position and size the custom subview. Create the constraints for the blue and red subviews in the custom subview.

Challenge 6.4 Backwards Compatibility In Code

For bonus points can you change [Challenge 6.3](#) to work for iOS 10?

Hints And Tips

1. You should not need to change the custom subview.
2. The view controller still needs to set the margins for the custom subview. The `directionalLayoutMargins` property needs at least

iOS 11 so use a `#available` test to fall back to `layoutMargins` for iOS 10.

3. The layout margins of the root view are not relative to the safe area in iOS 10. You also cannot change the root view margins.
4. You can no longer rely on the margins to keep your custom subview inside the safe area. Create your constraints to the safe area and use constants in your constraints for the spacing.
5. The safe area layout guide is only available from iOS 11. For iOS 10, use the top and bottom layout guides and the leading and trailing anchors. See [Checking For Safe Area Availability](#) for one way to do that.

Chapter 7

Layout Priorities And Content Size

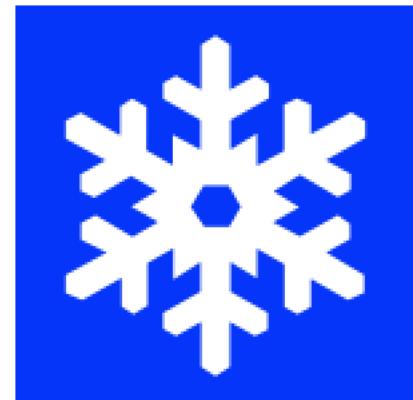
You cannot progress far with Auto Layout without understanding how to use layout priorities. In this chapter you learn:

- How to use priorities to create optional constraints and when to use them.
- How the standard UIKit controls can have a natural, intrinsic size to fit their content.
- How to use the `contentMode` property to control the scale and position of the contents of a view when the bounds of the view change.
- How to use content-hugging or compression-resistance priorities to stretch or squeeze the natural size of views to fit a layout. An essential Auto Layout technique to master.

These topics are often the ones that cause people to dislike Auto Layout. Don't panic! Take it one step at a time always keeping one eye on what problems each new technique can solve for you.

Layout Priorities

The layout engine treats any constraints you create as required constraints by default. The layout engine must satisfy all required constraints, or the layout is invalid. Sometimes you want to have an optional constraint. Consider this layout where I have two images of unequal size. I want to put a label below the images:



This label should be
below the tallest of the
two images

If we know that the image on the right is always the tallest we can add a vertical spacing constraint from the top of the label to the bottom of the image:



What if we don't know until runtime which of the two images is the tallest? This is where optional constraints come to the rescue.

Optional And Required Priorities

All constraints have a layout priority from 1 to 1000. The priority is of type `UILayoutPriority` and `UIKit` helpfully defines constants for arbitrary "low" and "high" values which it uses as default values:

- `.defaultLow` (250)
- `.defaultHigh` (750)
- `.required` (1000)

Constraints with a priority lower than `.required` (1000) are optional. The layout engine tries to satisfy higher priority constraints first. When it cannot fully satisfy an optional constraint, it does its best to get as close as possible.



Once you have activated a constraint you cannot change its priority from required to optional or vice versa. Doing so causes a runtime crash. You can change the priority of an optional constraint, as long as you keep it optional (< 1000).

Returning to our layout how can we use optional constraints to position the label? Think first, what we can say about our desired layout?

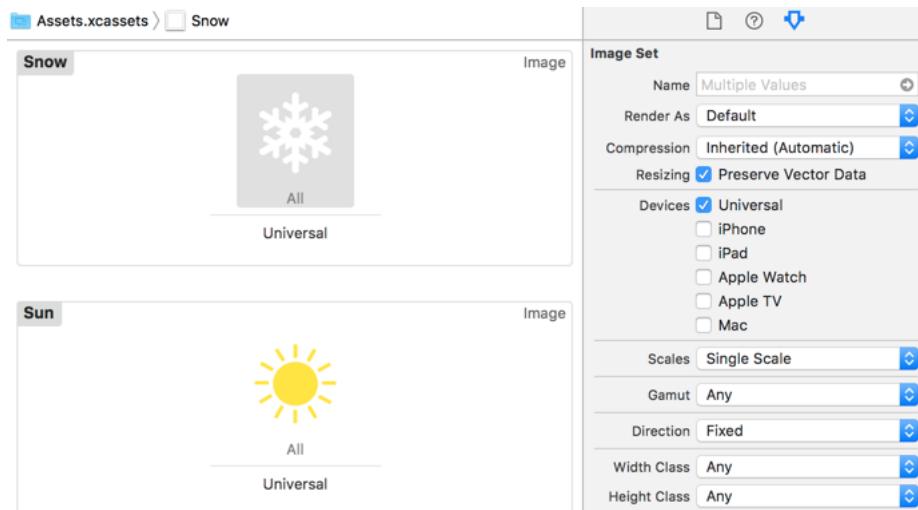
- The label should be at least a standard amount of spacing below the sun image.
- The label should be at least a standard amount of spacing below the snowflake image.
- The label should be as close to the top of the view as possible.

Words like **at least** or **at most** suggest a constraint using **inequalities**. A phrase like **as close as possible** suggests an **optional** constraint.

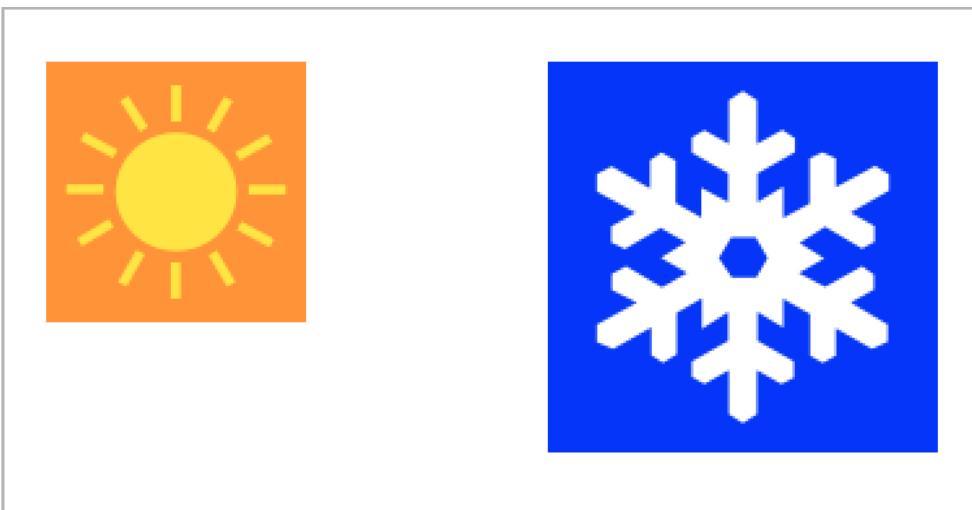
Optional Constraints In Interface Builder:

To create this layout, I'm using a sun image that's 100x100 points and a snowflake image that's 150x150 points (see sample code: [Priorities-v1](#)). The exact image size is not important. Use mine from the sample code or substitute them with your images:

1. Create a new Xcode project using the Single View App iOS template.
2. Add the sun and snow images from the sample code, or your images if you prefer, to the project asset catalog. I'm using PDF vector images:

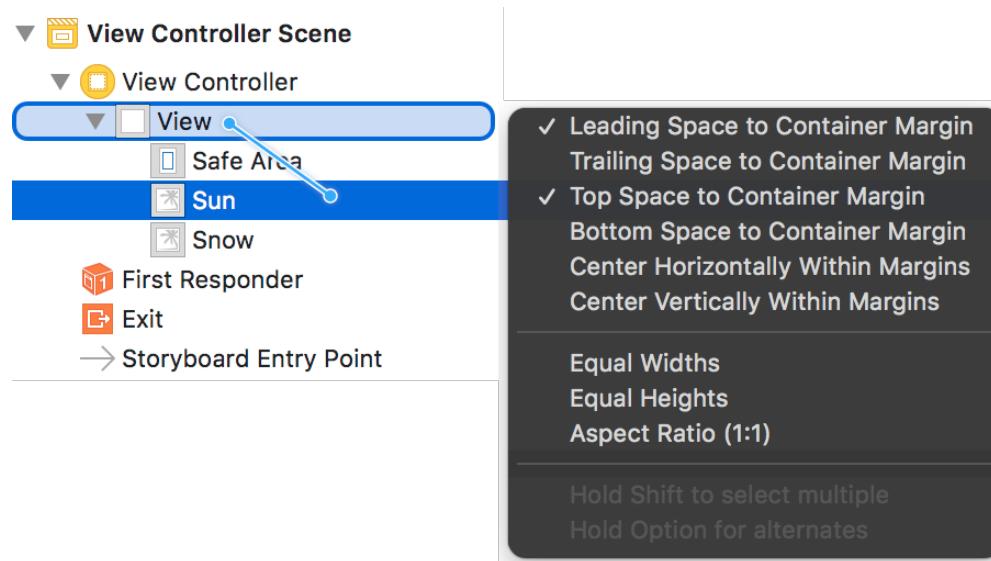


3. Drag two image views from the object library onto the view in the Interface Builder canvas. Position them against the margins in the top corners of the view and use the Attributes inspector to show the sun image on the left and the snow image on the right:

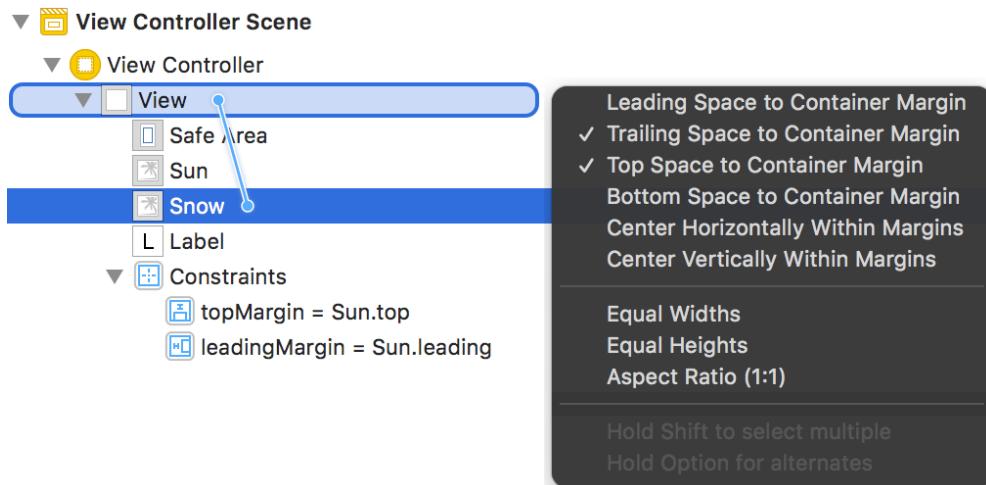


Change the background color of the sun image view to orange and the snowflake image view to blue.

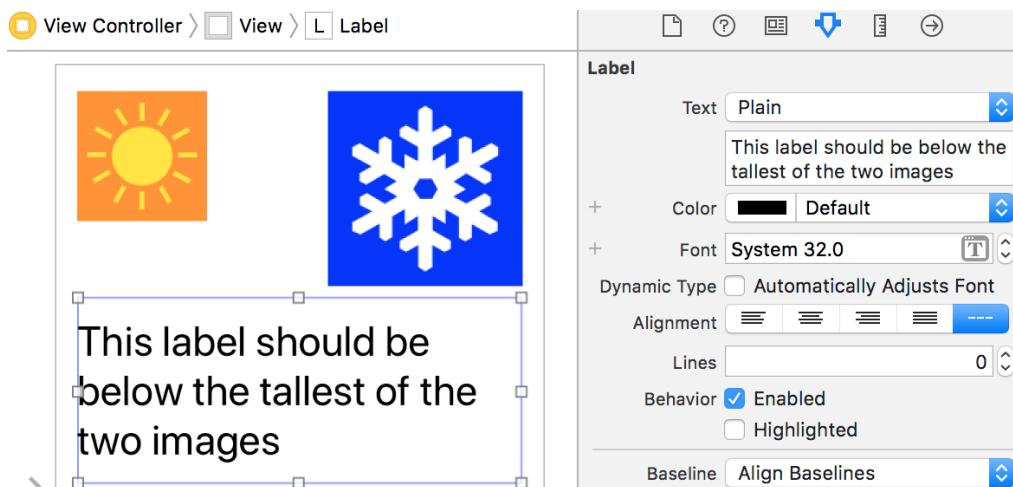
4. Fix the position of the sun image view by control-dragging from the image view to the root view in the document outline. Hold down the Option key and the Shift key to add constraints to the leading and top margins of the root view:



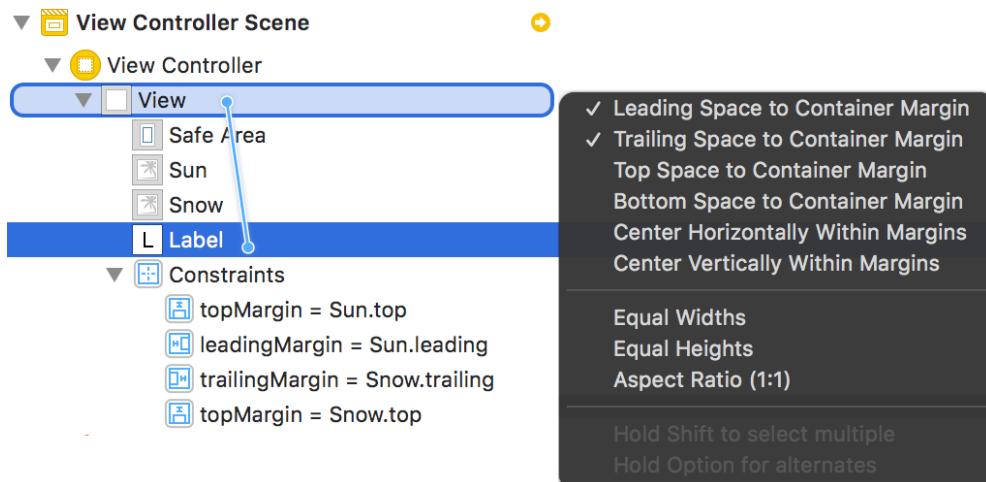
5. In the same way, pin the snowflake image view to the trailing and top margins of the root view:



6. Drag a label from the object library onto the canvas and position it below the two images. Add some text and increase the font size to 32 points. Size the label so that it fills the width between the leading and trailing margins. Set the number of lines for the label to zero to allow the text to wrap over multiple lines if necessary.

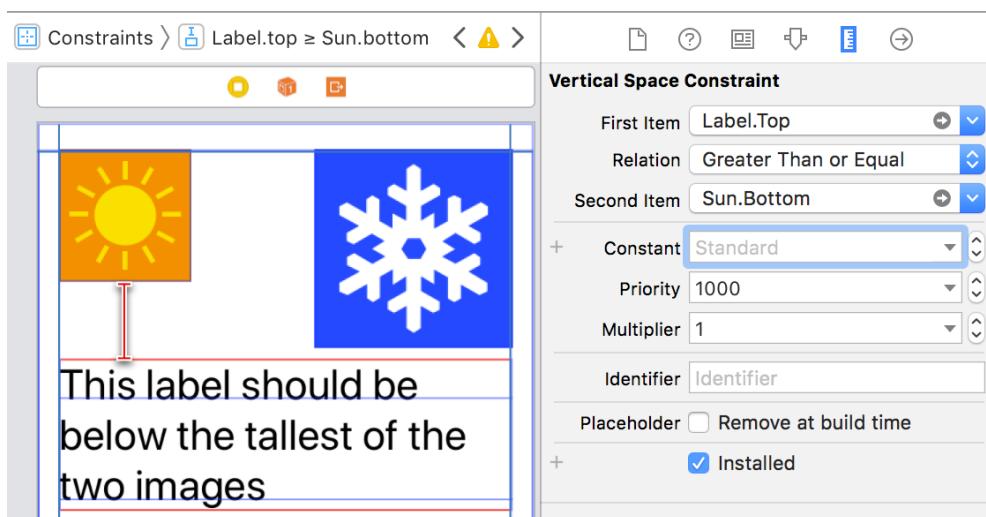


7. To fix the horizontal position control-drag from the label to the root view and create constraints to the leading and trailing margins.

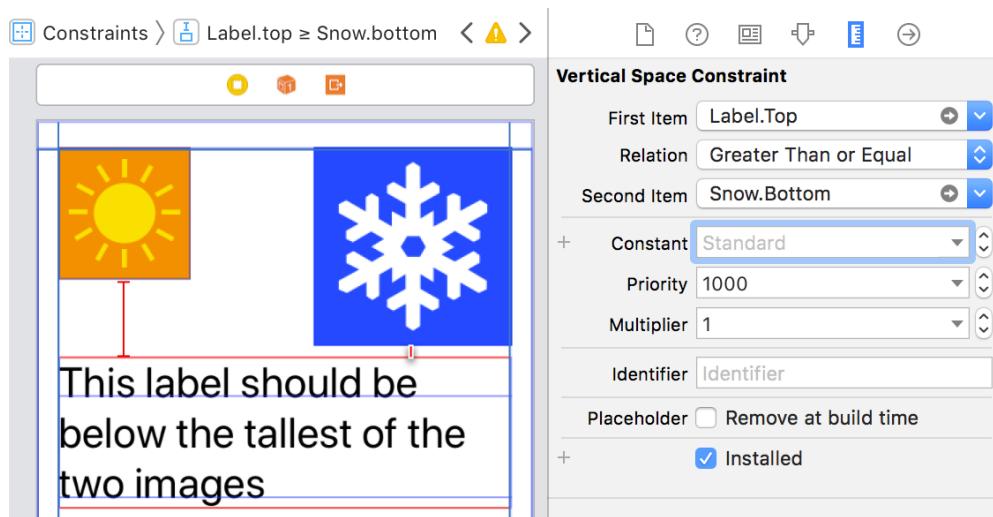


Note that this also fixes the width of the label.

8. Create the two inequality constraints that keep the label below the two images. First control-drag from the label to the sun image view and create a vertical spacing constraint.
9. Use the size inspector to change the relation of the constraint to "Greater Than Or Equal". Make sure the constant is using the "Standard Value":

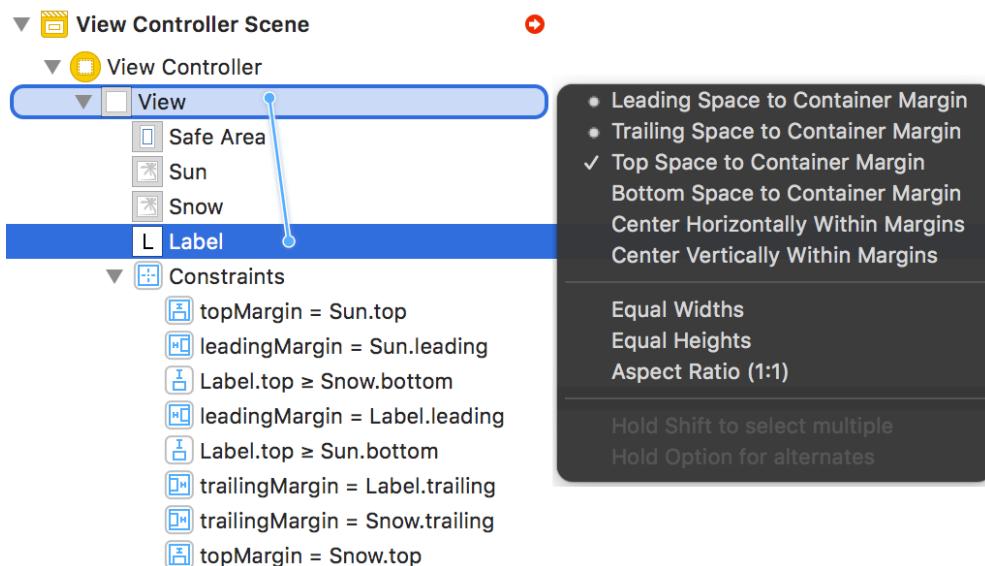


10. Create another "Greater Than Or Equal" inequality constraint from the label to the snowflake image:

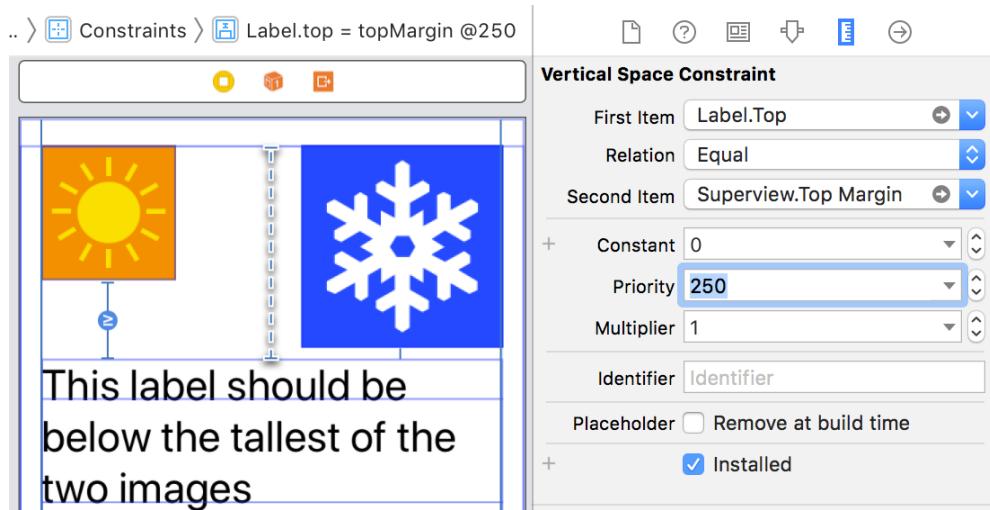


Both of these constraints are required constraints but because they are inequalities they don't fix the vertical position of the label.

11. To pull the label "as close as possible" to the top margin we need an optional constraint. Control-drag from the label to the root view in the document outline and with the Option key held down create a top space constraint to the container margin.



12. This constraint is not yet an optional constraint. Find and click on the constraint in the canvas or document outline and edit it using the attributes inspector. Change the priority to Low (250) and the constant value to zero:



Note how Interface Builder shows the optional constraint with a dotted line.

13. Build and run and check the label does stay below the tallest image.

Creating Optional Constraints In Code

Let's recreate the last example using a programmatic layout so we can see how to create an optional constraint in code (see sample code: [Priorities-v2](#)):

1. Start a new Xcode project (no storyboard needed) and add the two image resources to the asset catalog as in the last example.
2. I need two properties in the view controller for the image views. To avoid duplicating the setup code, we can add a convenience initializer to a private UIImageView extension in the view controller:

```
private extension UIImageView {
    convenience init(named name: String, backgroundColor: UIColor) {
        self.init(image: UIImage(named: name))
        self.backgroundColor = backgroundColor
        translatesAutoresizingMaskIntoConstraints = false
    }
}
```

Our two image view properties use this initializer:

```
private let sunView = UIImageView(named: "Sun",
    backgroundColor: .orange)
private let snowView = UIImageView(named: "Snow",
    backgroundColor: .blue)
```

3. We also need a label for the caption that's below the images:

```
private let captionLabel: UILabel = {
    let label = UILabel()
    label.translatesAutoresizingMaskIntoConstraints = false
    label.text = "This label should be below the tallest of
        the two images"
    label.font = UIFont.systemFont(ofSize: 32.0)
    label.numberOfLines = 0
    return label
}()
```

4. Build the view hierarchy in `setupView`:

```
override func viewDidLoad() {
    super.viewDidLoad()
    setupView()
}

private func setupView() {
    view.addSubview(sunView)
    view.addSubview(snowView)
    view.addSubview(captionLabel)
}
```

5. Create and activate the constraints starting with the sun view pinned to the top and leading margins (in `setupView()`):

```
let margins = view.layoutMarginsGuide
NSLayoutConstraint.activate([
    sunView.leadingAnchor.constraint(equalTo:
        margins.leadingAnchor),
    sunView.topAnchor.constraint(equalTo:
        margins.topAnchor),
```

6. Pin the snow view to the top and trailing margins:

```
snowView.topAnchor.constraint(equalTo:
    margins.topAnchor),
```

```
snowView.trailingAnchor.constraint(equalTo:  
margins.trailingAnchor),
```

7. Constrain the caption label to fill the width between the leading and trailing margins:

```
captionLabel.leadingAnchor.constraint(equalTo:  
margins.leadingAnchor),  
captionLabel.trailingAnchor.constraint(equalTo:  
margins.trailingAnchor),
```

8. Create the two inequality (\geq) constraints that position the label at least a standard amount of spacing below the two images:

```
captionLabel.topAnchor.constraintGreaterThanOrEqualToSystemSpacingBelow(sunView.bottomAnchor,  
multiplier: 1.0),  
captionLabel.topAnchor.constraintGreaterThanOrEqualToSystemSpacingBelow(snowView.bottomAnchor,  
multiplier: 1.0),
```

The system spacing methods were new in iOS 11. If you need to support iOS 10 and earlier replace these two constraints with 8 point constant constraints:

```
captionLabel.topAnchor.constraint(greaterThanOrEqualTo:  
sunView.bottomAnchor, constant: 8.0),  
captionLabel.topAnchor.constraint(greaterThanOrEqualTo:  
snowView.bottomAnchor, constant: 8.0),
```

9. Finally we need the optional top constraint for the label. First create the constraint from the label to the top margin as normal:

```
let captionTopConstraint =  
captionLabel.topAnchor.constraint(equalTo:  
margins.topAnchor)
```

10. Set the priority to `.defaultLow` (250) to make it optional:

```
captionTopConstraint.priority = .defaultLow
```

11. Then add it to the array of constraints to activate:

```
NSLayoutConstraint.activate([
```

```
// other constraints  
captionTopConstraint  
])
```



Set the priority of the optional constraint before you activate it to avoid a runtime exception.

Intrinsic Content Size

Some views have a natural size based on their content. A view wants to be this natural size unless you add constraints that stretch or squeeze it. **This natural size is also known as the intrinsic content size.**

Views without an intrinsic content size must have their width and height set with constraints.



All views have an `intrinsicContentSize` property. This is a `CGSize` with a width and a height. A view can use the value `UIViewNoIntrinsicMetric` when it has no intrinsic size for a dimension.

Standard UIKit Controls

Common UIKit controls like the button, label, switch, stepper, segmented control, and text field have both an intrinsic width and height:

Label

First

Second

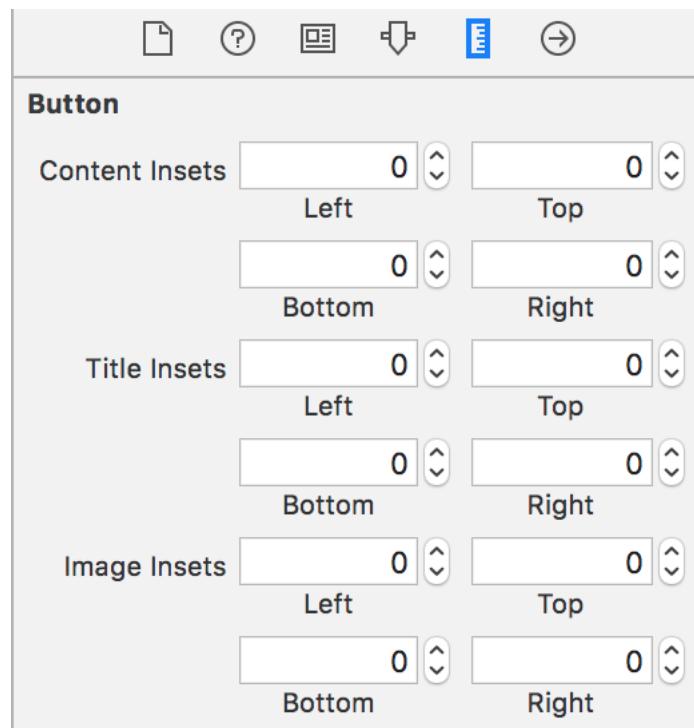
Button

Name



UIButton

The text and the font used can change the intrinsic content size of a button. If you use insets to add padding or move the image or title be aware that only `contentEdgeInsets` has any effect on the intrinsic content size.

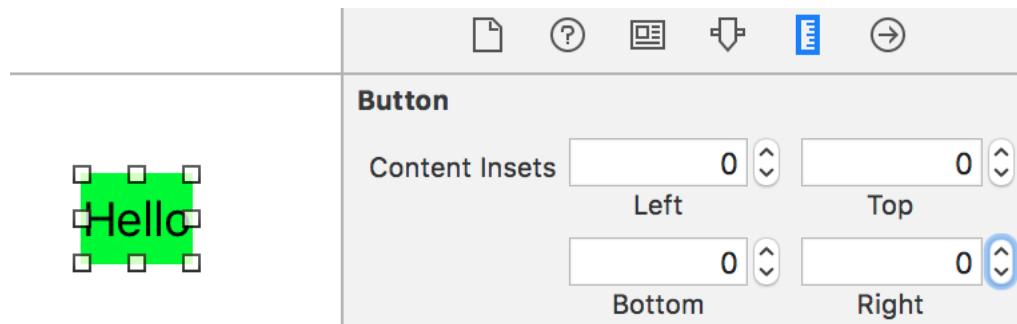


The `titleEdgeInsets` and `imageEdgeInsets` position the title and image during layout after the system sets the button size. They don't change the intrinsic content size.



To pad the size of a button add content edge insets rather than a fixed width or height constraint.

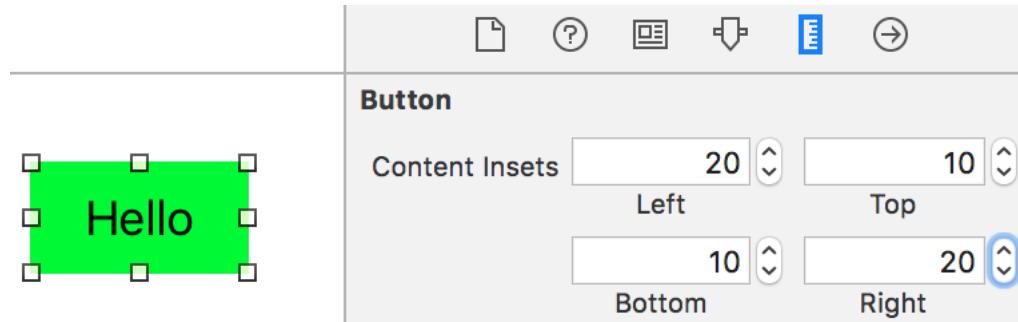
For example, suppose we have a custom button with default system font. We might create it in Interface Builder:



If we create the button in code and check its intrinsic content size. It has a size of 42 x 34:

```
let button = UIButton(type: .custom)
button.setTitle("Hello", for: .normal)
button.backgroundColor = .green
button.intrinsicContentSize // (w 42 h 34)
```

Adding top and bottom content insets of 10 points and bottom and left and right content insets of 20 points increases the intrinsic content size:



```
button.contentEdgeInsets = UIEdgeInsets(top: 10, left: 20,
    bottom: 10, right: 20)
button.intrinsicContentSize // (w 82 h 42)
```

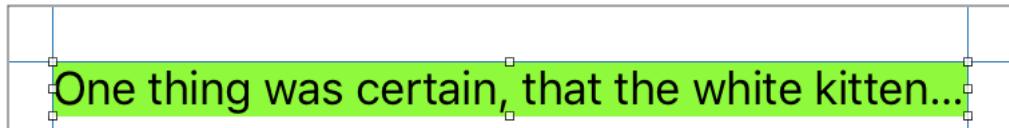
Changing layer properties like the corner radius doesn't change the intrinsic content size:

```
button.layer.cornerRadius = 10.0
button.intrinsicContentSize // (w 82 42)
```



UILabel

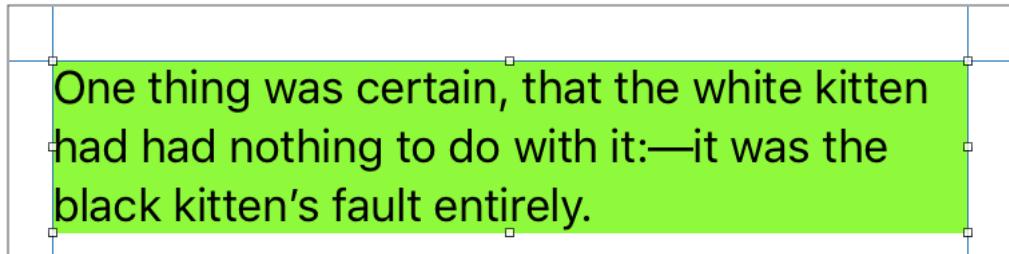
By default, a `UILabel` has an intrinsic content size that shows the text content on a single line. For long lines of text that can lead to truncation as here with a label constrained between the leading and trailing margins:



The “natural” intrinsic content size of the label is much wider than the visible bounds of the label:

```
label.intrinsicContentSize // (w 853 h 20.5)  
label.bounds.size // (w 343 h 20.5)
```

If you constrain the width of a label but leave the height free and set `numberOfLines` to 0 the intrinsic content size of the label adjusts for the number of lines needed to show the full text:



```
label.intrinsicContentSize // (w 333 h 64.5)  
label.bounds.size // (w 343 h 64.5)
```

UISlider and UIProgressView

The `UISlider` and its close relation the `UIProgressView` are unusual for `UIKit` controls in that they only have an intrinsic height, not a width. The thumb of a slider and the track of the progress view set their heights, but you must add constraints to fix the width of the track.



```
let slider = UISlider()  
slider.intrinsicContentSize // (w -1 h 30)
```

UIImageView

The size of the image sets the intrinsic content size of an image view. An empty image view doesn't have an intrinsic content size.

```
let imageView = UIImageView()
imageView.intrinsicContentSize // (w -1 h -1)

imageView.image = UIImage(named: "Star")
imageView.intrinsicContentSize // (w 100 h 100)
```

UITextView

A text view that has scrolling enabled doesn't have an intrinsic content size. You must constrain the width and height. The text view shows the text in the available space scrolling if needed.

With scrolling disabled a text view acts as a `UILabel` with `numberOfLines` set to zero. Unless you constrain the width a text view has the intrinsic content size for a single line of text (assuming no carriage returns). Once you constrain the width, the text view sets its intrinsic content size height for the number of lines needed to show the text at that width.

Views With No Intrinsic Content Size

A plain old `UIView` has no content so has no intrinsic content size. Scroll views, web views and text views with scrolling allowed don't have an intrinsic content size. You must add constraints for the width and height. The view then shows its content in the available space scrolling if necessary.

Custom Views

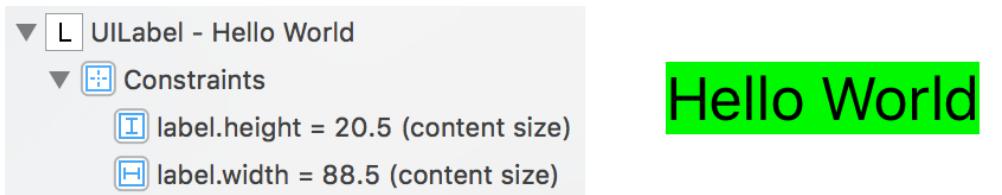
When you create a custom subclass of `UIView`, you can choose to override `intrinsicContentSize` and return a size based on the content of your custom view. If your view doesn't have a natural size for one dimension return `UIViewNoIntrinsicMetric` for that dimension:

```
// Custom view with an intrinsic height of 100 points
class CustomView: UIView {
    override var intrinsicContentSize: CGSize {
        return CGSize(width: UIViewNoIntrinsicMetric, height: 100)
    }
    // ...
}
```

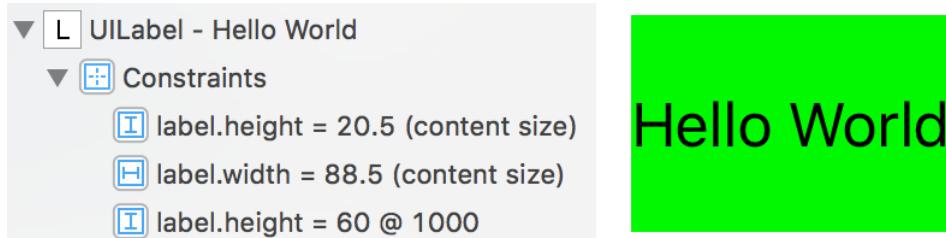
If the intrinsic content size of your custom view changes tell the layout engine by calling `invalidateIntrinsicContentSize()`.

Intrinsic Content Size In The View Debugger

The Auto Layout engine creates special constraints for the intrinsic content size of a view. If you're curious, you can see these constraints in the view debugger. They are of type `NSContentSizeLayoutConstraint` which is a private subclass of `NSLayoutConstraint` so you cannot create them directly:



The intrinsic content size constraints are always labeled `(content size)` so they stand out against normal height and width constraints. For example, if I add a constraint to this label to fix the height at 60:

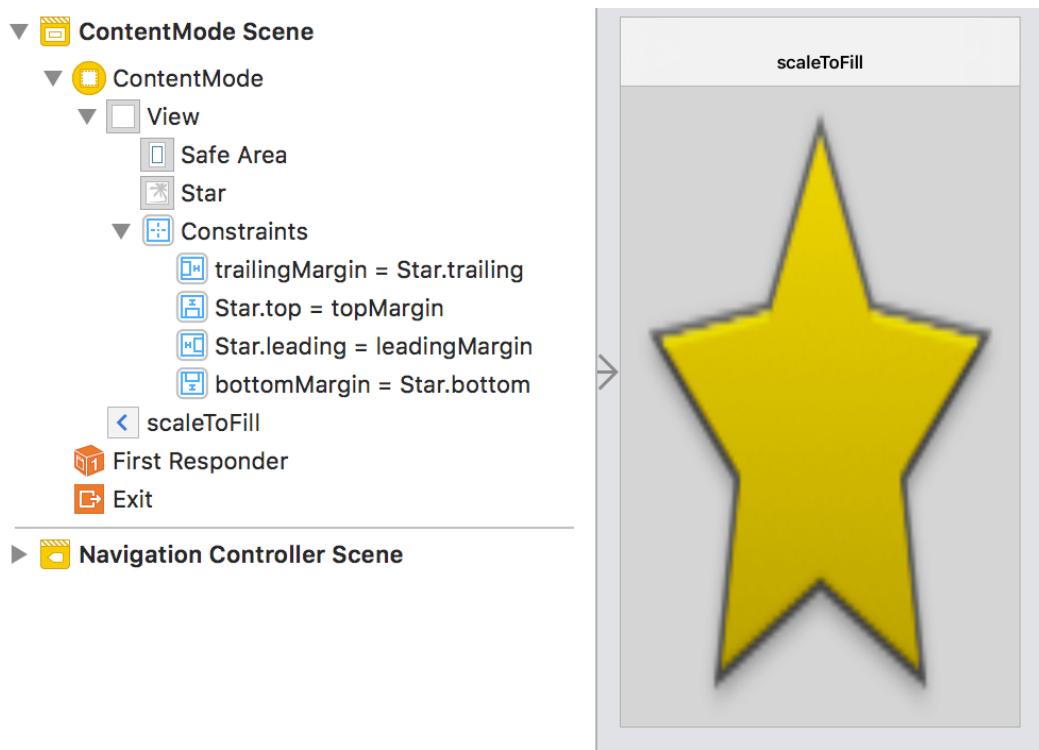


Our height constraint overrides the intrinsic height and stretches the label beyond its natural height.

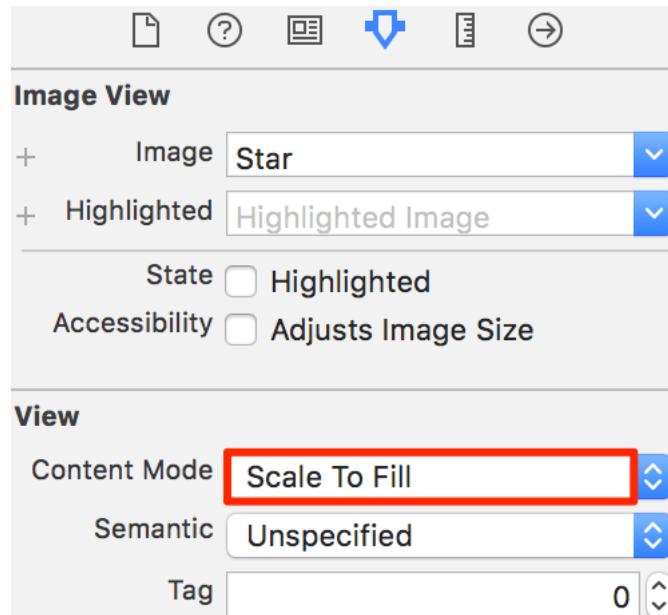
Content Mode

The `contentMode` property of `UIView` controls how to adjust a view when its bounds change. The system will not, by default, redraw a view each time the bounds change. That would be wasteful. Instead, depending on the content mode, it can scale, stretch or keep the contents in a fixed position.

To see how this works I have an image view containing a vector image of a star pinned to the margins of the root view (see sample code: [ContentMode](#)):



In Interface Builder you set the content mode for a view using the Attributes inspector:



You can also set the content mode of a view in code:

```
imageView.contentMode = .scaleAspectFit
```

There are thirteen different content modes, but it's easiest to think of three main groups based on the effect:

- Scaling the content (with or without maintaining the aspect ratio)
- Positioning the content
- Redrawing the content

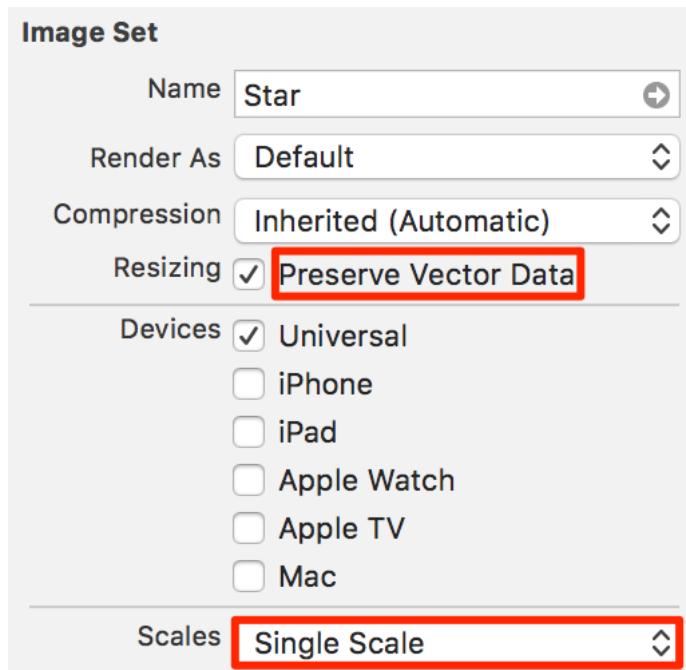
Scaling the View

Three modes have the effect of scaling or stretching the view contents to fill the available space when the bounds changes.

- `.scaleToFill`: Stretches the content to fill the available space without maintaining the aspect ratio. The default mode.
- `.scaleAspectFit`: Scales the content to fit the space maintaining the aspect ratio.
- `.scaleAspectFill`: Scales the content to fill the space maintaining the aspect ratio. The content can end up larger than the bounds of the view resulting in clipping.



If your deployment target is iOS 11 or later and you want to use PDF vector images that are scaled smoothly without blurring at runtime make sure you select “Preserve Vector Data” in the asset catalog (you only see the difference when running on a device).



Positioning the View

If you don't want to scale or stretch the view, you can pin it to one of nine possible positions.

- `.center`
- `.top, .bottom, .left, .right`
- `.topLeft, .topRight, .bottomLeft, .bottomRight`

Note that the image view is still filling the space between the margins. The image view positions the image within its bounds based on the mode:



Redrawing the View

The `.redraw` mode triggers the `setNeedsDisplay()` method on the view when the bounds change allowing the view to redraw itself. You probably want this mode if you have a custom `UIView` that implements `draw(_ :)` to draw its content within the view bounds.

Content Hugging And Compression Resistance

If you use Interface Builder to create your constraints it will at some point offer you this helpful suggestion:



The Content Priority Ambiguity problem that Interface Builder is warning you about can also be hiding in your layout code.

Stretching And Squeezing

Views like labels and image views have a natural (intrinsic) content size that they want to be. Sometimes there's not enough space and the layout engine has to squeeze a view smaller than its natural size to fit. Other times the layout engine has to stretch a view beyond its natural size to fill a space.

When there are several views in a layout how does the layout engine decide which view to stretch or squeeze? This is where content-hugging and compression-resistance priorities come into play.

Don't Squeeze Me - Compression-Resistance

All views have both a horizontal and vertical Compression-Resistance priority. These tell the layout engine how strongly a view resists being squeezed below its natural size in each dimension. The higher the Compression-Resistance priority, the more a view resists squeezing.

When the layout engine needs to squeeze a view to fit in a space, it chooses the view with the lowest Compression-Resistance priority. If there's not one view and only one view with the lowest priority the layout is ambiguous.

Don't Stretch Me - Content-Hugging

All views have both a horizontal and vertical Content-Hugging priority. These tell the layout engine how strongly a view resists stretching beyond its natural size in each dimension. The higher the Content-Hugging priority, the more a view resists stretching.

When the layout engine needs to stretch a view to fill a space it chooses the one with the lowest Content-Hugging priority. If there's not one view and only one view with the lowest priority the layout is ambiguous.



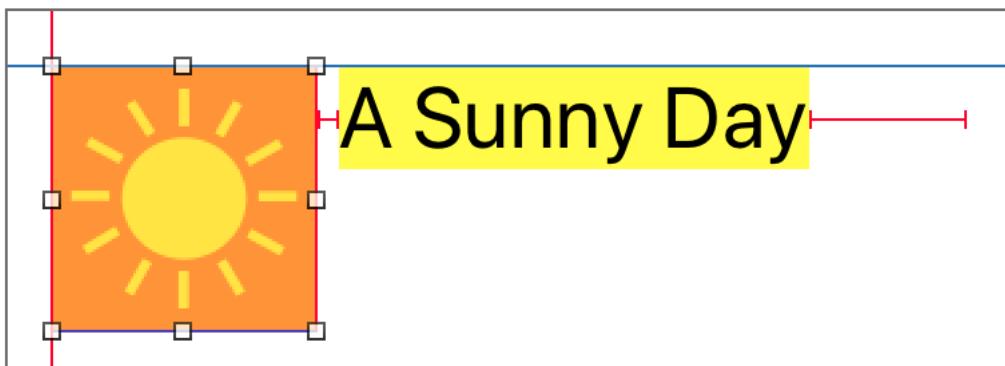
Changing the Content-Hugging or Compression-Resistance priority doesn't affect views with no intrinsic content size.

A Practical Example Using Interface Builder

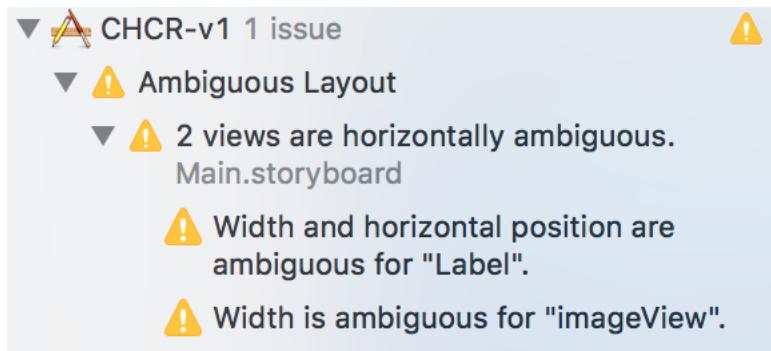
Let's look at some situations where you need to change either the Content-Hugging or Compression-Resistance priorities (see sample code: [CHCR-v1](#)).

Stretching A View

Look at this setup with an image view and a label arranged horizontally. I pinned the image view to the leading margin of the superview and the label to the trailing margin. I pinned both to the top margin and added a standard amount of horizontal spacing between the views.



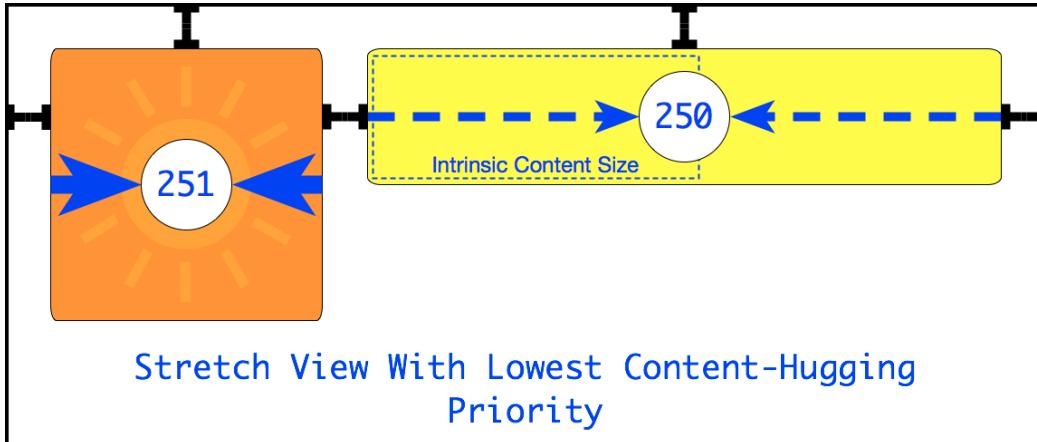
I've shown the views at their natural intrinsic content sizes. The widths of the two views are not enough to fill the space between the margins. Interface Builder warns you of an ambiguous layout:



The layout engine wants to stretch one of the views to fill the space but which one? Well, that depends on the Content-Hugging priorities. If you create this layout with Interface Builder both the image view and the label have a default horizontal Content-Hugging priority of 251.

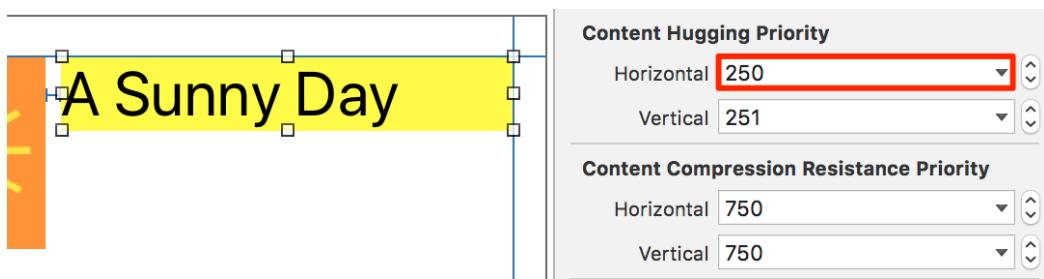
(This is an example of Interface Builder trying to be helpful. If you create the layout in code the priority of both views is 250. Either way, the layout is ambiguous.)

I want to stretch the text label, so I need to make sure it has the lowest Content-Hugging priority. For example, I can keep the image view priority at 251 and lower the label priority to 250:

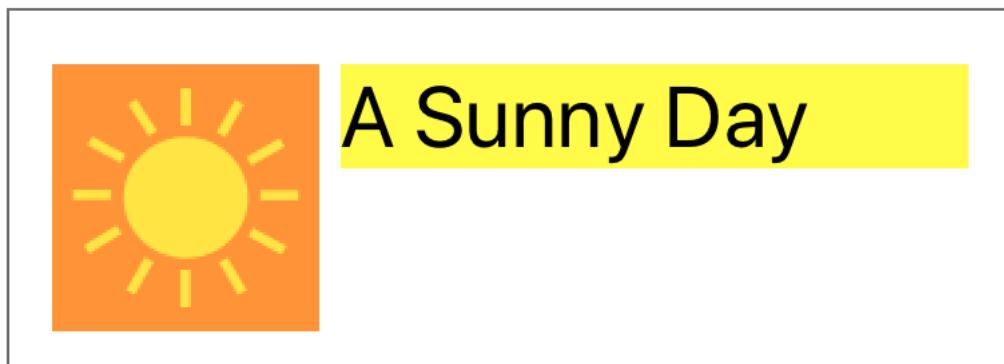


Notice that the absolute value of a priority is often not so significant. What counts is the value relative to the other views involved in the layout.

Use the Interface Builder size inspector to change the horizontal Content-Hugging priority of the label to 250:

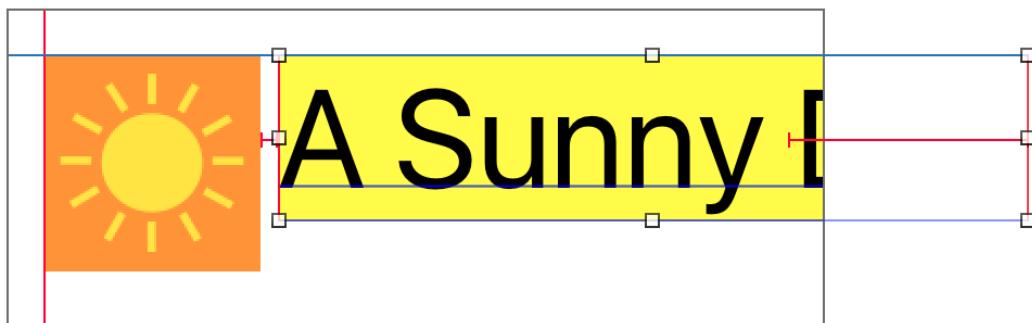


Once we remove the ambiguity the layout engine stretches the label to fill the available space keeping the image view at its natural size:

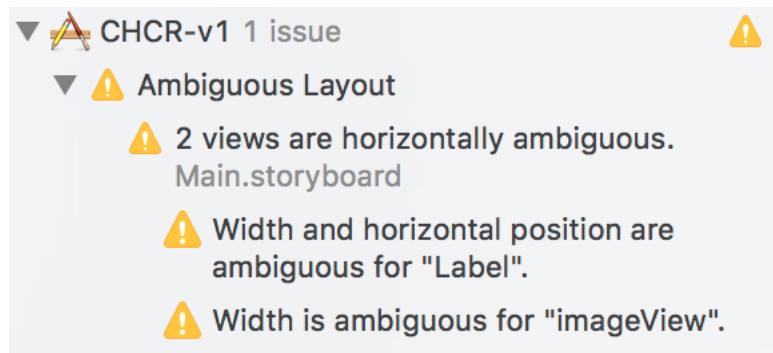


Squeezing A View

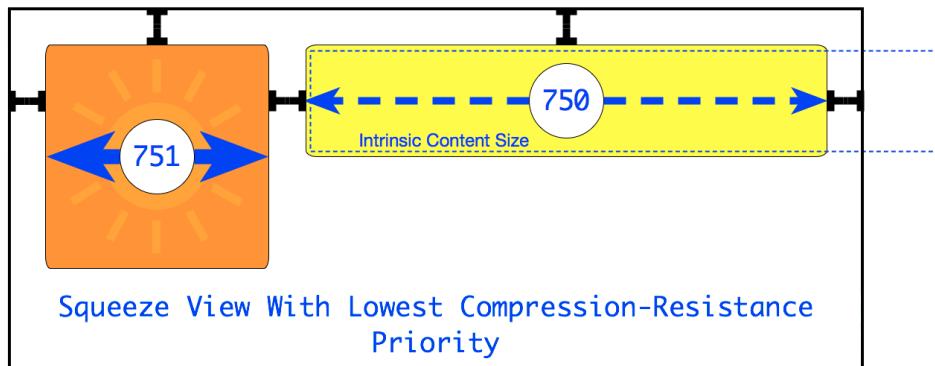
What would happen if our layout was too big to fit in the available space? This can happen easily enough when localizing or working with [Dynamic Type](#). Here's what happens to our layout when I double the font size of the label:



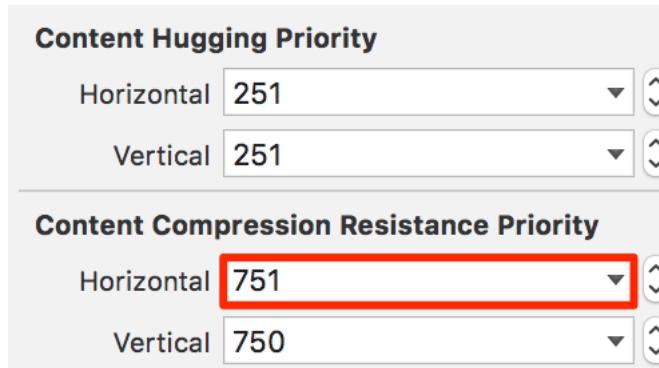
What you see may be different, but in my case, the text label has grown beyond the bounds of the superview. Our layout no longer fits within the margins and is ambiguous:



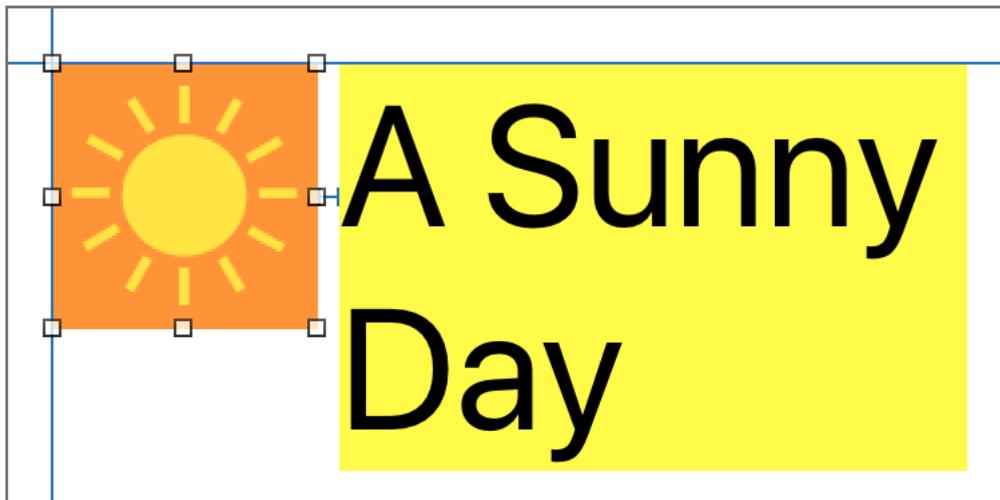
To satisfy the constraints the layout engine looks for the view with the lowest Compression-Resistance priority to squeeze. Unfortunately, both views have a horizontal Compression-Resistance value of .defaultHigh (750). Raising the priority for the image view to 751 fixes the problem:



Use the Interface Builder size inspector to change the horizontal Compression-Resistance priority of the image view to 751:



Our layout should now fit with the image view staying at its natural size and both views fitting between the margins. I set the `numberOfLines` property of the label to 0 so it can flow over more than one line if needed.



Summarizing the horizontal priorities we set for the two views:

View	Content Hugging	Compression Resistance
Image View	251	751
Text Label	250	750

- The label has the lowest Content-Hugging priority to make it stretch when our layout is too small to fit in the available space.
- The label has the lowest Compression-Resistance priority to squeeze it when our layout is too big to fit in the available space.

Here's how the layout looks in portrait and landscape:



In portrait, the label is squeezed to fit and flows over multiple lines. In landscape, the label is stretched to fill the extra space. In both cases, the image view maintains its natural content size.

Defaults When Creating Views

Most views have a `.defaultLow(250)` content hugging priority when created and a compression resistance of `.defaultHigh(750)`. I summarized

the defaults for some common views and controls below. These defaults apply to both horizontal and vertical priorities.

Views	Content Hugging	Compression Resistance
UIView UIButton UITextField UITextView UISlider UISegmentedControl	250	750
UILabel UIImageView UISwitch UIStepper UIDatePicker UIPageControl	250 (Code) 251 (IB)	750
	750	750

Notes:

1. Controls like the switch, stepper, data picker, and page control should always display at their natural size so have `.defaultHigh` (750) priorities.
2. If you create a `UILabel` or `UIImageView` with Interface Builder, they have a Content-Hugging priority of 251. If you create them in code, you get the default value of 250. Interface Builder assumes that most of the time you want labels and images to stay at their natural content size.

Working With Priorities In Code

Let's repeat our last example of stretching and squeezing views in code (see sample code: [CHCR-v2](#)). I'll skip the details of the view setup and constraints and jump to the creation of the image view:

```
private let sunImage: UIImageView = {
    let view = UIImageView(image: UIImage(named: "Sun"))
    view.translatesAutoresizingMaskIntoConstraints = false
    view.setContentHuggingPriority(.defaultLow + 1, for: .horizontal)
    view.setContentCompressionResistancePriority(.defaultHigh +
        1, for: .horizontal)
    view.backgroundColor = .orange
    return view
}()
```

We increase the horizontal Content-Hugging and Compression-Resistance priorities for the image view to one more than their default values. Remember that views created in code have a default Content-Hugging priority of `.defaultLow` (250) and Compression-Resistance priority of `.defaultHigh` (750). So it's the label that's stretched or squeezed to fit the available space between the margins.

Key Points To Remember

When we first looked at constraints we had a simple rule of thumb for answering the question [How Many Constraints Do I Need?](#).

To fix the size and position of each view in a layout we needed *at least* two horizontal and two vertical constraints for every view.

Note the *at least* as the two constraints per view “rule” only works with required, equality constraints. Once we introduce optional and inequality constraints it gets more complicated.

- All constraints have a layout priority from 1 to 1000. By default, constraints are `.required` which corresponds to a value of 1000.
- Constraints with a priority less than `.required` are optional. The layout engine tries to satisfy higher priority constraints first but always tries to get as close as possible for optional constraints.
- Combine optional constraints with required inequality constraints to pull a view as close as possible towards a size or position without violating other constraints.
- Once you have added a constraint to a view you cannot change its priority from required to optional or vice versa.

We can also relax our rule of thumb when working with views that have an intrinsic content size. This includes many of the standard UIKit controls like labels, switches, and buttons.

- The intrinsic content size of a view is the natural size a view wants to be to fit its content.
- Under the covers UIKit adds width and height constraints for us that set the intrinsic size of the view.
- You can always override the intrinsic size by adding constraints.

Views with an intrinsic content size will not always fit perfectly in the available size of their superview. When Auto Layout has to stretch or squeeze views to make them fit it takes into account the relative content priorities of the views:

- Views have both horizontal and vertical *Compression-Resistance* priorities that tell Auto Layout how much a view resists being *squeezed* in that dimension. Auto Layout squeezes the view with the lowest Compression-Resistance priority first.
- Views have both horizontal and vertical *Content-Hugging* priorities that tell Auto Layout how much a view resists being *stretched* in that dimension. Auto Layout stretches the view with the lowest Content-Hugging priority first.
- Changing the content priority of a view that doesn't have an intrinsic content size has no effect.

Not sure when to change content priorities? Ask yourself these two questions:

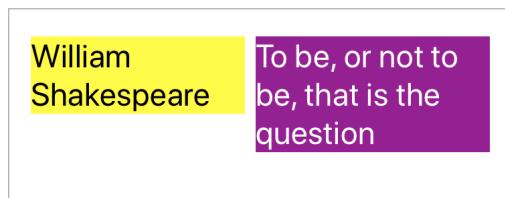
- Can my layout ever be too big to fit in the available space? If so decide which view to squeeze first and give it the lowest Compression-Resistance priority.
- Can my layout ever be too small to fit in the available space? If so decide which view to stretch first and give it the lowest Content-Hugging priority.

Test Your Knowledge

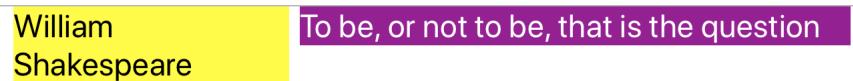
Practice using optional constraints when you need to pull a view as close as possible to a size or location. Learn to spot when you're working with views that have an intrinsic content size and need to change the content priority to stretch or squeeze a view.

Challenge 7.1 Twice As Big If Possible

Two multi-line labels arranged horizontally show an author name and quotation. Both labels are using the 24 pt system font. I want the left author label fixed to the leading and top margins and the right quotation label fixed to the trailing and top margins. There's a standard amount of horizontal spacing between the labels.



The author label must be at least 160 points wide (but it can be wider). The widths of the two labels should fill the available space with the quotation label being twice the width of the author label if possible. If there's insufficient space, it should be as close as possible to twice the width.



1. Build this layout using Auto Layout. You can choose to use a storyboard or create the layout programmatically (or do it both ways!). If you use a storyboard, you should not need to write any code.
2. Run your layout on different devices in both portrait and landscape and check that the author label is always at least 160 points wide. When there's space, the quotation label should be twice the size of the author label with both labels growing to fill the available space.

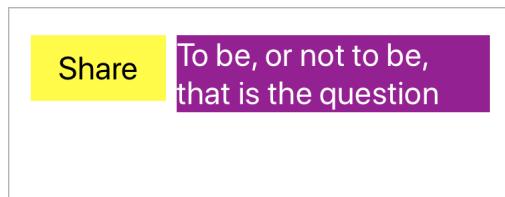
Hints And Tips

1. When creating the labels set the number of lines to zero so that the text wraps across multiple lines.
2. What type of constraint should you be thinking about when you see "at least 160 points wide"?
3. What type of constraint should you be thinking about when you see "if possible"?
4. The width constraint for the author label needs a greater than or equal relation. It's a required constraint ("must be").
5. The width constraint between the two labels is an optional ("if possible") constraint. We want it satisfied but only if there's sufficient width. Otherwise, the layout engine should get as close as it can.
6. The priority of the optional width constraint needs to be less than 1000. The exact value is not important, using 750 is common.

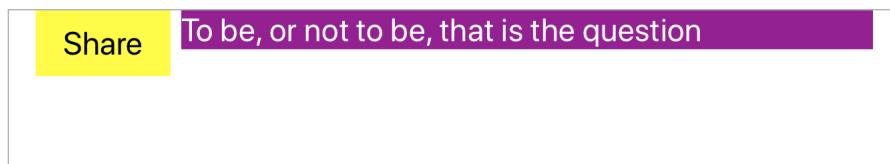
7. You don't need to change content hugging or compression resistance priorities.

Challenge 7.2 Stretch Or Squeeze?

This time I have a share button and a quotation label arranged horizontally. As in the last challenge, the text is 24 pt system font and a standard amount of spacing separates the button from the label. I fixed the items to the top and side margins:



The button should stay at its natural size and the label resize to fill the available width. The text can flow over multiple lines if required. Here's how it looks in landscape:



1. Build this layout using Auto Layout. You can again choose to use a storyboard or create the layout programmatically. If you use a storyboard, you should not need to write any layout code.
2. Run your layout on multiple devices in both portrait and landscape to check that only the quotation label changes size to fit the available space.

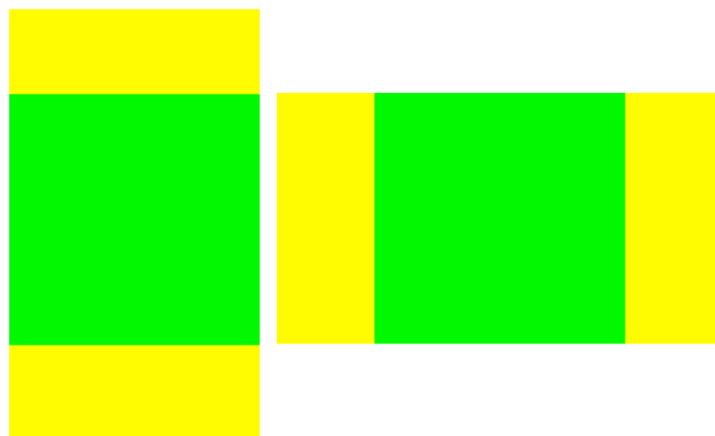
Hints And Tips

1. I added a small amount of padding to the button by setting the left and right content insets to 20 and top and bottom to 10.
2. Set the number of lines to 0 for the label, so the text flows over multiple lines.
3. In portrait, there's not enough width for the label to fit with a single line of text. Squeeze the label by making its horizontal compression resistance priority lower than the button priority.

4. In landscape, the label must stretch beyond its natural size to fill the space, so its content hugging priority needs to be the lower than the button priority.
5. The label has a default content hugging priority of 251 in Interface Builder and 250 if created in code. The button always has a default content hugging priority of 250. The button and label have default compression resistance priorities of 750.

Challenge 7.3 A Big As Possible Square

Suppose I have a green view that must be square and in the center of the screen. I want it to be as big as possible while staying fully on-screen. For an iPhone, in portrait, the square is as wide as the screen. In landscape, it's as tall as the height of the screen:



1. Build this layout using Interface Builder or in code.
2. Test on a variety of devices from the smallest iPhone SE up to the larger iPads. Verify that the green view remains square, centered in the view and is as large as possible while fitting entirely on-screen.

Hints And Tips

1. You need to use a combination of inequality and optional constraints for this layout.
2. Start by describing the position of the green view, make the green view square, then work on the width and height.
3. Remember when something about our layout must be true it's a required constraint. When we want something as close as possible, it's an optional constraint.
4. Make the green view square by giving it equal width and height.

5. Imagine the square expanding from the center. The width must never be greater than the width of the root view. The height must never be greater than the height of the root view.
6. Inequality constraints are not enough to fix the height or width of the green view. You need to add a constraint that pulls either the width or height **as close as possible** to the width or height of the root view without violating the other constraints.
7. You can solve this with 5 required constraints (2 of which are inequalities) and 1 optional constraint.

Chapter 8

Stack Views

Apple introduced the stack view back in iOS 9 and made a bold claim for it during a session on Auto Layout at WWDC 2015:

Start with Stack View, use constraints as needed

—WWDC 2015 Session 218 Mysteries of Auto Layout, Part 1

I would not go that far. Stack views are not without problems. To use them fully you still need some understanding of constraints and layout priorities, but they can reduce the amount of boilerplate code for some common layouts.

At first glance, the stack view is a class that allows you to layout views in either a vertical column or a horizontal row. What makes it convenient is that it takes care of adding the constraints for the views it manages.

By embedding stack views in other stack views, you can build complex layouts without a lot of painful Auto Layout code. Most of the time you only need to add constraints to fix the size and position of the stack view itself.



A stack view doesn't automatically scroll its contents like a table or collection view. See the later chapter on scroll views if you need [A Scrolling Stack View](#).

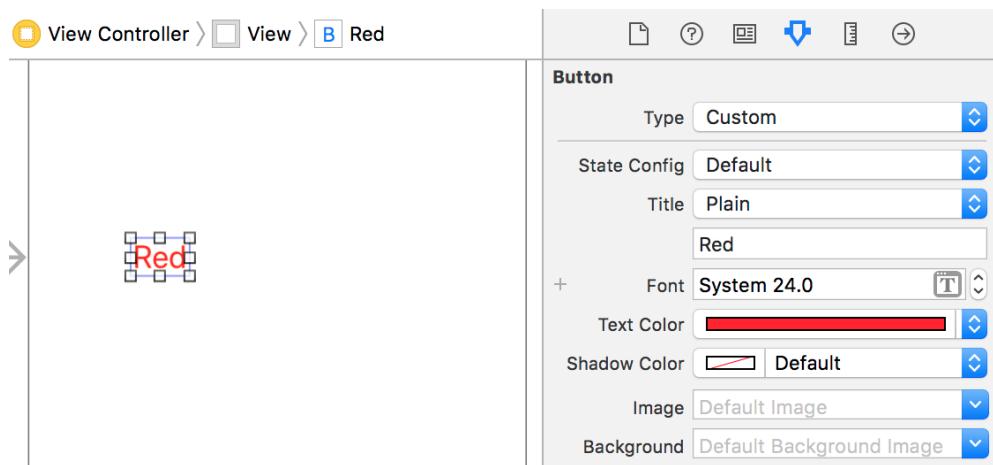
Getting Started With Stack Views

Before we dive too deep into the details, let's take a quick tour of what you can do with a stack view.

Creating A Stack View With Interface Builder

Let's start with an example using Interface Builder (see sample code: [StackViews-v1](#)):

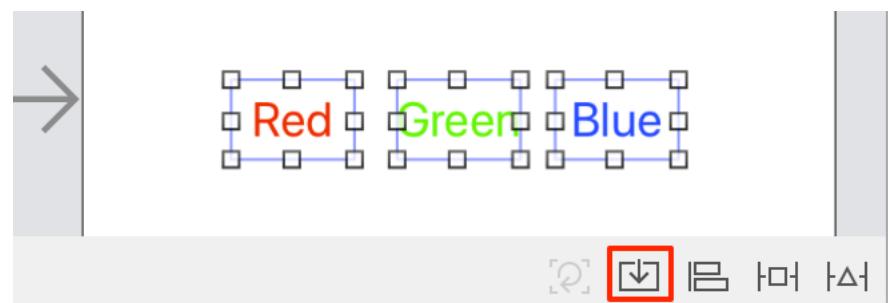
1. Create a new Xcode project using the Single View App iOS template and drag a button from the object library into the view canvas. Using the attributes inspector set the title, font size, and color. I'm using a 24 point red font:



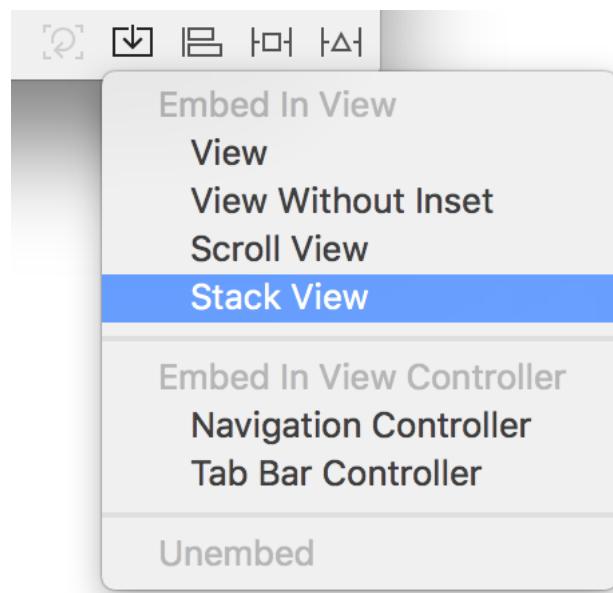
2. Add two more buttons and change the titles and color to create green and blue buttons. Position the buttons in a roughly horizontal row (don't worry about being exact):

Red Green Blue

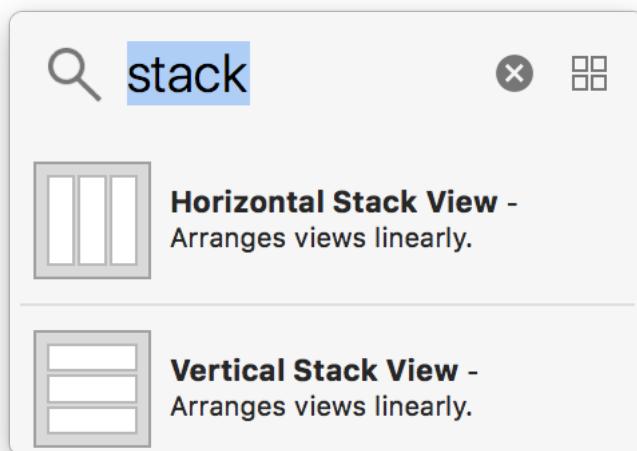
3. Now select all three buttons and use the **[Embed In]** tool in the toolbar at the bottom of the Interface Builder window. (Or use the menu **Editor > Embed in > Stack View**).



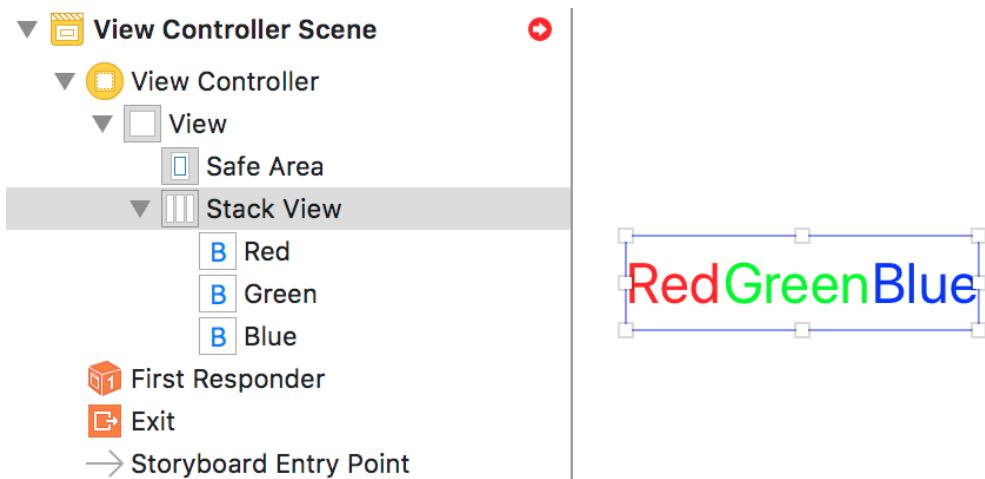
Select Stack View from the menu:



I prefer using the embed tool, but you can also drag a stack view from the Xcode object library onto the view canvas and drop views onto it to embed them:

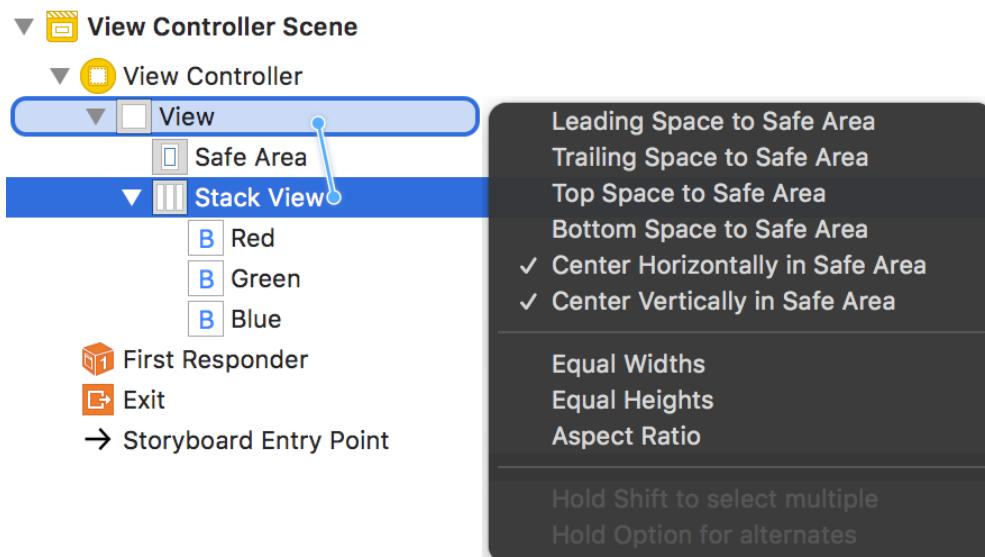


4. Interface Builder creates a horizontal stack view for us and adds the three buttons to the stack view.



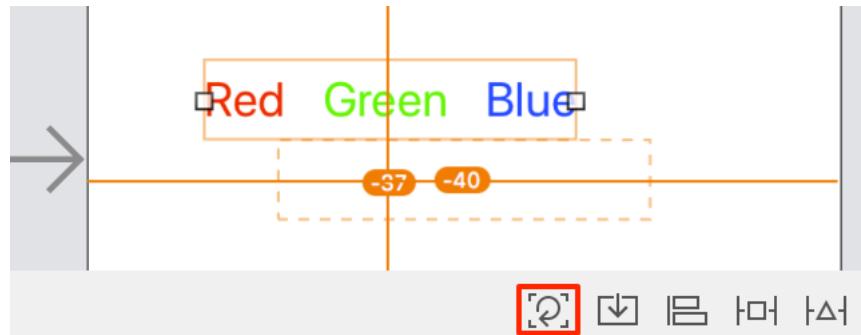
It does its best to guess whether we want a horizontal or vertical stack view based on the positions of the added views. You can always change the axis using the attributes inspector if Interface Builder gets it wrong.

5. Fix the position of the stack view by adding constraints to center it in the safe area. Control-drag from the stack view to the root view in the document outline. Hold down the Shift key and select the center horizontally and vertically constraints:



If the stack view was not already in the center, you might see a

warning that the frame will be different at runtime. Use the **[Update Frames]** tool in the toolbar so that Interface Builder updates the position of the stack view:



6. If you now select the stack view, you can experiment with changing the configuration in the attributes inspector. Try changing the axis from horizontal to vertical and back again. You might also want to add some spacing between the buttons or even try changing the distribution.



I added 16 points of spacing and changed the distribution to “Fill Equally” which gives the buttons equal width.

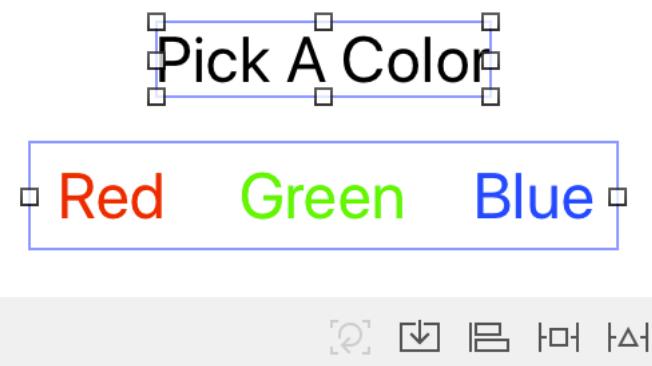
Embedding Stack Views In Stack Views

The real power of stack views comes when you realize that you can embed stack views inside other stack views. You can build up complex layouts without having to manage the constraints for all the subviews. Let’s extend our last layout with a label that we want to position vertically above our row of buttons (see sample code: [StackViews-v2](#)):

1. Drag a label onto the canvas and position it roughly above the row of buttons. Don’t put it too near or you’ll add it to the stack view. Change the label text and increase the font size to 24 points:



2. Let's create a vertical stack view that contains two subviews - our new label and the horizontal stack view. Select both the label and the stack view and use the **[Embed In]** tool at the bottom of the canvas to add them to a new stack view:

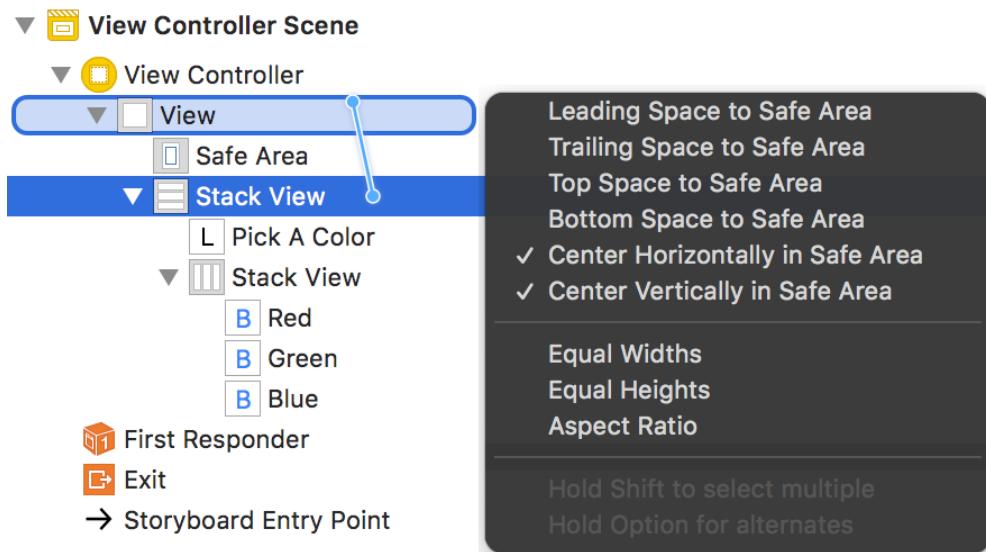


3. This time you should get a vertical stack view:



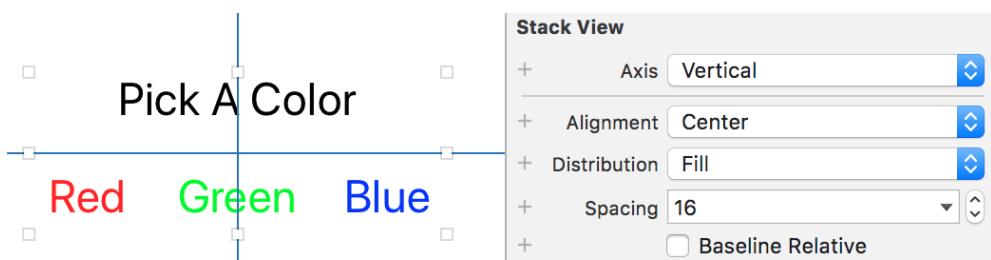
What else happened? We lost the center constraints we added for the horizontal stack view. A stack view manages the constraints for its views, so Interface Builder removes constraints for views you add to a stack view.

4. As before, control-drag from the new stack view to the root view and add constraints to center horizontally and vertically:



Use the **[Update Frames]** tool if the stack view is not in the right position after adding the constraints:

5. Make any changes you want to the configuration of the vertical stack view in the attributes inspector (make sure you have the vertical stack view selected):



The vertical stack view has an alignment that centers the subviews. Try changing it to leading or trailing and see what happens. As before you may want to add some spacing between the label and the horizontal stack view.

Creating A Stack View In Code

Using a stack view with programmatic layouts reduces the amount of code you need to write. Let's repeat the last example, but instead of adding the stack views in Interface Builder let's create everything in code in the view controller (see sample code: [StackViews-v3](#)):

1. Start a new Xcode project without a storyboard (see [Xcode Project](#)

(And File Templates). The setup of the view controller follows our usual template. I added a private enum for the view metric constants:

```
// RootViewController.swift
import UIKit

class RootViewController: UIViewController {

    private enum ViewMetrics {
        static let fontSize: CGFloat = 24.0
        static let spacing: CGFloat = 16.0
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        setupView()
    }

    private func setupView() {
        // Setup view and constraints
    }
}
```

2. To avoid repeating ourselves let's create an extension to UIButton to create our custom buttons:

```
private extension UIButton {
    static func customButton(title: String, color: UIColor,
        fontSize: CGFloat) -> UIButton {
        let button = UIButton(type: .custom)
        button.setTitle(title, for: .normal)
        button.setTitleColor(color, for: .normal)
        button.titleLabel?.font = UIFont.systemFont(ofSize:
            fontSize)
        return button
    }
}
```

3. Add three properties to our view controller for the buttons:

```
private let redButton = UIButton.customButton(title:
    "Red", color: .red, fontSize: ViewMetrics.fontSize)
private let greenButton = UIButton.customButton(title:
    "Green", color: .green, fontSize: ViewMetrics.fontSize)
private let blueButton = UIButton.customButton(title:
    "Blue", color: .blue, fontSize: ViewMetrics.fontSize)
```

4. Create another property for the button stack view. This time make it a lazy property so we can access the button properties after the view controller is initialized.

```
private lazy var buttonStackView: UIStackView = {
    let stackView = UIStackView(arrangedSubviews:
        [redButton, greenButton, blueButton])
    stackView.spacing = ViewMetrics.spacing
    stackView.distribution = .fillEqually
    return stackView
}()
```

The stack view defaults to the `.horizontal` axis, so we only need to set the spacing and distribution.

5. Before we create the vertical stack view we first need to create the label:

```
private let colorLabel: UILabel = {
    let label = UILabel()
    label.text = "Pick a color"
    label.font = UIFont.systemFont(ofSize:
        ViewMetrics.fontSize)
    return label
}()
```

6. Create a second stack view that contains the label and our first button stack view:

```
private lazy var rootStackView: UIStackView = {
    let stackView = UIStackView(arrangedSubviews:
        [colorLabel, buttonStackView])
    stackView.translatesAutoresizingMaskIntoConstraints = false
    stackView.axis = .vertical
    stackView.alignment = .center
    stackView.spacing = ViewMetrics.spacing
    return stackView
}()
```

This time we do need to set the axis of the stack view to vertical. Add some spacing and change the alignment to `.center`.

You may have noticed that I only disabled the autoresizing mask for the root stack view and not for the buttons and label. A stack view disables the autoresizing mask translation for the views it manages. So the button stack view takes care of the buttons, and the root

stack view manages the button stack view.

7. Finally, with all of the views created add the top stack view to the root view:

```
private func setupView() {  
    view.addSubview(rootStackView)
```

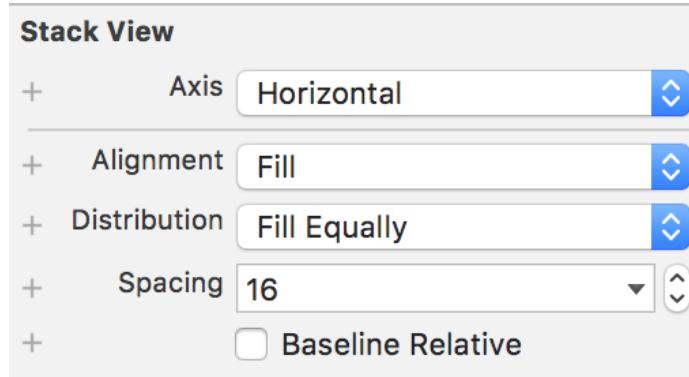
8. To center the stack view, add constraints between the center X and Y anchors of the stack view and the safe area layout guide.

```
NSLayoutConstraint.activate([  
    rootStackView.centerXAnchor.constraint(equalTo:  
        view.safeAreaLayoutGuide.centerXAnchor),  
    rootStackView.centerYAnchor.constraint(equalTo:  
        view.safeAreaLayoutGuide.centerYAnchor)  
])
```

9. Those are the only two constraints that we need to add. The stack views do the rest.

A Closer Look At Stack Views

When you create a stack view, you decide if you want a horizontal or vertical axis and add some views for the stack view to arrange. You fine-tune the layout by changing the default distribution, alignment, spacing and margins used by the stack view.



The configuration options are (mostly) available both in the attributes inspector in Interface Builder or programmatically. Let's look at each of them.

Stack View Arranged Views

A stack view manages the layout of views in its `arrangedSubviews` property. This can be confusing at first as a stack view also has a `subviews` property inherited from `UIView`.

Adding views to a stack view

When you drag and drop a view onto a stack view in Interface Builder or use the *Embed In* tool, you're adding views to both `arrangedSubviews` and `subviews`. When using stack views in code, you can add arranged views when you create the stack view or add (append to the end) or insert views later:

```
let stackView = UIStackView(arrangedSubviews: [view1, view2,
    view3])
stackView.addArrangedSubview(view4)
stackView.insertArrangedSubview(view5, at: 0)
```

These methods add views to both `arrangedSubviews` and `subviews`. The first view in `arrangedSubviews` is the leftmost view for a horizontal stack and at the top for a vertical stack.

The order of the views in `subviews` sets which views appear in front should they overlap (from back to front). The order of `subviews` doesn't change if the view is already a member when you add it to the stack view.

Don't use `addSubview` when you want `addArrangedSubview`. Adding the view to `subviews` making it visible as part of the view hierarchy, but the layout is not managed by the stack view so it will be missing constraints:

```
// Add to subviews
stackView.addSubview(redButton)

// Add to arrangedSubviews and subviews
stackView.addArrangedSubview(redButton)
```

Removing views from a stack view

You can use `removeArrangedSubview(_:)` to remove a view from a stack view but be careful. This removes the view from `arrangedSubviews`, and the stack view no longer manages the layout of the view. The view is not removed from `subviews` so is still visible unless you hide it or remove it with `removeFromSuperview()`.

```
// remove from arrangedSubviews (view still visible)
stackView.removeArrangedSubview(redButton)

// remove from arrangedSubviews and subviews
redButton.removeFromSuperview()
```

To remove a view from both `arrangedSubviews` and `subviews` in one go use `removeFromSuperview`.



You may find it easier to hide a view by setting the `isHidden` property rather than removing it. The stack view only cares about visible views and doesn't add constraints for a hidden view.

Stack View Axis

A stack view has a horizontal axis by default. Change it by setting the `axis` property:

```
// Set the axis to vertical
stackView.axis = .vertical

// Change to horizontal
stackView.axis = .horizontal
```

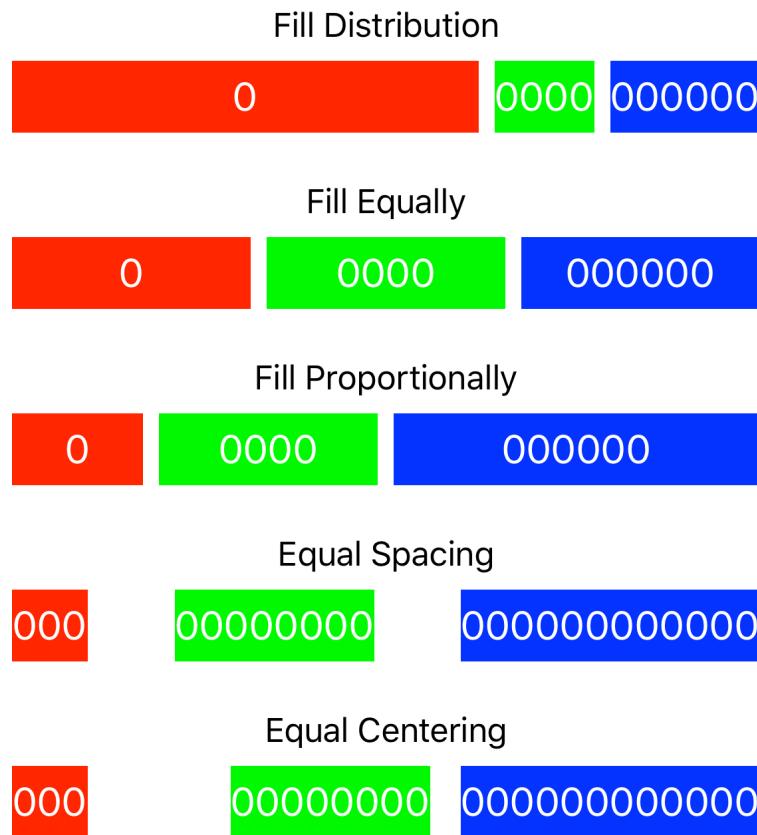
Stack View Distribution

A stack view pins the first and last views in its arranged subviews to the edges/margins of the stack view along the axis of the stack view. For a horizontal stack view, this is the leading and trailing edge/margin of the stack view. For a vertical stack view the top and bottom edge/margin.



By default, the stack view adds constraints to the edges of the stack view. To use constraints to the margins of the stack view, set the `layoutMarginsRelativeArrangement` property (see [Stack View Margins](#)).

You change the way a stack view distributes its arranged subviews to fill the space along the axis with the `distribution` property. It has five possible values:

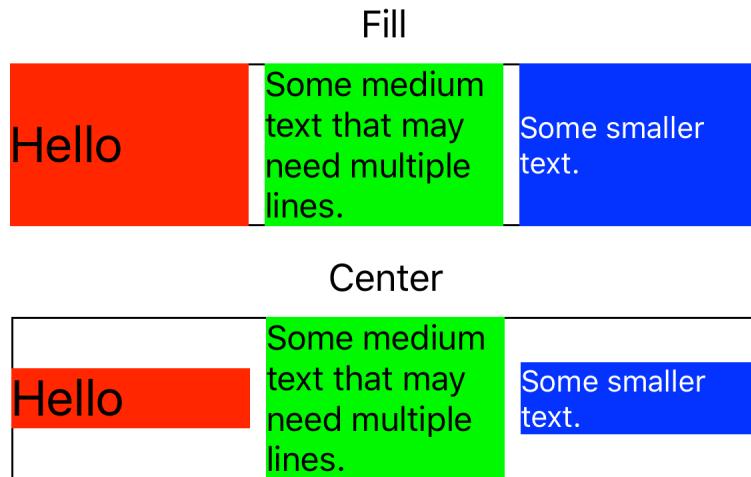


- `.fill` (the default distribution): Tries to fill the available space keeping the views at their intrinsic content size. If the space is not filled it stretches the view with the lowest content hugging priority. If the views are too big, it shrinks the view with the lowest compression resistance priority.
- `.fillEqually`: Resizes all views to the same size sufficient to fill the space together with any spacing between the views.
- `.fillProportionally`: Resizes views proportionally based on their intrinsic content size maintaining the relative size of each view.
- `.equalSpacing`: If there's sufficient space, this keeps views at their intrinsic content size and fills the space by evenly padding the spacing between views. Otherwise, it shrinks the view with the lowest compression resistance priority to maintain the minimum spacing of the `spacing` property.
- `.equalCentering`: Pads the spacing between the views to try and create equal distance between the centers of each view. If necessary, the view with the lowest content compression priority shrinks to maintain the minimum spacing of the `spacing` property.

Stack View Alignment

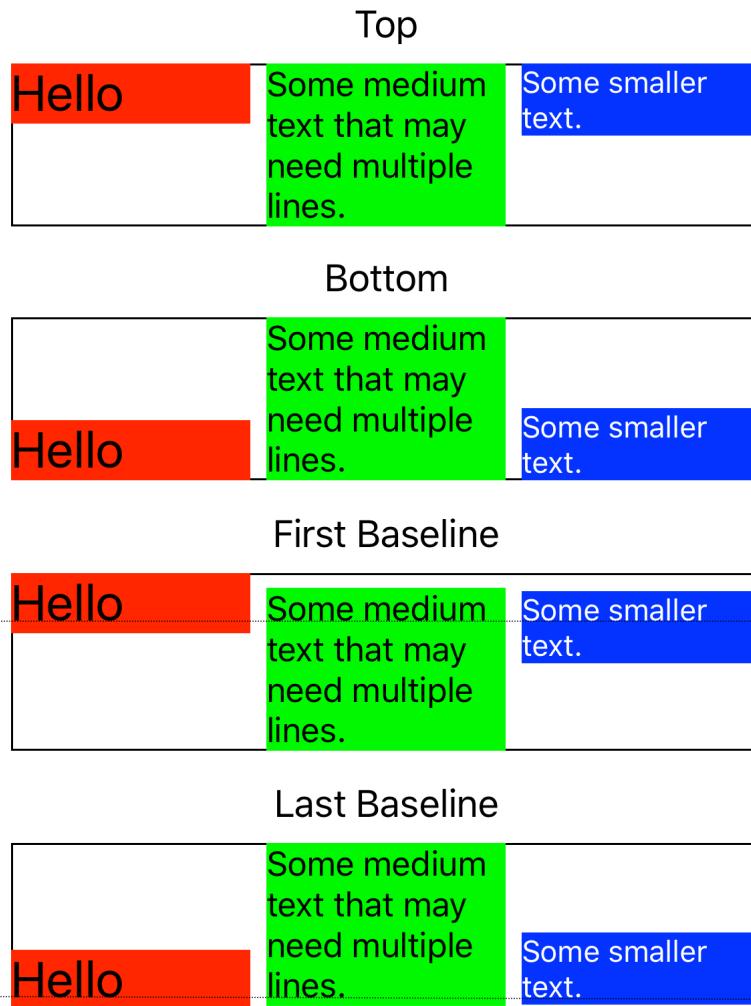
The `.alignment` property changes the way views align perpendicular to the stack view axis. There are eight possible values, but not every value applies to both horizontal and vertical stack views.

The `.fill` alignment is the default and together with the `.center` alignment works for both stack view orientations:



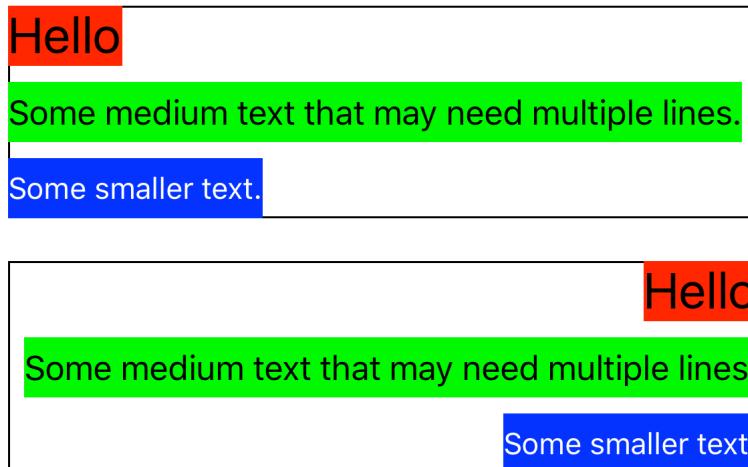
The `.fill` alignment stretches and squeezes the arranged views perpendicular to the stack view axis to fill the available space. If the stack view is not constrained perpendicular to the axis, the arranged views take their intrinsic content size, and the largest arranged view sets the stack view size in that direction,

The `.top`, `.bottom`, `.firstBaseline`, `.lastBaseline` alignments only apply to horizontal stack views:



Apple warns that the first and last baseline alignments only work for views at their intrinsic content size. If you stretch or squeeze the views the baseline can be wrong.

For vertical stack views you can use leading and trailing alignments:



Stack View Spacing

The `spacing` property sets the space between arranged subviews along the axis of the stack view. For the `fill`, `fillEqually` and `fillProportionally` distributions, this is the exact spacing to use, and the stack view stretches or squeezes views to fit.

For the `.equalSpacing` or `.equalCentering` distributions this is the minimum spacing to use. The padding between views may increase beyond this spacing, but the stack view squeezes views if needed to keep at least the minimum spacing.

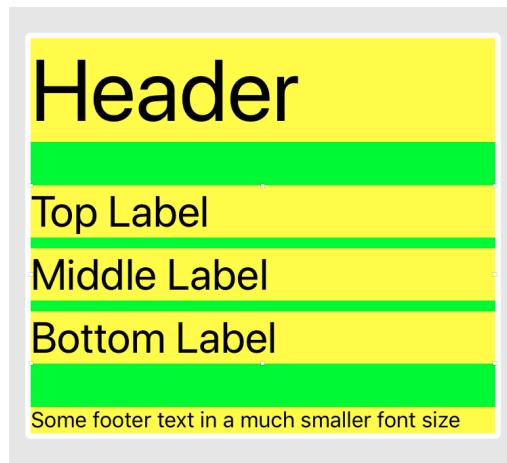
```
// Set the spacing to 16 points
stackView.spacing = 16
```



Use a negative value to overlap views. The order of the views in `subviews` controls the appearance from background to foreground of the views.

Custom Spacing (iOS 11)

The `spacing` property applies spacing evenly to the arranged subviews of the stack view. If you want the spacing between subviews to vary you need some extra steps. Take a look at this layout with five labels:



The spacing between the three central labels is 8 points, and the spacing after the header and before the footer is 32 points. Before iOS 11 you would do this either with nested stack views or by adding extra spacer views into the stack view which is a pain.

Starting with iOS 11 you can customize the spacing between views. Interface Builder doesn't support this, so you need to configure the stack view in code.

Suppose we have a stack view configured with 8 points of spacing. To get the extra spacing after the header label and before the footer label:

```
// available in iOS 11
stackView.setCustomSpacing(32.0, after: headerLabel)
stackView.setCustomSpacing(32.0, after: bottomLabel)
```



You always set the spacing after the arranged subview. There's no method to set the spacing before a view.

Standard and Default Spacing (iOS 11)

Apple also added two new properties on the `UIStackView` class in iOS 11 that define default and system spacing:

```
class let spacingUseDefault: CGFloat
class let spacingUseSystem: CGFloat
```

You don't directly use the values these properties return. You use them to set or reset the custom spacing after a view. For example to use the system defined spacing after the top label:

```
stackview.setCustomSpacing(UIStackView.spacingUseSystem,  
    after: topLabel)
```

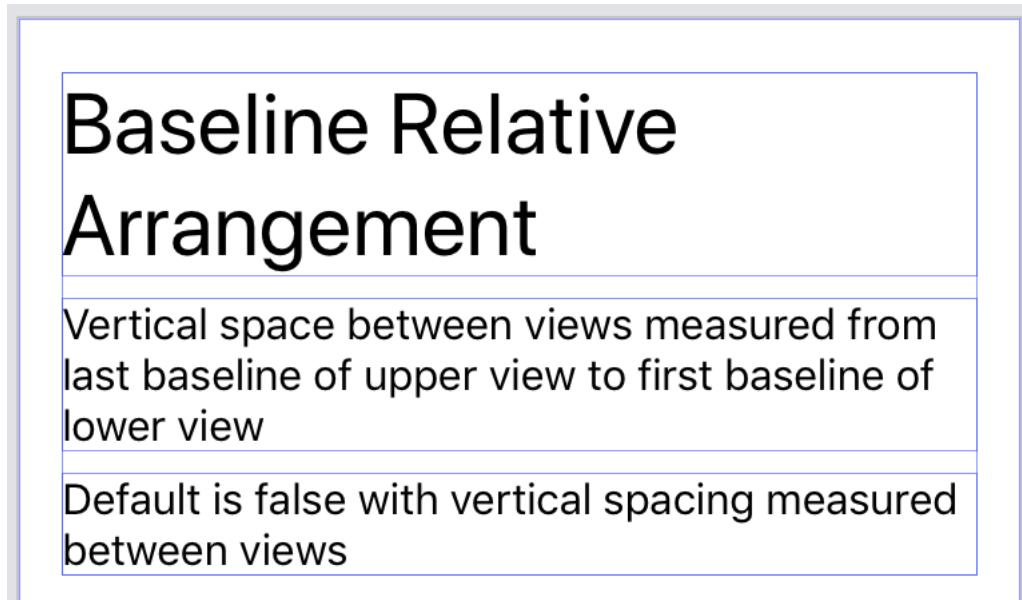
To reset the spacing after this label back to the value of the spacing property (removing the custom spacing):

```
stackview.setCustomSpacing(UIStackView.spacingUseDefault,  
    after: topLabel)
```

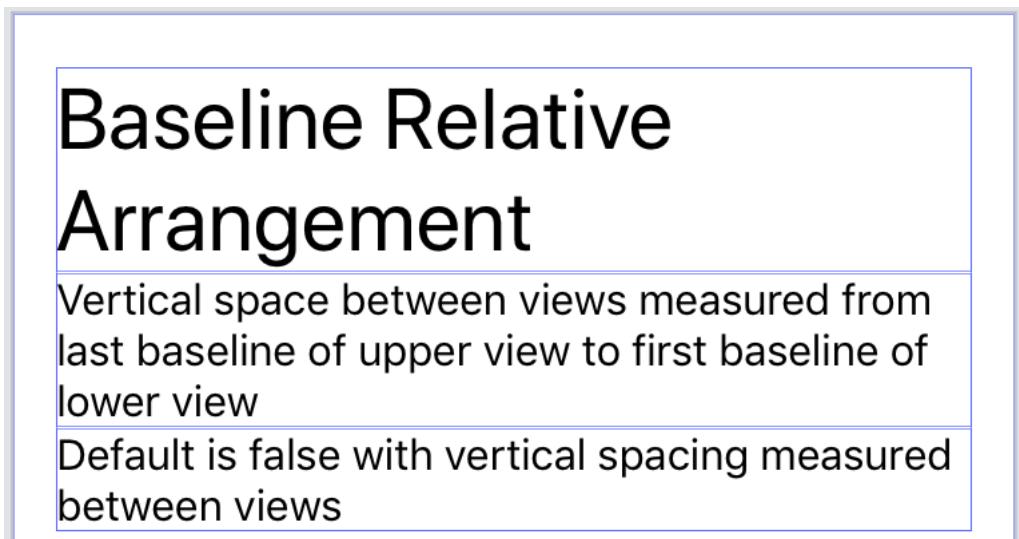
Baseline Relative Arrangement

This property only makes sense when you're working with text in a vertical stack view. Set it to `true` to space views based on the text baselines (last to first) rather than the view edges (bottom to top).

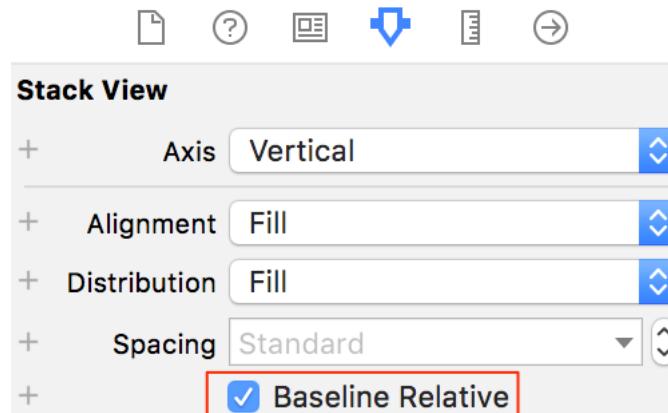
Suppose we have a vertical stack view with three text labels and a standard amount of spacing. By default this spaces the labels 8 points apart measured from their edges:



Setting the `baselineRelativeArrangement` property to `true` spaces the labels measured from the last baseline of a label to the first baseline of the label below:



If you're using Interface Builder the setting is in the attributes inspector for a stack view just under the spacing:



If you're building a stack view in code set it directly on the stack view (the default is false):

```
stackView.isBaselineRelativeArrangement = true
```

Stack View Margins

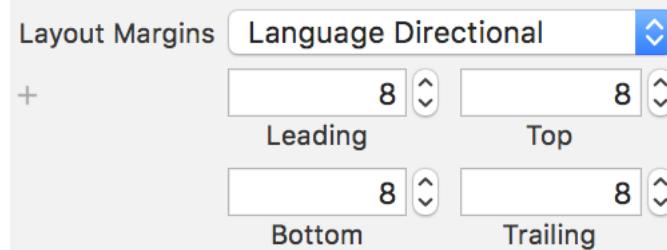
A stack view has a `layoutMargin` property the same as any other view to add spacing between the edges of the stack view and its arranged subviews. By default, a stack view doesn't use the margin. It creates a layout relative to its edges.

Set the `layoutMarginsRelativeArrangement` property to `true` to have the stack view layout relative to its margins. For example, to have 8 points

of margin between the stack view edges and its arranged subviews:

```
// 8 point margins
stackView.isLayoutMarginsRelativeArrangement = true
stackView.directionalLayoutMargins =
    NSDirectionalEdgeInsets(top: 8.0, leading: 8.0, bottom: 8.0,
                           trailing: 8.0)
```

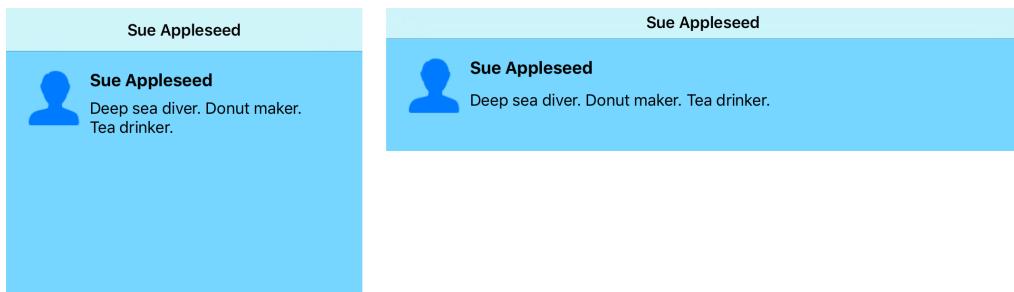
If you're using Interface Builder, there's no direct way to set this property. Instead, you add an explicit margin for the stack view:



Note that Apple added directional layout margins in iOS 11. For iOS 10 and earlier use the fixed layout margins. See [Changing The Size Of Margins](#) for more details.

Stack Views And Layout Priorities

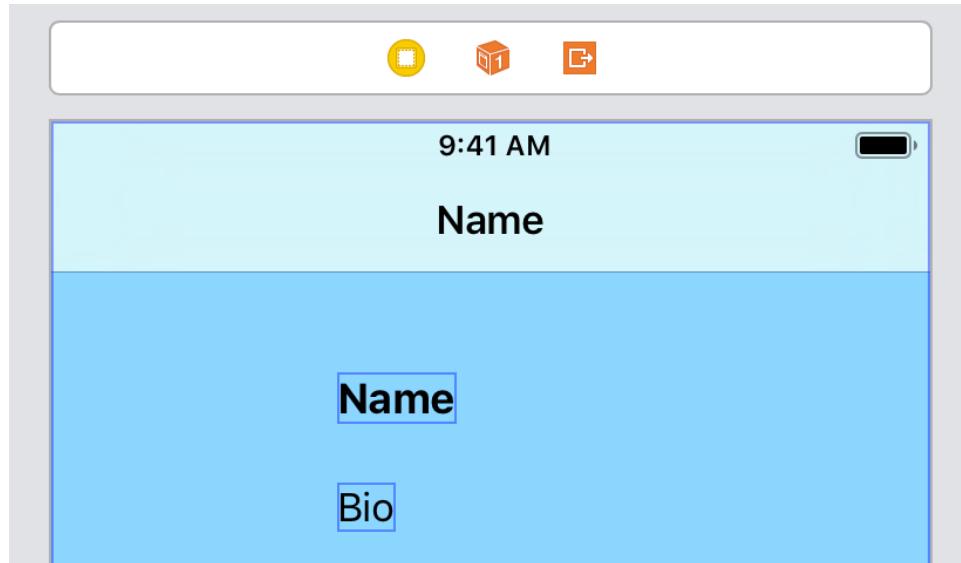
Stack views are convenient. They save you from writing a lot of boilerplate constraint code. They don't remove the need for you to understand and sometimes manage layout priorities. If you use the fill, equal spacing or equal centering distributions you may still need to change the content hugging or compression resistance priorities of the managed views to get a working layout. Here's an example of a user profile page:



This layout, shown in portrait and landscape on an iPhone 8, has an image view and two labels. The image view on the left shows the user profile picture. The two labels stacked vertically on the right show the username

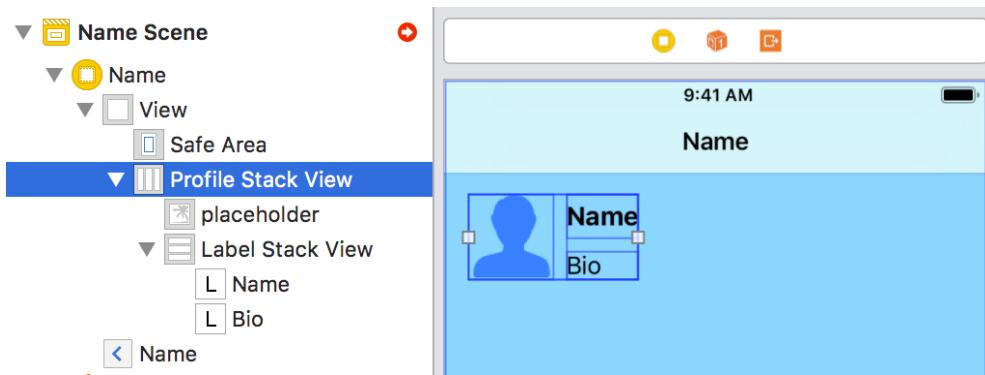
and a short bio description. Let's build this with stack views (see sample code: [StackProfile-v1](#)):

1. Starting from the Single View App iOS template. Embed the view controller in a navigation controller and add two labels to the root view of a view controller:



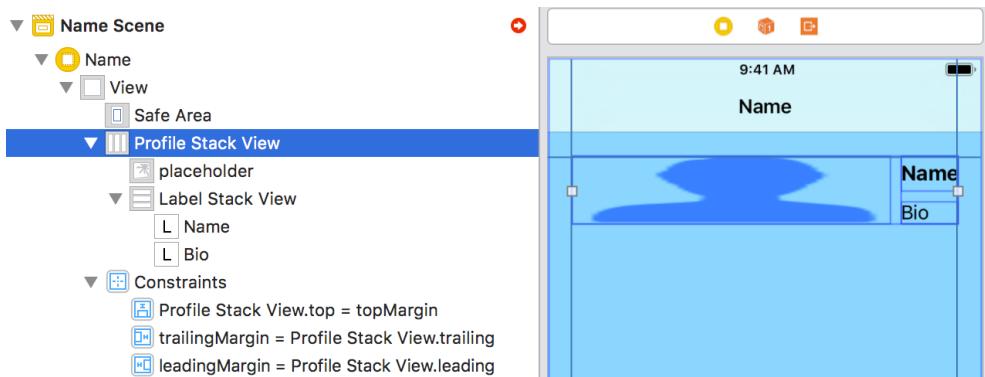
The name label uses an 18pt bold system font. The bio label uses a 17pt regular system font. Both labels have the numbers of lines set to zero to use multiple lines when needed.

2. Select both labels and embed them in a vertical stack view. Set the distribution and alignment to `.fill` and use a standard amount of spacing.
3. Drag an image view from the object library onto the canvas to the left of the labels. I have a placeholder template image in the asset catalog that is a 60x60 point vector image. Select the image view and label stack view and embed both in a horizontal stack view:

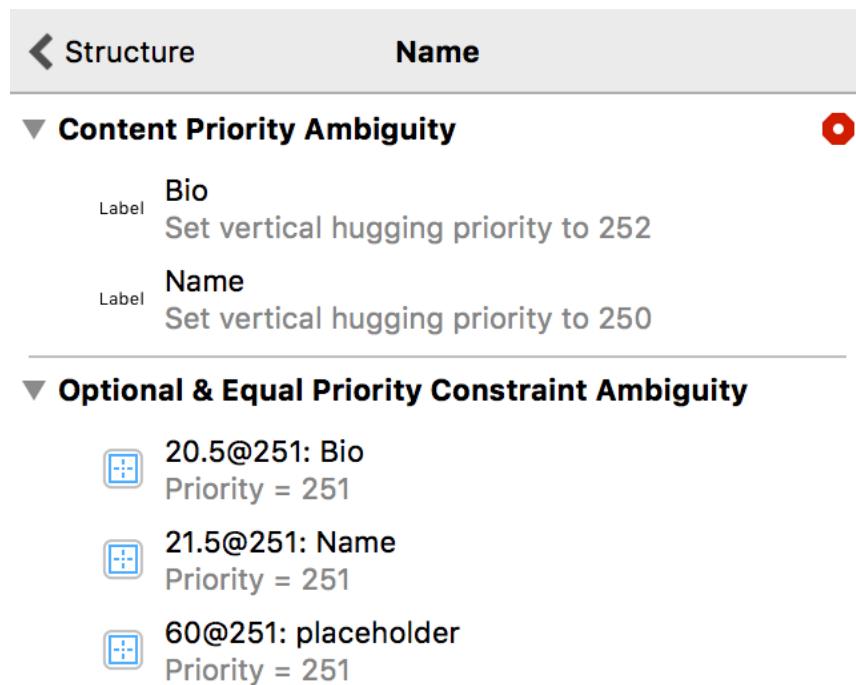


Configure the stack view to have `.fill` distribution and alignment and use a standard amount of spacing.

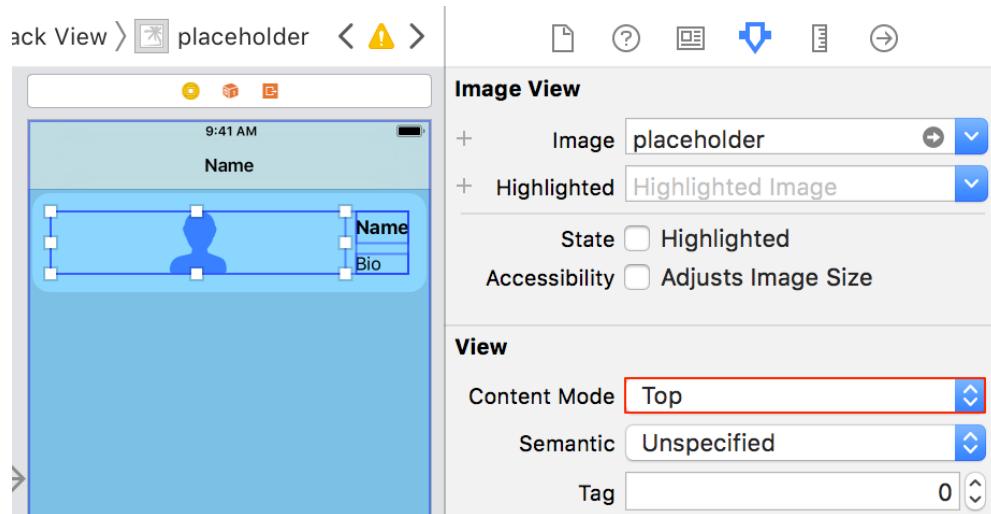
4. Add constraints to pin the top profile stack view to the leading, top and trailing margins of the superview. I also increased the margins of the root view to 20 points on all sides:



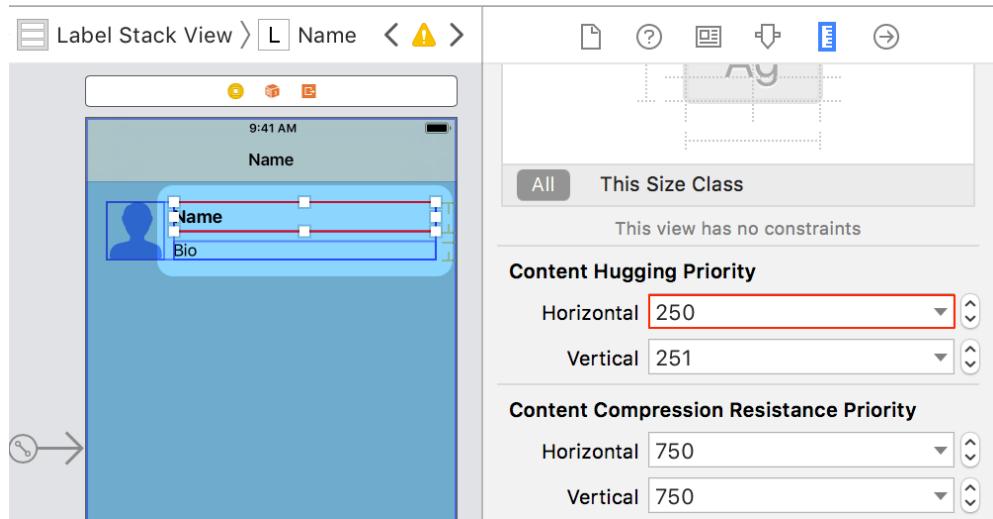
The layout is ambiguous both horizontally and vertically at this point:



5. First fix the **content mode** of the image view. This defaults to “Scale To Fill” which stretches the image. I want it to stay fixed at the top of the layout. Use the attributes inspector to change it to “Top”:

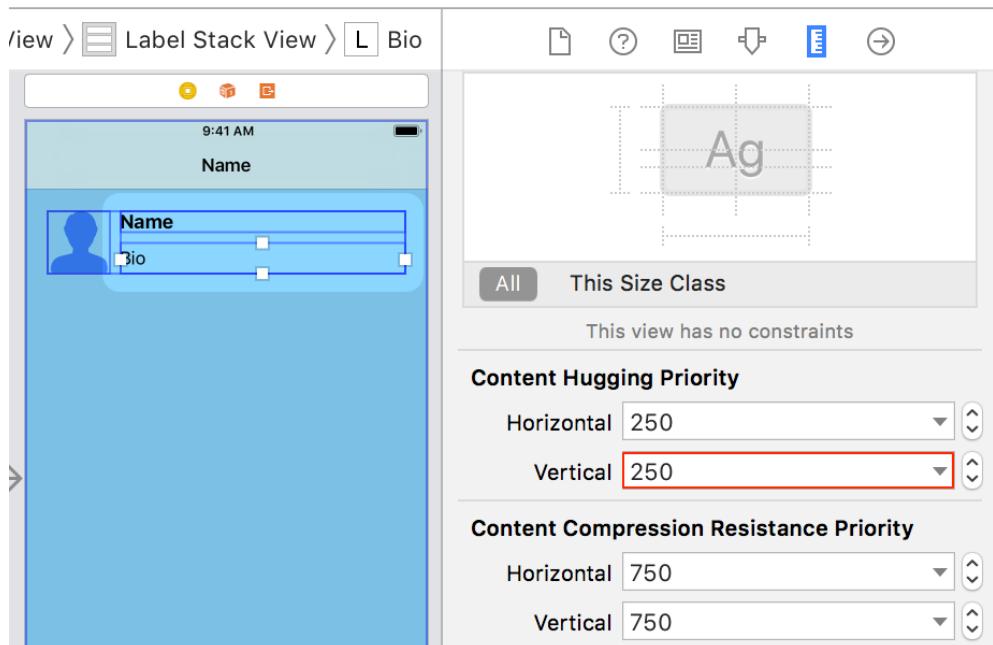


6. We don't want the image view to stretch horizontally. When using Interface Builder the image view and labels all have a default horizontal content-hugging priority of 251. To fix the ambiguity and make the labels stretch before the image view lower the priority of both labels to 250:



7. The height of the profile stack view is set either by the height of the image view or the height of the label stack view. If the label stack view height is smaller than the image view, it's stretched to fill the space. Both labels have default vertical content-hugging priorities of 251 making the vertical layout ambiguous.

To make the bio label stretch first use the size inspector to lower its priority to 250:



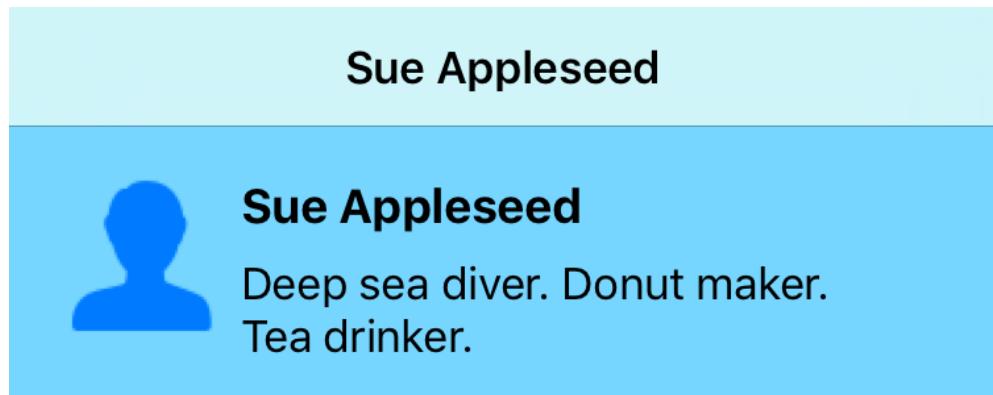
Summarizing the horizontal and vertical content-hugging priorities

for the three views:

View	Horizontal Hugging	Vertical Hugging
Image View	251	251
Name Label	250	251
Bio Label	250	250

Horizontally, the two labels stretch first. Vertically, the bio label stretches.

8. Populating the layout with some data in the view controller to see the working layout:



When creating the same layout in code, we can set the layout priorities when creating the image view and labels (see sample code: [StackProfile-v1](#)):

1. The default content hugging priority when creating image views and labels in code is 250. When we create the name label property in the view controller we need to increase the vertical content hugging priority to 251:

```
private let nameLabel: UILabel = {
    let label = UILabel()
    label.font = UIFont.boldSystemFont(ofSize:
        ViewMetrics.nameFontSize)
    label.numberOfLines = 0
```

```
label.setContentHuggingPriority(.defaultLow + 1, for:
    .vertical)
return label
}()
```

2. When creating the image view, we set the content mode and increase the horizontal content hugging priority to 251:

```
private let profileImageView: UIImageView = {
    let imageView = UIImageView()
    imageView.contentMode = .top
    imageView.setContentHuggingPriority(.defaultLow + 1,
        for: .horizontal)
    return imageView
}()
```

3. The bio label uses the default priorities:

```
private let bioLabel: UILabel = {
    let label = UILabel()
    label.font = UIFont.systemFont(ofSize:
        ViewMetrics.bioFontSize)
    label.numberOfLines = 0
    return label
}()
```

4. With the views created we can create the vertical label stack view:

```
private lazy var labelStackView: UIStackView = {
    let stackView = UIStackView(arrangedSubviews:
        [nameLabel, bioLabel])
    stackView.axis = .vertical
    stackView.spacing = UIStackView.spacingUseSystem
    return stackView
}()
```

5. Then the horizontal profile stack view:

```
private lazy var profileStackView: UIStackView = {
    let stackView = UIStackView(arrangedSubviews:
        [profileImageView, labelStackView])
    stackView.translatesAutoresizingMaskIntoConstraints = false
    stackView.spacing = UIStackView.spacingUseSystem
    return stackView
}()
```

Note that we need to disable the auto resizing mask constraints for the top stack view.

6. Finally, we can add the constraints when the view loads to pin the top stack view to the margins:

```
let margin = view.layoutMarginsGuide
NSLayoutConstraint.activate([
    profileStackView.leadingAnchor.constraint(equalTo:
        margin.leadingAnchor),
    profileStackView.topAnchor.constraint(equalTo:
        margin.topAnchor),
    profileStackView.trailingAnchor.constraint(equalTo:
        margin.trailingAnchor)
])
```

Dynamically Updating Stack Views

Stack views automatically update the layout of their arranged subviews when you make a change. This includes:

- Adding or removing an arranged subview.
- Changing the `isHidden` property on any of the arranged subviews.
- Changing the `axis`, `alignment`, `distribution` or `spacing` properties.

Even better, you can animate any of these changes.

Animating Changes To A Stack View

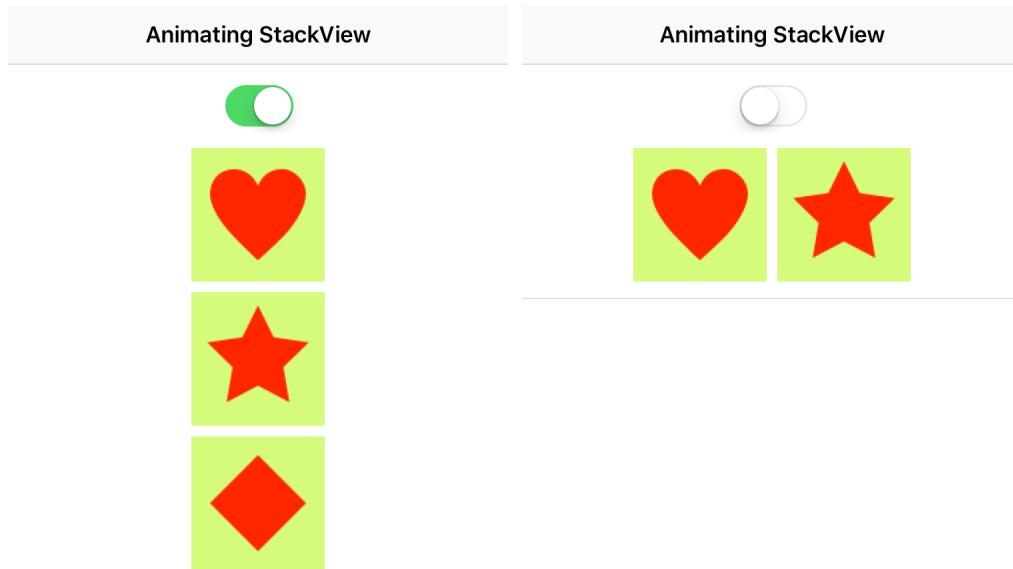
To animate changes you make to a stack view embed them in a `UIView` animation block. For example, to create an animation that takes 0.25 seconds to run:

```
UIView.animate(withDuration: 0.25) {
    // Change stack view properties here
}
```

Or if using property animators introduced with iOS 10:

```
UIViewPropertyAnimator.runningPropertyAnimator(withDuration:
    0.25, delay: 0, options: [], animations: {
    // Change stack view properties here
}, completion: nil)
```

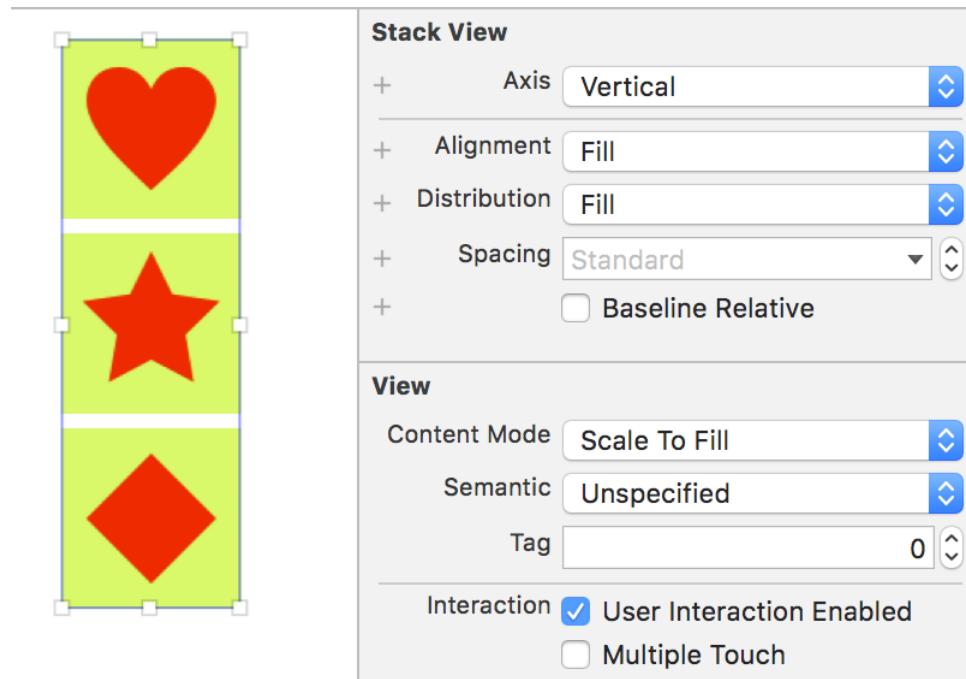
To see how this works let's animate switching the axis of a stack view between vertical and horizontal.



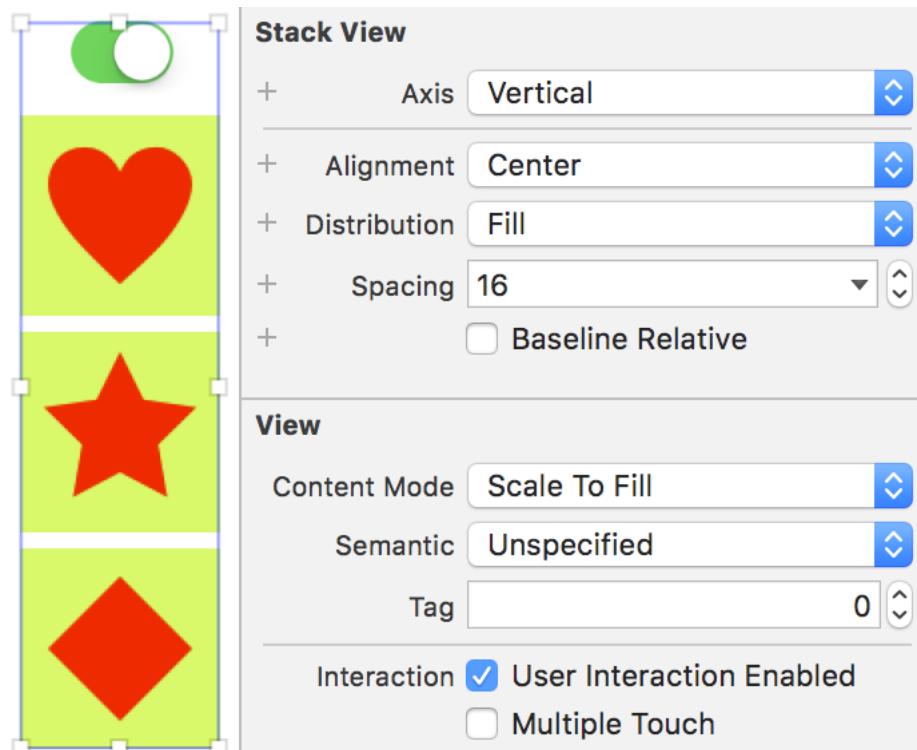
I also hide the last arranged subview in the stack view when horizontal. It's often better to hide a view rather than add and remove it. A neat feature of stack views is that they automatically adjust the layout when you hide or unhide the arranged views (see sample code: [AnimatingStack-v1](#)):

I'm using three image views in this project that are 100x100 points in size. You can find these in the sample code or add your own.

1. Starting from the Single View App iOS template add the three images to the asset catalog. Then drag three image views from the object library onto the root view of the view controller and set the image of each to one of the images from the asset catalog.
2. Embed the three images in a vertical stack view, keep the default fill alignment and distribution. Use the "Standard" system spacing:

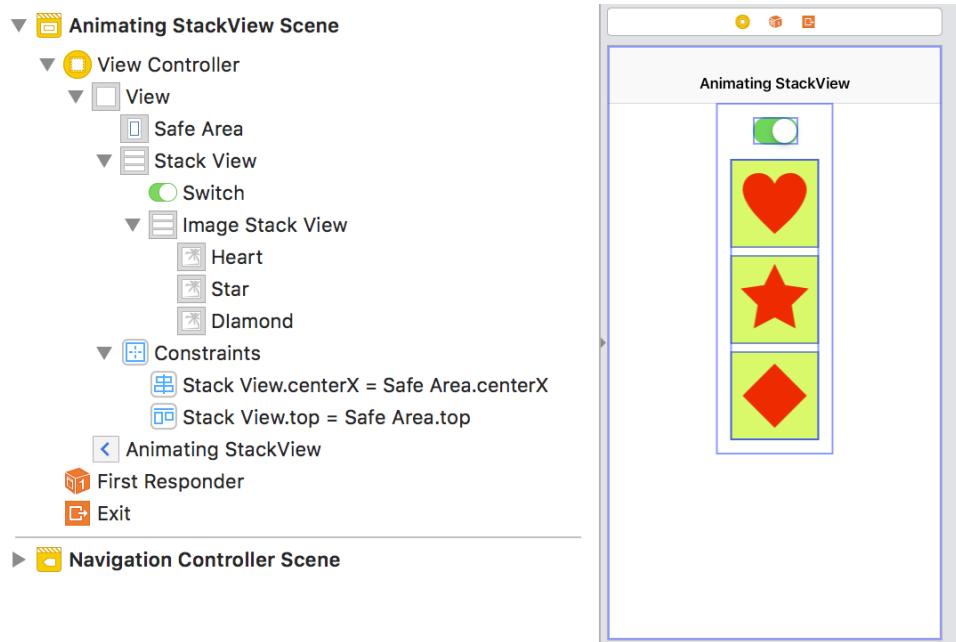


3. Add a switch to the layout, placing it above the image stack view and then embed both the switch and the image stack view in another vertical stack view with 16 points of spacing and a center alignment:



4. Add constraints to center the top stack view in the safe area and pin

it to the top of the safe area:



I also embedded the view controller in a navigation controller to add a title and gave the top stack view a 16 point margin for some extra spacing.

- To animate changes to the image stack view, we need connections from the storyboard to outlets in the view controller. Using the Xcode assistant editor control-drag from the inner image stack view and the switch into the view controller to create outlets:

```
@IBOutlet private var imageStackView: UIStackView!
@IBOutlet private var axisSwitch: UISwitch!
```

- We also need a connection from the switch to an action method in the view controller for when the user changes the switch value:

```
@IBAction private func axisChanged(_ sender: UISwitch) {
}
```

- Let's also create a helper method to configure the stack view based on the position of the switch:

```
private func configureAxis() {
    imageStackView?.axis = axisSwitch.isOn ? .vertical :
        .horizontal
```

```
if let lastImageView =  
    imageStackView.arrangedSubviews.last {  
    lastImageView.isHidden = !axisSwitch.isOn  
}  
}
```

8. We can use this method to configure our image stack view when the view is first loaded:

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    configureAxis()  
}
```

9. The `axisChanged` method also uses the `configureAxis` method to update the stack view configuration in an animation block:

```
@IBAction private func axisChanged(_ sender: UISwitch) {  
    let animator = UIViewPropertyAnimator(  
        duration: 2.0, dampingRatio: 0.2,  
        animations: {  
            self.configureAxis()  
        })  
    animator.startAnimation()  
}
```

I'm using a springy animation but you can use any of the animation options. If you're still supporting iOS 9 use the older style `UIView` animation block:

```
UIView.animate(withDuration: 2.0, delay: 0,  
    usingSpringWithDamping: 0.2,  
    initialSpringVelocity: 0,  
    options: [], animations: {  
        self.configureAxis()  
    }, completion: nil)
```

If you run this project on an iPhone and rotate the device to landscape, you will notice that it doesn't fit. The bottom image gets truncated when the stack view is vertical. It would be better if it adapted automatically when rotated. We'll see how to do that when we look at [Adapting Properties For Size Classes](#).

Adding Background Views

How would you give your stack view a background color? If you try to set the background color in Interface Builder or in code, it may surprise you that it does nothing. The reason is that **a stack view is a non-rendering subclass of UIView**. Its job is to manage the layout of views you add to `arrangedSubviews`.

Remember that even though `UIStackView` is not a normal subclass of `UIView`, it still has a `subviews` property. You cannot do it with Interface Builder, but it's perfectly ok to add extra subviews to a stack view that are not part of `arrangedSubviews`.

We can create a background view and add it as a subview of the stack view. If we insert it at index 0 in `subviews`, it appears behind all other subviews. If we add it to the end of `subviews`, it appears in front of all other subviews.

Since the background view is not an arranged subview, we need to take care of the constraints to pin it to the edges of the stack view.

Adding A Background View

Let's give our animated stack view a purple background (see sample code: [AnimatingStack-v2](#)):

1. Connect an outlet in the view controller to the root container stack view in the storyboard. Use the assistant editor to control-drag from the stack view into the controller:

```
@IBOutlet private var containerStackView: UIStackView!
```

2. Add a `setupView()` method to create and add the background view to the stack view:

```
private func setupView() {
    let backgroundView = UIView()
    backgroundView.translatesAutoresizingMaskIntoConstraints = false
    backgroundView.backgroundColor = .purple
    backgroundView.layer.cornerRadius = 10.0
    containerStackView.insertSubview(backgroundView, at: 0)
}
```

The last line is the crucial line. We make the background view the first view in `subviews`, so it appears behind the other subviews.

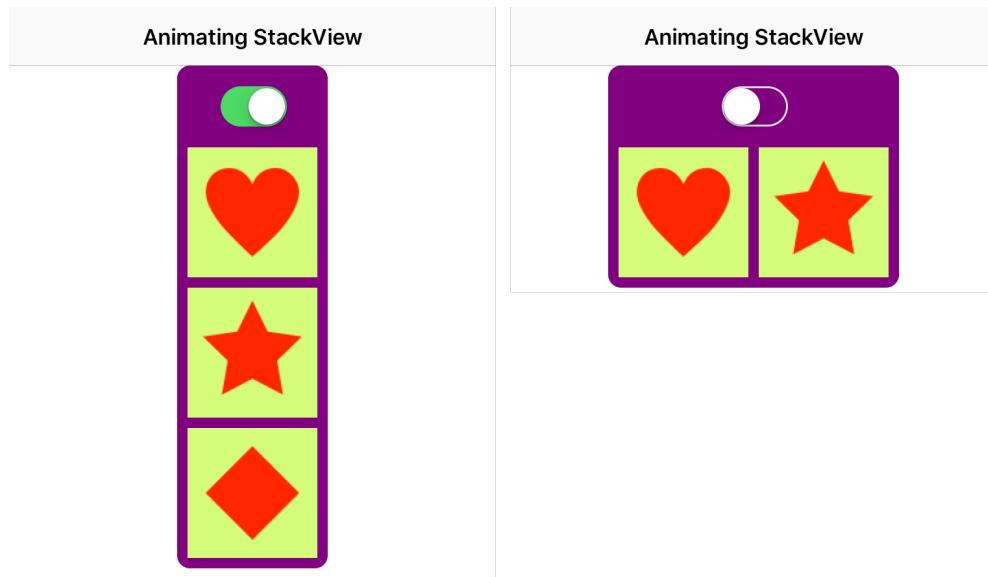
3. Add constraints to pin the background view to the edges of the stack view:

```
NSLayoutConstraint.activate([
    containerStackView.leadingAnchor.constraint(
        equalTo: backgroundView.leadingAnchor),
    containerStackView.trailingAnchor.constraint(
        equalTo: backgroundView.trailingAnchor),
    containerStackView.topAnchor.constraint(
        equalTo: backgroundView.topAnchor),
    containerStackView.bottomAnchor.constraint(
        equalTo: backgroundView.bottomAnchor)
])
```

4. Finally we call the `setupView()` method from `viewDidLoad`:

```
override func viewDidLoad() {
    super.viewDidLoad()
    setupView()
    configureAxis()
}
```

5. Here's how it looks when the stack view is vertical and horizontal:



Adding a background view to a stack view is something I do often enough that I like to make it into an extension of `UIStackView` (see sample code: [AnimatingStack-v3](#)):

Let's create a method to add an unarranged view to a stack view. This

method takes care of creating the view, setting its color and radius, adding it to the `subviews` array at the specified index and then activating constraints to pin it to the edges of the stack view:

```
// UIStackView+Extension.swift
import UIKit

public extension UIStackView {

    @discardableResult
    func addUnarrangedView(color: UIColor, radius: CGFloat = 0,
                           at index: Int = 0) -> UIView {
        let view = UIView()
        view.translatesAutoresizingMaskIntoConstraints = false
        view.backgroundColor = color
        view.layer.cornerRadius = radius
        insertSubview(view, at: index)
        NSLayoutConstraint.activate([
            view.leadingAnchor.constraint(equalTo: leadingAnchor),
            view.trailingAnchor.constraint(equalTo: trailingAnchor),
            view.topAnchor.constraint(equalTo: topAnchor),
            view.bottomAnchor.constraint(equalTo: bottomAnchor)
        ])
        return view
    }
}
```

We can then write one-line methods to add background and foreground views:

```
@discardableResult
func addBackground(color: UIColor, radius: CGFloat = 0) ->
    UIView {
    return addUnarrangedView(color: color, radius: radius, at:
        0)
}

@discardableResult
func addForeground(color: UIColor, radius: CGFloat = 0) ->
    UIView {
    let index = subviews.count
    return addUnarrangedView(color: color, radius: radius, at:
        index)
}
```

This reduces the view setup code to a single line in our view controller:

```
containerStackView.addBackground(color: .purple, radius: 10.0)
```

Stack View Oddities

Stack views are a convenience that saves us from manually creating constraints for common horizontal and vertical layouts. Unfortunately, they are not without issues, undocumented features or even bugs.

Baseline Alignment Not Working



This is a bug (#32291130) that at the time of writing exists in all iOS versions including iOS 12.0.

Using a `.firstBaseline` or `.lastBaseline` alignment can sometimes lead to surprising results. The problem comes when a view other than the first arranged subview extends furthest above or below the baseline. Take this example of three single line labels where the second label uses a font twice the size of the other labels:

```
let stackView = UIStackView(arrangedSubviews: [label1, label2, label3])
stackView.spacing = 8.0
stackView.alignment = .firstBaseline
```

If I center this stack view just below the top layout guide I expect to see something like this:



I gave the labels a green background and the stack view a yellow background to make it easy to see the frame sizes. The stack view width and height are not constrained, so each label has its intrinsic content size.

The stack view sets its height by the label edges that extends furthest above and below the baseline.

What you get if you try this is a stack view that appears to ignore the height of the second label. Instead, the stack view fixes its top edge to the first label:



If you use the Xcode view debugger to examine the constraints used by the stack view you can easily see why this happens. There's a constraint from the top of the stack view to the top of the first label:

```
UISV-canvas-connection: stackView.top <= label.top @ 1000
```

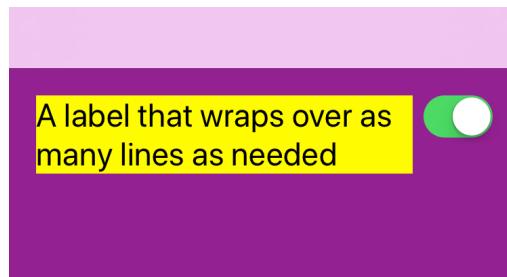
Note that this constraint has a less than or equal to relation. The top of the stack view can never be below the top edge of the first label. The problem is that this constraint is missing for the other two labels. Adding the missing constraints fixes the problem (you can add these constraints in code or with Interface Builder):

```
NSLayoutConstraint.activate([
    stackView.topAnchor.constraint(lessThanOrEqualTo:
        label2.topAnchor),
    stackView.topAnchor.constraint(lessThanOrEqualTo:
        label3.topAnchor)
])
```

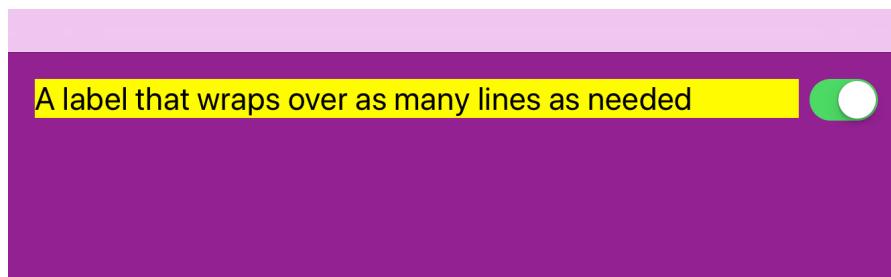
For a `.lastBaseline` alignment replace `topAnchor` with `bottomAnchor` and use a greater than or equal to relation.

Stack Views With Multi-Line Labels

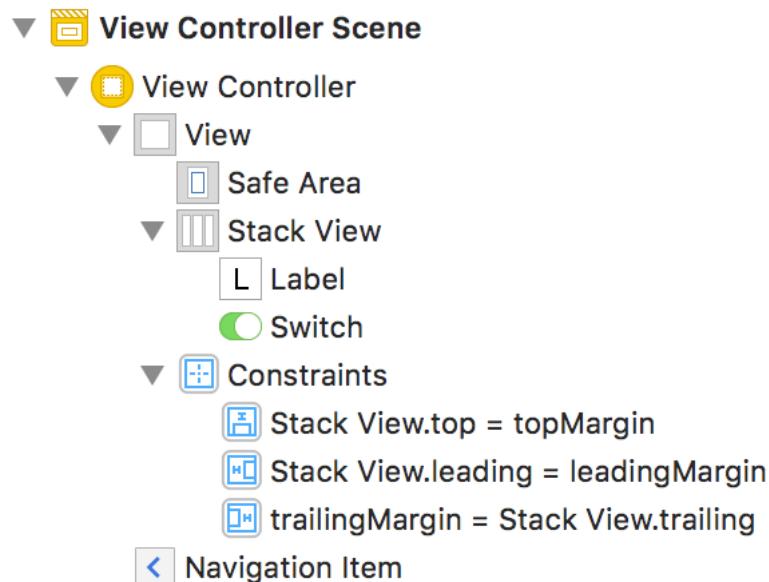
Take a look at this layout with a label and switch. In portrait the label wraps over two lines:



In landscape, the label stretches to fill the available space. The switch keeps its natural size (see sample code: [MultiLine-v1](#)):



When deciding how to build this layout it seems an obvious candidate for a horizontal stack view. I can constrain the stack view to the leading, top and trailing margins. This fixes the position and width but allows the height to grow with the content:



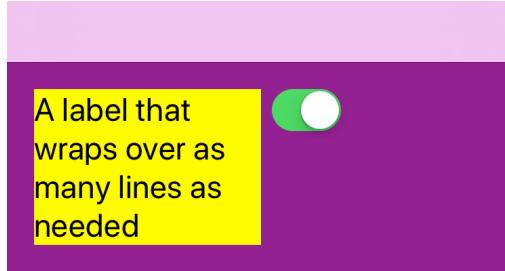
The stack view is using a `.fill` distribution and `.top` alignment. To allow the label to use as many lines as needed, I set the `numberOfLines` property to 0.

I want the switch to stay at its intrinsic content size and the label to stretch and squeeze to fill the available width. That means keeping the horizontal content priorities for the label lower than for the switch.

The default content-hugging priority of the label (251) is already lower than the switch (750). I need to lower the compression-resistance priority of the label (to 749) to make it lower than the default for the switch (750):

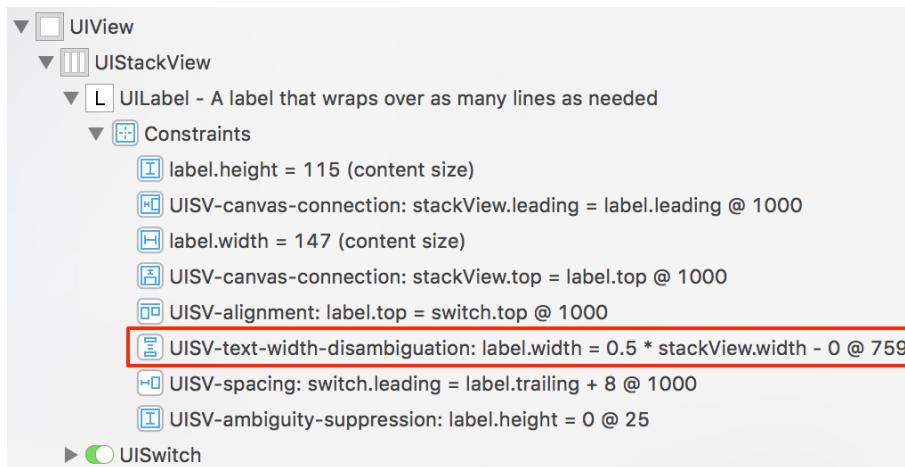
View	Content Hugging	Compression Resistance
Label	251	749
Switch	750	750

That should be enough. Unfortunately, this is what happens:



What's going on? The stack view has resized both the label and the switch to what looks like half the width of the stack view.

This is one of the problems with using a stack view. It's an opaque container adding constraints for us which is convenient until it does something unexpected. Luckily we can peek inside using the view debugger to see the constraints it has created:



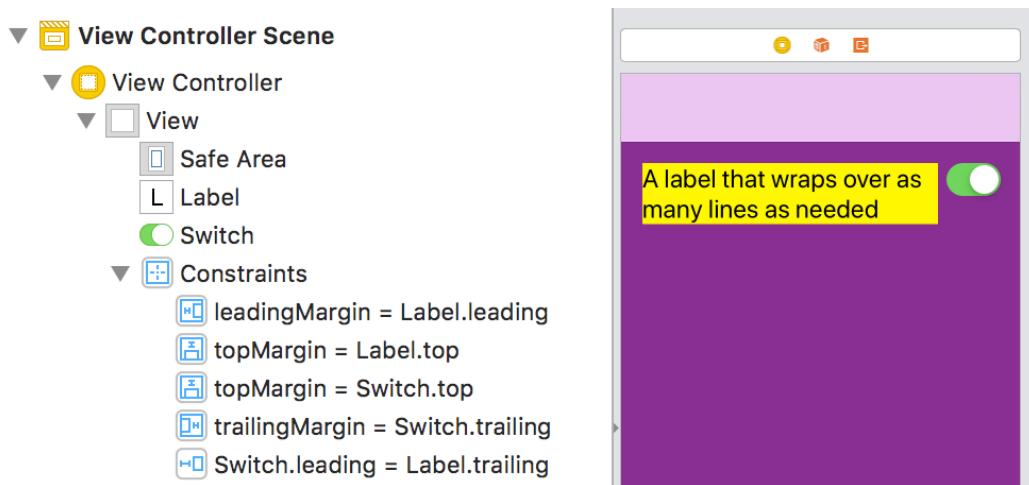
The `UISV-text-width-disambiguation` constraint is the one that stands out:

```
label.width = 0.5 * stackView.width - 0 @ 759
```

This is an optional constraint added by the stack view with a priority of 759 that wants to make the width of the label half the width of the stack view. Our switch has a content hugging priority of 750 which means this disambiguation constraint takes priority forcing both the label and switch to resize.

There's an easy workaround. We can increase the content hugging priority of the switch to 760. That fixes the problem, but it's not a solution that feels good to me. You should not need to use the view debugger to understand how to configure your layout priorities. At the time of writing, this workaround is still necessary in iOS 12. Hopefully, Apple will either fix or at least document this behavior in the future.

A stack view is a convenience that saves us from creating constraints for some common layouts. When you end up fighting with the stack view, don't be afraid to use a more straightforward approach and create the layout yourself (see sample code: [MultiLine-v2](#)):



Not using a stack view means I need to add a couple more constraints. I also still need to lower the compression resistance priority of the label. This time though I know exactly why each constraint is there and the layout works as we expect.

Key Points To Remember

Some hints and tips to help you get the most out of stack views:

- Not all distributions and alignments make sense in all configurations of a stack view. For example, if you don't constrain the width of a horizontal stack view only the `.fill` and `.fillEqually` distributions make sense and then only if all (`.fill`) or at least one (`.fillEqually`) of the arranged subviews have an intrinsic content width.
- Don't forget you still need to adjust content-hugging and compression-resistance priorities any time the stack view can squeeze or stretch the arranged subviews to fit the available space.
- If you're creating your stack views in code don't forget to set `translatesAutoresizingMaskIntoConstraints` to `false` for the top stack view. The stack view does it for you for any arranged views (including other stack views) you add to the stack view.
- A common mistake is to forget that a view stays in the `subviews` array of a stack view after you call `removeArrangedSubview` to remove it. You either need to use `removeFromSuperview` to remove the view yourself or hide it if you don't want to see it on screen.
- Be careful any time you add extra constraints to the views managed

by a stack view. It's easy to create a conflict between your constraints and the constraints that the stack view creates. If you find you need to add a lot of extra constraints, you may want to rethink using a stack view.

- Don't forget that you can set a margin for a stack view when you want to pad the content.
- Stack views are convenient, but also have some limitations and issues. Don't be afraid to skip them and create the constraints yourself if you find it more manageable.

Test Your Knowledge

Try these challenges to get familiar with using a stack view.

Challenge 8.1 Fixed Width, Flexible Height

A typical setup for a stack view is to fix its position and allow either the width or height to vary to fit the content. In this first layout, I want to center a label and three buttons vertically. The views fill the available width between the margins of the superview. The label text is center aligned with a 24pt system font. The buttons are using 18pt system font. There's a standard amount of vertical spacing.

Engine Power



1. Build this layout using a stack view. You can use Interface Builder or build it in code (or do both).
2. If you use Interface Builder, you should not need to write any code.

Hints And Tips

1. You should only need to create three constraints for this layout.
2. You don't need to change content hugging or compression resistance priorities.
3. You can build this layout with a single vertical stack view using the default `.fill` distribution and alignment.
4. If you create the stack view in code, remember that the default axis is `.horizontal`.

Challenge 8.2 Stretch To Fill

I pinned this layout to the margins of the root view on all sides forcing the content to stretch to fill the available space. The layout is a single image view, and a button arranged vertically separated by a standard amount of spacing.

- You can use any image you want. I'm using a vector based asset.
- The button is using a 24 point system font.
- The image should fill the available space, keeping the button at its natural size.



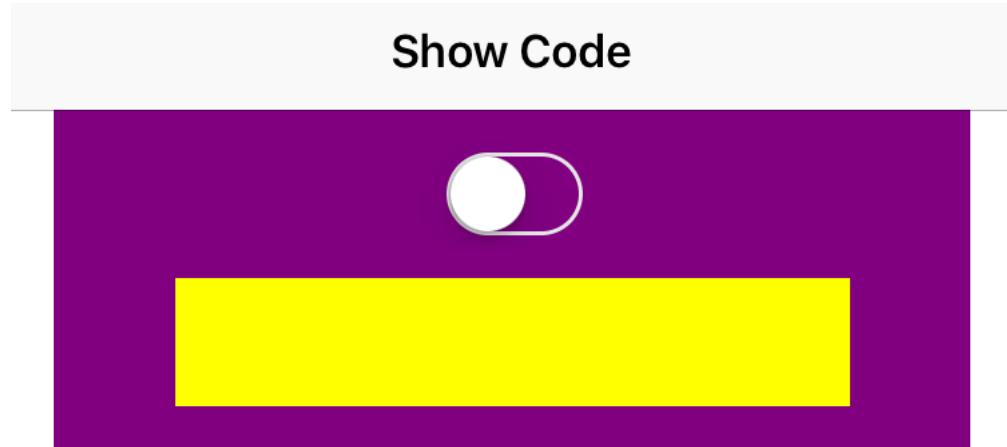
1. Build this layout using a stack view. You can again choose to use Interface Builder or build it in code.
2. If you use Interface Builder, you should not need to write any code.

Hints And Tips

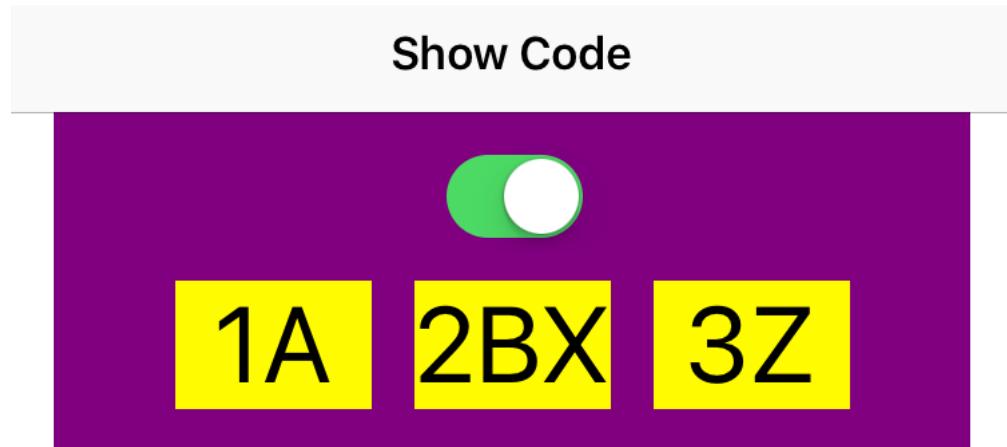
1. You should only need to create four constraints to pin the stack view to the margins.
2. My vector image is only 100 x 100 points in size, so I'm using an aspect fit content mode to control how it scales.
3. The button has a default content hugging priority of 250.
4. The image view has a default content hugging priority of 250 when created in code or 251 when created in Interface Builder.
5. You want the image view height to stretch and the button to keep its intrinsic content height.
6. You want the image view to have a lower vertical content hugging priority than the button.

Challenge 8.3 Show The Secret Code

In this layout we have a secret code that I want to hide with a yellow banner:



I want to show the code only when the switch is in the on position:



I'm using three labels to show the “secret” code in a 40pt system font. The labels have equal width, are center aligned and separated by 16 points of spacing.

I centered the switch and positioned it 16 points above the secret code labels.

I want the purple background pinned to the top, leading and trailing margins with an internal 16 point margin on all sides.

1. Build this layout using stack views.
2. For bonus points animate the action of showing and hiding the secret

code.

Hints And Tips

1. This layout may look tricky. Don't let the details put you off. You can create this layout with two stack views and three constraints.
2. You cannot build this layout entirely in Interface Builder. You need to add the purple background and yellow banner in code as well as the switch action method to show and hide the banner.
3. Start with the three text labels to show the secret code. Once you have that figured out add the switch above the labels. Finally, add the purple background and the yellow banner.
4. The three text labels need to be in a horizontal stack view. Which distribution do they need to make them all have equal width?
5. Once you have the horizontal stack view done, embed it in a vertical stack view with the switch. To center the switch, you need to change the default alignment of the vertical stack view to `.center`.
6. The vertical stack view has a margin of 16 points on all sides. See [Stack View Margins](#) if you need help.
7. See [Adding Background Views](#) for help on adding the purple background and yellow foreground views to the root vertical stack view.
8. The only three constraints you need to add are to pin the root stack view to the top, leading and trailing margins of the root view.
9. See [Animating Changes To A Stack View](#) for help on creating the animation. Animate changing the alpha property of the banner view from 0 (transparent) to 1.0 (opaque).

Chapter 9

Understanding The Layout Engine

I found learning Auto Layout to be tricky because I didn't have a good mental model of what the layout engine was doing. It felt like a closed magic box. I would feed it some constraints, and a layout would pop out that might or might not be what I expected.

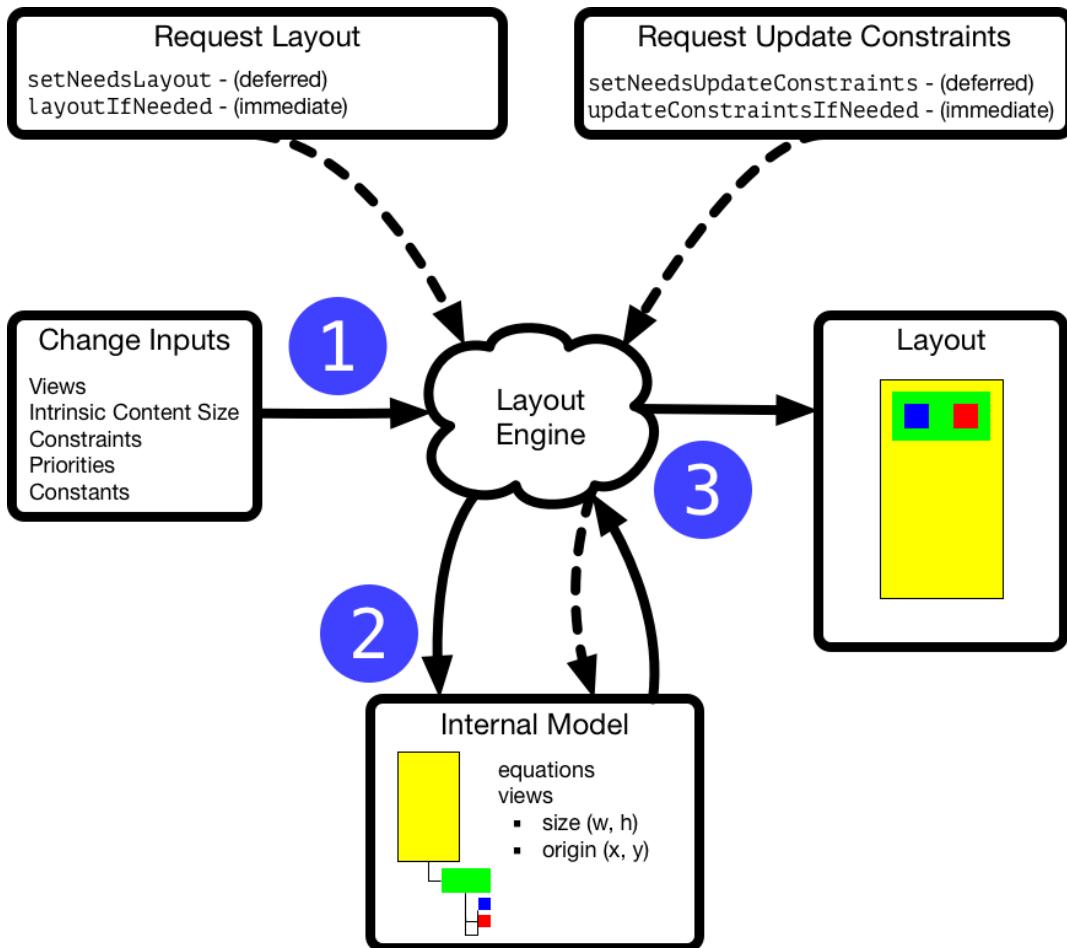
In this chapter, I'm going to look inside the layout engine to help you build a better mental model of what's happening. Along the way we look at some of the ways you can interact with the Auto Layout engine. Topics you learn in this chapter:

- How does the layout engine create a layout based on your constraints and when does it update your views.
- The methods you can override in your views and view controllers to interact with the layout engine.
- The difference between `setNeedsLayout` and `layoutIfNeeded` and when to use them.
- Why you probably don't need to use `updateConstraints`.
- How to animate the changing of constraints.
- How to override `layoutSubviews` to create dynamic layouts that you cannot directly describe with constraints.
- How to use alignment rectangles to position views with shadows or badges.

The Layout Pass

You may not have noticed but when you add, change or remove constraints the frames of your views don't update right away. Recalculating the layout

and updating the display for every change would be inefficient. Instead, your changes schedule the layout engine that belongs to the window to run a layout pass at the next opportunity on the application run loop.



A typical layout cycle has several steps:

- Trigger:** You change an input to the layout engine. This can be by adding or removing subviews, changing the intrinsic content size, activating/deactivating constraints or changing the priority or constant value of a constraint.
- Update Model:** The layout engine has an internal model of the size and position of each view and the equations that describe the relationships between those views. Changing an input causes the layout engine to update its internal model and solve the equations for new values for the size and position of each view.

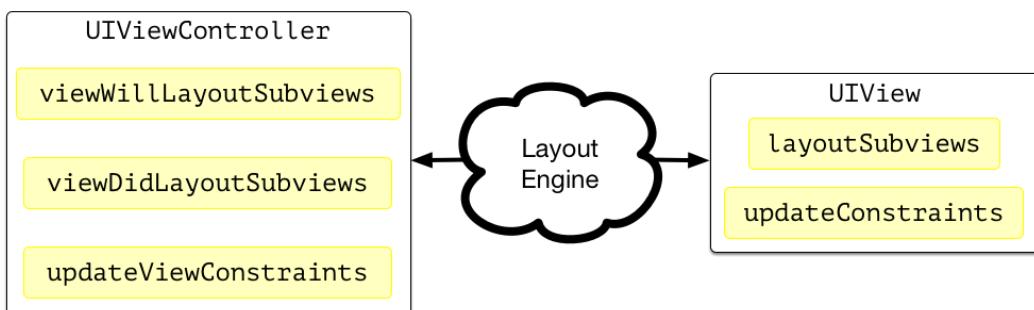
At this point, only the internal model has changed. Views that now have a new size or position in the model call `setNeedsLayout()` on

their superview which schedules a deferred layout pass to run some time later on the application run loop.

3. **Deferred Layout Pass:** When the layout pass runs it makes two passes over the view hierarchy. The first pass gives you a last opportunity to update any constraints. The second pass finally calls `layoutSubviews()` on each view allowing them to update the size and position of their subviews to match the internal model repositioning any views as needed.

There are some methods you can override in both the view controller and view to interact with the layout engine during the two layout passes:

- In the update constraints pass the layout engine calls the view controller's `updateViewConstraints` method and for views with pending updates the `updateConstraints` method.
- In the update layout pass the layout engine calls the view controller's `viewWillLayoutSubviews` and `viewDidLayoutSubviews` methods and the `layoutSubviews` method of any views needing layout.



Let's look at the two steps in the layout pass in more detail:

Updating Constraints

The first pass through the view hierarchy is from bottom-up to allow you to change the constraints before the layout engine repositions the views. The layout engine calls `updateConstraintsIfNeeded` on each view to check if it has the latest constraints. For views marked as needing to update their constraints, the layout engine calls their `updateConstraints` method where you can make the changes.

Call `setNeedsUpdateConstraints` to request an update constraints pass for a view. If you want the layout engine to update its model immediately call the `updateConstraintsIfNeeded` method. We'll see an example of this later in this chapter.

Repositioning Views

The second pass through the view hierarchy is a top-down pass to reposition the views. It's only during this pass that the layout engine updates the view frames to match its internal model.

For each view needing layout the default behavior of `layoutSubviews()` sets the bounds and center of each subview to the new values from the layout engine model. If you have a custom layout that you cannot describe with constraints you can override `layoutSubviews()`. See [Custom Layouts](#) for an example.

Call `setNeedsLayout` or `layoutIfNeeded` to request a layout pass for a view. The key difference between these methods is when the layout pass happens to update the view:

- `setNeedsLayout`: returns immediately without updating the layout. Instead, it marks the layout of the view as changed and schedules a deferred layout pass to run on the application run loop.
- `layoutIfNeeded`: calls `layoutSubviews` on the receiver if there are pending changes to force the layout engine to immediately update the size and position of subviews from its internal model.



Only call these methods on the main thread.

Should You Use `updateConstraints`?

Apple's documentation has not always been clear about when you should use `updateConstraints` or `updateViewConstraints`. This has led some developers to assume they needed to use these methods to create or update their constraints. Since these methods are called each time there's a layout pass it's common to see code like this to ensure initial constraints are only added once:

```
private var hasInitialConstraints = false

override func updateConstraints() {
    if !hasInitialConstraints {
        hasInitialConstraints = true
        NSLayoutConstraint.activate([
            // Add initial constraints
        ])
    }
}
```

```
    }
    super.updateConstraints()
}
```

This isn't wrong, just unnecessary. Apple's present recommendation for `updateConstraints` is much more explicit:

You should only override this method when changing constraints in place is too slow, or when a view is producing a number of redundant changes.

In practice this means you rarely need to use `updateConstraints` or `updateViewConstraints`:

- Create your initial constraints in Interface Builder or in code when creating the view. For example from `viewDidLoad` or the initializer of a custom view.
- When you need to change constraints in response to an external event make those changes in place. This might be in the action method of a control, in the callback of a view controller responding to a size change or when handling the arrival of new data.

Only if you find that changing constraints in place is too slow should you move those changes to `updateConstraints`. You would then schedule the update pass by calling `setNeedsUpdateConstraints`. Consider it a performance optimization to try for time-critical layout before falling back to manual frame-based layout.

If you do use `updateConstraints`:

- Remember you're in the middle of a layout pass so keep it quick.
- Don't remove and recreate all your constraints each time. At WWDC 2018 Apple warned that unnecessary churning of constraints in this way was one of the most common errors they see.
- To avoid creating a loop, don't call `setNeedsUpdateConstraints` from inside `updateConstraints`.
- Call `super.updateConstraints()` as the last step.

Animating Constraints

Perhaps the most common example of directly interacting with the layout engine is when you want to animate changes to your constraints. As we have seen, changing constraints doesn't directly update the layout. A later layout pass updates the views. To animate those changes you

call `layoutIfNeeded` to force the layout engine to update the size and position of views inside an animation block. Using block-based property animators here's a typical code snippet to animate constraint changes:

```
// Change constraints
widthConstraint.constant = 100.0

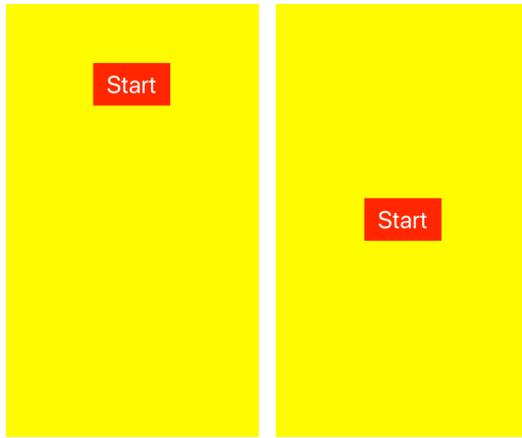
let animator = UIViewPropertyAnimator(duration: 1.0, curve:
    .easeInOut) {
    // Update layout
    self.view.layoutIfNeeded()
}
animator.startAnimation()
```

If you're using the older `UIView` animation it might look like this:

```
UIView.animate(withDuration: 1.0) {
    self.view.layoutIfNeeded()
}
```

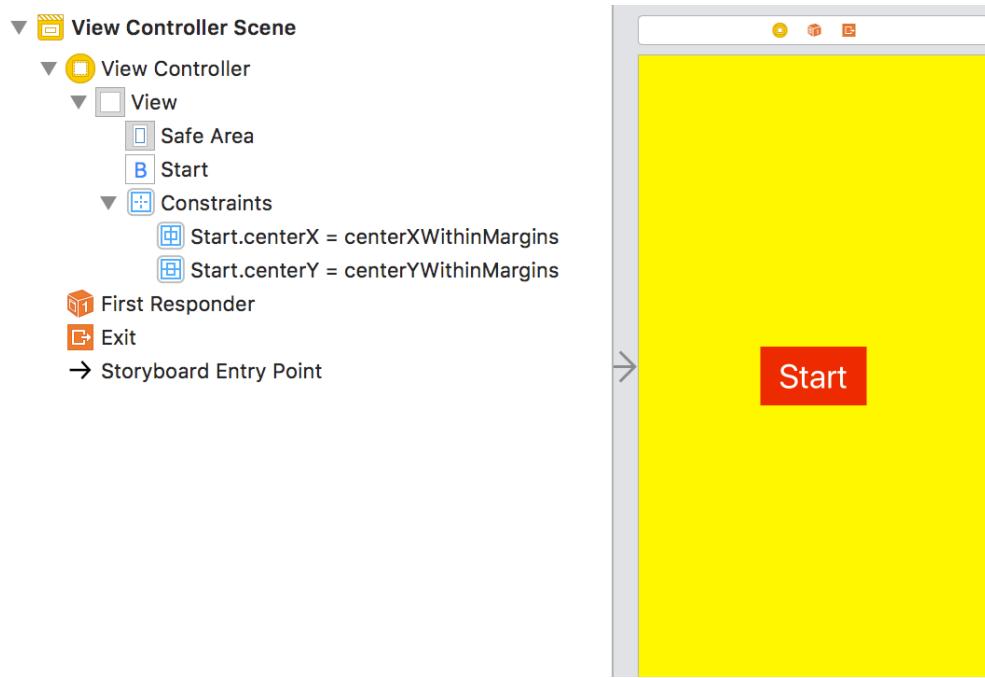
You can change the constraints inside the animation block if you prefer. It's the changes you force the layout engine to apply immediately with `layoutIfNeeded` that are animated.

Take a look at this layout with a button that starts out near the top of the screen (left image):



When the view appears I want the button to drop down into the center of the screen (right image) to bring it to the attention of the user (see sample code: [Animate-v1](#)):

1. To build this layout, I started from the Single View App Xcode template and added a button to my storyboard:



I added constraints to center the button horizontally and vertically between the margins. This is the final position of the button. When this view loads I want the button to be near the top of the screen. I'll do that by adjusting the constant value of the vertical center constraint.

2. We need an outlet in the view controller that connects to the vertical constraint in Interface Builder. Using the Assistant Editor control-drag from the centerY constraint in Interface Builder to the view controller to create the connection:

```
@IBOutlet var centerConstraint: NSLayoutConstraint!
```

3. To keep my code tidy I'm using a private enum to store the vertical offset of the button together with some animation settings we'll use shortly:

```
private enum AnimationMetrics {  
    static let offset: CGFloat = -200  
    static let duration: TimeInterval = 1.0  
    static let delay: TimeInterval = 1.0  
    static let damping: CGFloat = 0.4  
}
```

4. When my view first loads I want to position the button using the offset.

I do that by setting the constant value of the vertical constraint in `viewDidLoad`:

```
override func viewDidLoad() {
    super.viewDidLoad()
    centerConstraint.constant = AnimationMetrics.offset
}
```

- When the view appears on screen, I animate it moving back to the center position without the vertical offset.

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    animateButton(withDuration: AnimationMetrics.duration,
                  damping: AnimationMetrics.damping, delay:
                  AnimationMetrics.delay)
}
```

- In the `animateButton` method we first change the constant value of the center constraint. Setting the value to `0` moves the button back to the vertical center of the screen.

```
private func animateButton(withDuration duration:
                           TimeInterval, damping: CGFloat, delay: TimeInterval = 0)
{
    centerConstraint.constant = 0.0
```

Remember from our discussion of the layout engine that at this point the center position of the button has not yet changed.

- Inside the animation block (I'm using a spring animation) we call `setNeedsLayout` to force the layout engine to update the button position:

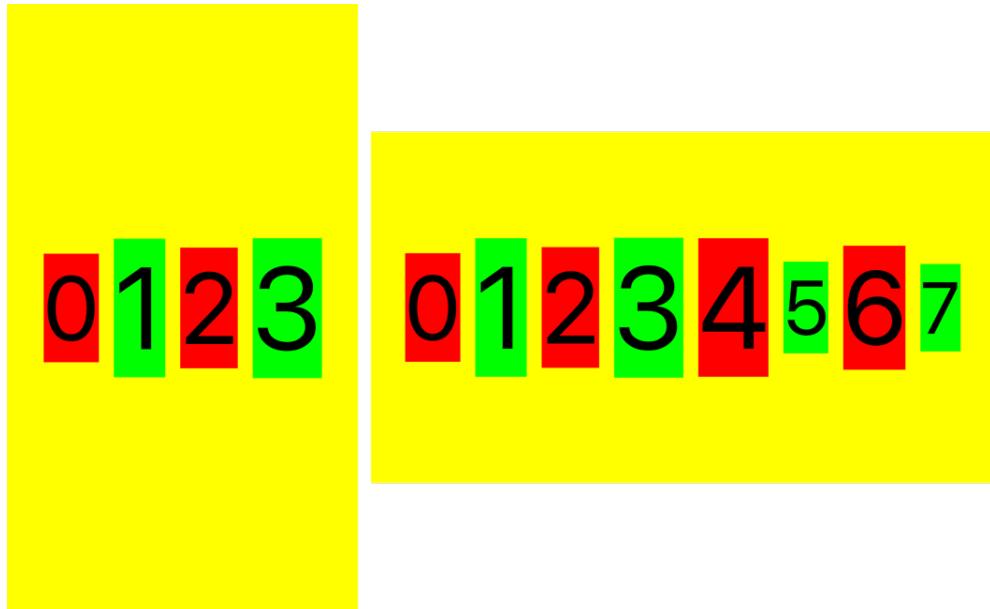
```
let animator = UIViewPropertyAnimator(duration:
                                         duration, dampingRatio: damping, animations: {
    self.view.layoutIfNeeded()
})
```

- Finally, I start the animation after a short delay:

```
animator.startAnimation(afterDelay: delay)
```

Custom Layouts

What do you do if you have a layout that you cannot easily describe with Auto Layout constraints? Take a look at this layout where I have a collection of views in a horizontal row:



Assume that we don't know how many views we have or how big each view is until runtime. How can we show as many of the views as possible in the available width without scrolling? (See sample code: [Layout-v1](#)).

1. Let's create a custom view that accepts the set of views to display. I called this custom view `PreviewPane`. The external interface looks like this:

```
class PreviewPane : UIView {  
    var spacing: CGFloat { get set }  
    func show(_ items: [UIView])  
}
```

It has a single property to configure the spacing between the views and a method that accepts an array of views to show.

2. We follow our usual approach for creating a custom view that works when used with or without Interface Builder:

```
class PreviewPane: UIView {  
  
    override init(frame: CGRect) {
```

```
super.init(frame: frame)
    setupView()
}

required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
    setupView()
}

private func setupView() {
    // view and constraint setup
}
}
```

3. A stack view takes care of the horizontal layout for us so let's add one as a private property of the view. We use the `spacing` property to configure the stack view:

```
var spacing: CGFloat = 16.0 {
    didSet {
        stackView.spacing = spacing
    }
}

private lazy var stackView: UIStackView = {
    let stackView = UIStackView()
    stackView.translatesAutoresizingMaskIntoConstraints = false
    stackView.spacing = spacing
    stackView.alignment = .center
    return stackView
}()
```

4. The view setup takes care of adding the stack view to the view hierarchy. I pinned the stack view to the top and bottom of the custom view and centered it horizontally:

```
private func setupView() {
    addSubview(stackView)
    NSLayoutConstraint.activate([
        stackView.centerXAnchor.constraint(equalTo:
            centerXAnchor),
        stackView.topAnchor.constraint(equalTo: topAnchor),
        stackView.bottomAnchor.constraint(equalTo:
            bottomAnchor)
    ])
}
```

```
}
```

Note that we have not fixed the width of the stack view. We'll see how to keep the width of the stack view within the bounds of the superview later.

5. To add views to the stack view, we need to implement the `show` method:

```
func show(_ items: [UIView]) {
    stackView.arrangedSubviews.forEach {
        $0.removeFromSuperview()
    }
    items.forEach {
        stackView.addArrangedSubview($0)
    }
}
```

This replaces any views already in the stack view with the new set of views.

6. In the view controller we set up our preview view, constraining it to the leading and trailing margins of the superview:

```
class PreviewViewController: UIViewController {
    private lazy var previewPane: PreviewPane = {
        let view = PreviewPane()
        view.translatesAutoresizingMaskIntoConstraints = false
        return view
    }()

    override func viewDidLoad() {
        super.viewDidLoad()
        setupView()
        refreshRandomViews()
    }

    private func setupView() {
        view.backgroundColor = .yellow
        view.addSubview(previewPane)

        let guide = view.layoutMarginsGuide
        NSLayoutConstraint.activate([
            previewPane.leadingAnchor.constraint(equalTo:
                guide.leadingAnchor),
            previewPane.centerYAnchor.constraint(equalTo:
                guide.centerYAnchor),
        ])
    }
}
```

```
    previewPane.trailingAnchor.constraint(equalTo:  
guide.trailingAnchor)  
])  
}
```

7. The method to create the random views and show them in the preview pane:

```
private let itemCount = 10  
  
private func refreshRandomViews() {  
    var views = [UIView]()  
    for count in 0..<itemCount {  
        let view = UILabel()  
        view.text = "\((count)"  
        let size = CGFloat(Float(arc4random_uniform(64)) + 64.0  
        view.font = UIFont.systemFont(ofSize: size)  
        view.backgroundColor = (count % 2 == 0) ? .red :  
.green  
        views.append(view)  
    }  
    previewPane.show(views)  
}
```

8. This works, but we have done nothing to stop the horizontal stack view from growing beyond the bounds of the superview:



Let's use our knowledge of the layout engine to override the default layout. The trick is to let the layout engine do its layout pass to set the bounds of each view. We can then check the width of the stack view to see if it's too big or too small and add or remove views as needed.

9. Before we look at changing the layout we need a property to hold the extra views that we remove from the stack view:

```
private var overflow = [UIView]()
```

We should clear this when we get a new set of views to preview:

```
func show(_ items: [UIView]) {  
    // ...  
    overflow.removeAll()  
}
```

10. We need to override `layoutSubviews` in our `PreviewPane`:

```
override func layoutSubviews() {  
    super.layoutSubviews()  
    // we can modify subviews here  
}
```

After we call `super.layoutSubviews()` the layout engine has updated the bounds and center of our subviews based on the views and constraints.

11. Let's first deal with the situation where the stack view is wider than the bounds of the superview. We want to remove the last view in the stack view and then ask the layout engine to update the layout again:

```
while stackView.bounds.width > bounds.width,  
let extraView = stackView.arrangedSubviews.last {
```

We can remove the extra view with `removeFromSuperview()` so it's no longer part of the layout and insert it at the front of the `overflow` array, so it's the first view we add back when we have space:

```
extraView.removeFromSuperview()  
overflow.insert(extraView, at: 0)
```

The stack view now has a different set of constraints. We tell the layout engine by calling `updateConstraintsIfNeeded` so it uses them to update its internal model:

```
updateConstraintsIfNeeded()  
super.layoutSubviews()  
}
```

Then to have the layout engine immediately update the bounds of the stack view from the now updated layout model we again call `super.layoutSubviews()`.

In this way we loop through the views in the stack view removing them one at a time, asking the layout engine to recalculate the layout until the stack view fits in the bounds of the preview superview.

12. We handle adding the extra views back into the stack view in a similar way:

```
while let nextItem = overflow.first,  
    stackView.bounds.width + spacing +  
    nextItem.intrinsicContentSize.width <= bounds.width {  
    stackView.addArrangedSubview(nextItem)  
    overflow.remove(at: 0)  
    updateConstraintsIfNeeded()  
    super.layoutSubviews()  
}
```

If we have extra views we check the intrinsic content size of the first one to see if it fits in the stack view without overflowing the bounds of the preview view.

If there's space we add the view to the stack view, remove it from the overflow and again tell the layout engine to take into account the updated constraints before updating the bounds and center of our subviews.

Avoiding Layout Loops

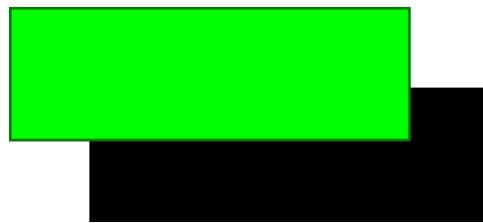
It's easy to create a layout loop when changing the view hierarchy and constraints in `viewWillLayoutSubviews` or `layoutSubviews`. To be safe don't change the layout of any views outside of your immediate subtree.

Be careful with `setNeedsUpdateConstraints` and `setNeedsLayout`. Both of these methods schedule another layout pass potentially creating a loop.

Alignment Rectangles

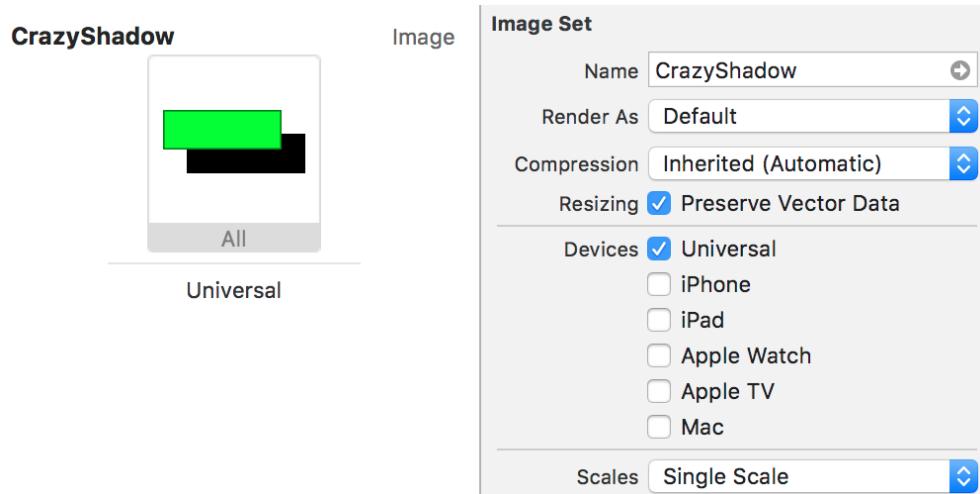
When the layout engine is aligning views, it doesn't use the view frame. Instead, it uses the view alignment rectangle. The alignment rectangle defaults to matching the view frame so you can usually ignore it.

What if you have a view that includes a drop shadow or glow effect or some other non-content addition such as a badge which can throw the default alignment off? This has become less common since Apple switched to a flat design style in iOS 7 but suppose I have an image with a crazy shadow that I want to center in its superview:

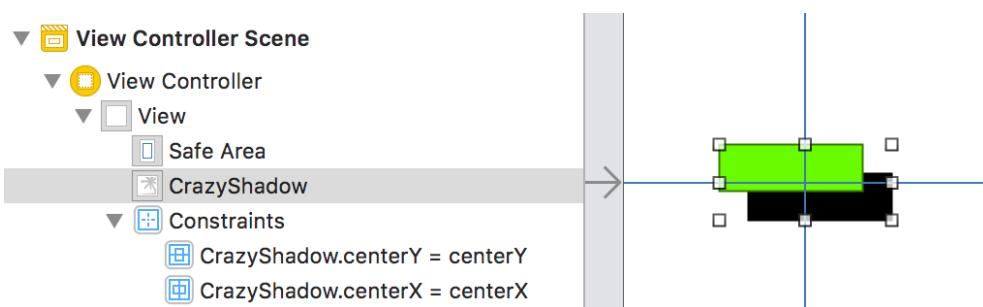


I'm using a pdf image that's 180 x 80 points in size including the shadow for this example. You can find this image in the sample code: [Alignment-v1](#).

1. Create a new Xcode project using the Single View App iOS template and add the pdf image file to the asset catalog as a Single Scale vector image:

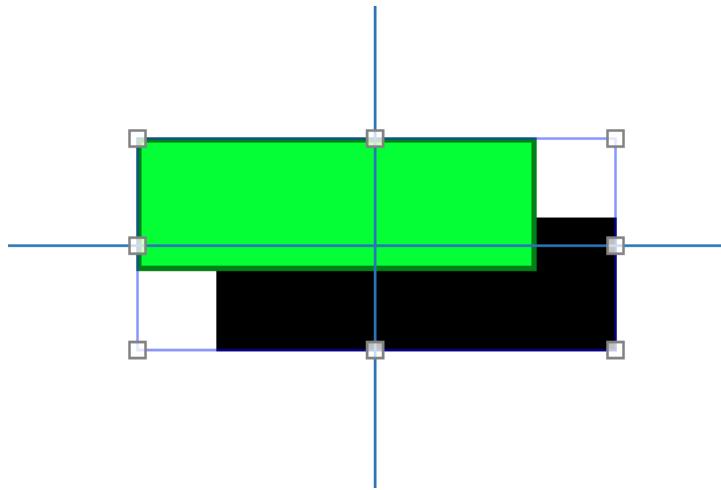


2. Drag a new image view onto the Interface Builder canvas and set the image name to the name of our image in the asset catalog. Add two constraints to center the image view horizontally and vertically:



3. Take a closer look at the Interface Builder canvas, and you can

already see the problem. I've centered the image view not the green rectangle:



Make sure you have turned on Editor > Canvas > Show Layout Rectangles in the Xcode menu to have Interface Builder show you the layout rectangles (unfortunately this doesn't seem to show any custom alignment rectangles we might set):



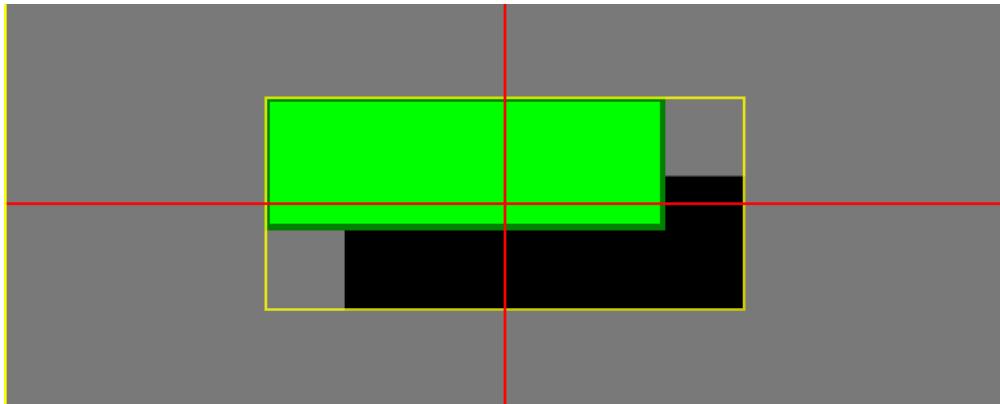
Debugging Alignment Rectangles

To see the alignment rectangles the layout engine is using you need to use a launch argument to turn on a view debug mode. In the Xcode scheme editor (Product > Scheme > Edit Scheme...) add the launch argument `-UIViewShowAlignmentRects` with a value of YES to the run action. The leading “-” is important:

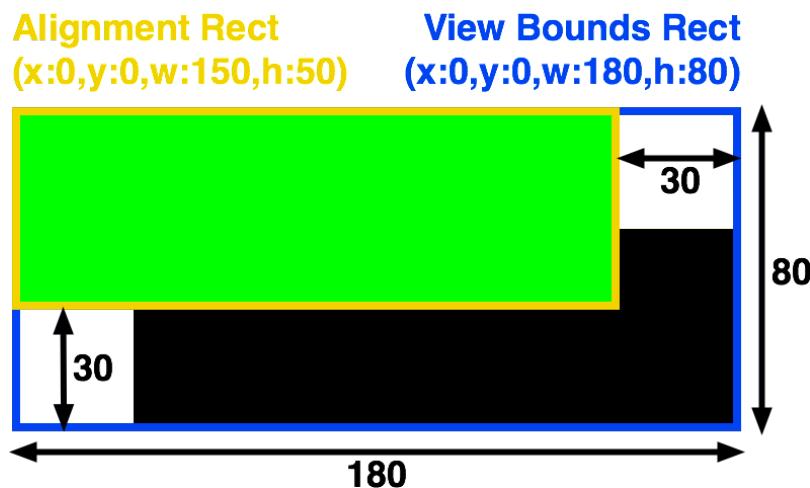


The alignment rectangle for the image view is now highlighted in yellow when we run. Note how it matches the view frame including the drop

shadow (I changed the background color and added red center lines to make it easier to see):



Auto Layout centers the yellow alignment rectangle in the view. It doesn't know we want the green box centered. To ignore the drop shadow we need a new alignment rect (shown in yellow below) that's inset from the image frame (shown in blue) on the bottom and right by 30 points.

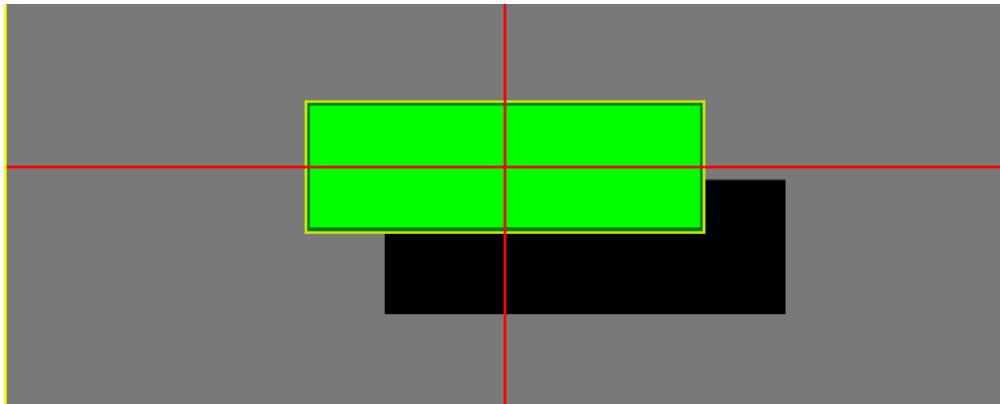


Alignment Rectangles In The Asset Catalog

If you're storing the image in the Xcode Asset Catalog you can directly change its alignment rectangle. Use the attributes inspector to add the insets (30 points in this case):

Alignment	0	0
Left		Top
Bottom	30	30
Right		

If we were using individual images instead of a PDF we would add insets of 30 pixels for the 1x, 60 pixels for the 2x and 90 pixels for the 3x image.



The yellow alignment rectangle now matches the green rectangle, and we have the view centered the way we want.

Alignment Rectangles In Code

If you're not using images stored in the asset catalog, perhaps because you create them at runtime, you can change the alignment rectangle in code. By default when you create an image, it has an alignment rectangle that matches its frame. In other words, it has a top, left, bottom and right inset of zero. Creating an image with a different alignment rectangle is a two-step process (see sample code: [Alignment-v2](#)):

1. Create the original image as usual.
2. Use the original image to create a new image with alignment rectangle insets.

Let's extend UIImageView with a convenience initializer that allows us to include the insets:

```
extension UIImageView {  
    convenience init(named name: String, top: CGFloat, left:  
        CGFloat, bottom: CGFloat, right: CGFloat) {  
        let insets = UIEdgeInsetsMake(top, left, bottom, right)  
        let originalImage = UIImage(named: name)  
        let insetImage =  
            originalImage?.withAlignmentRectInsets(insets)  
        self.init(image: insetImage)  
    }  
}
```

Using this to create our inset image view:

```
let imageView = UIImageView(named: "CrazyShadow", top: 0,  
    left: 0, bottom: 30, right: 30)
```

Key Points To Remember

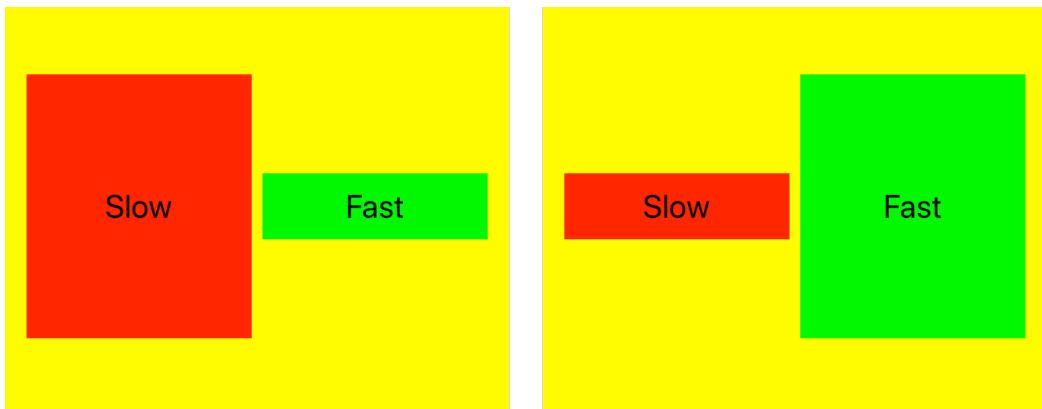
- Activating, changing or deactivating constraints doesn't directly update the frames of views. Instead, it updates the layout engine model and schedules a layout pass that runs later on the application run loop.
- There are two layout passes over the view hierarchy. The first pass allows you to update the constraints. The second pass updates the view layout by changing the size and position of views to match values from the layout engine model.
- Call `setNeedsLayout` to manually schedule an update layout pass. Call `layoutIfNeeded` to force an immediate update to the view frames from the model.
- You rarely need to use `updateConstraints` or its view controller companion `updateViewConstraints`. The system calls these methods frequently, so they are not the best choice to do your initial constraint setup. Use them when changing constraints in place is too slow.
- If you're not using Interface Builder to create your constraints try to create them once when creating your views. If you need to change your constraints at runtime, prefer to activate and deactivate constraints over adding and removing constraints.
- To animate constraint changes call `layoutIfNeeded` inside a view animation block.
- Override `layoutSubviews` to create more dynamic layouts.
- The layout engine uses the alignment rectangle not the view frame when positioning views.

Test Your Knowledge

See how well you understand the layout engine by solving these layout challenges:

Challenge 10.1 Animating Constraints

I centered two buttons vertically in the root view of a view controller. There's a standard amount of horizontal spacing between the buttons which are of equal width and stretched to fill the space between the leading and trailing margins:



The view controller can be in one of two states depending on which button the user taps:

- Tapping the red slow button should make it four times the height of the green fast button which should be at its natural size (shown on the left).
 - Tapping the green fast button should make it four times the height of the red slow button which should be at its natural size (shown on the right).
1. Build this layout either using Interface Builder or in code (or do both).
 2. The view should be in the “slow” state when it first appears with the red button four times the height of the green button.
 3. Animate the change between the two states. When the user taps the smaller button, it should grow in height as the taller button shrinks.

Hints And Tips

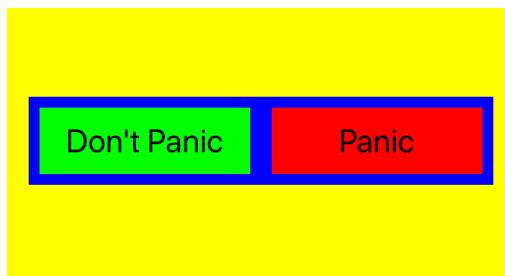
1. If you use Interface Builder create all your constraints in Interface Builder.
2. You cannot change the multiplier of a constraint after you have created it.

3. Create a height constraint for each button that makes it four times the height of the other button.
4. Only one of the height constraints should be active (installed in Interface Builder) at a time.
5. If using Interface Builder create outlets for the two height constraints so you can activate and deactivate them in the view controller.
6. To avoid conflicts deactivate the active height constraint first before activating the other constraint.
7. You can choose any animation effect you want. For example, a linear curve that lasts 0.25 seconds:

```
let animator = UIViewPropertyAnimator(duration: 0.25,  
curve: .linear) {  
    self.view.layoutIfNeeded()  
}  
animator.startAnimation()
```

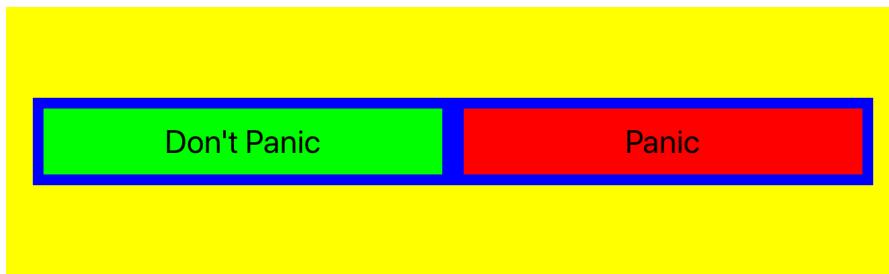
Challenge 10.2 Overriding The Layout Engine

In this layout, I have a blue container view with two equal-width buttons arranged horizontally. The buttons are inset from the edges of the container view by the standard 8 point margin and separated by 16 points of spacing:

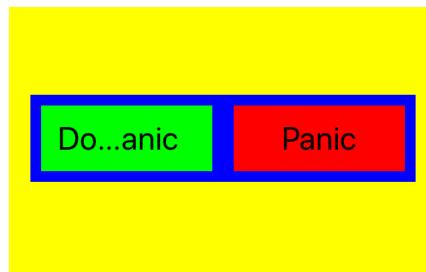


I centered the blue container view vertically and pinned it to the leading and trailing margins of the superview. The buttons are using a 24 point system font with 10 points of content inset on all sides. I want them at their natural height and stretched to fill the available width.

For example, here's how they look in landscape on an iPhone 8:



The challenge is to avoid truncating either button label when the available width is too small for the horizontal arrangement. For example, on an iPhone SE in portrait the longer button label gets truncated:



When there's not enough space to use the horizontal layout it should fallback to a vertical layout:



This can happen even on larger devices like the iPhone 8 if a localization increases the label size. For example, testing with the double-length pseudolanguage:



1. Build this layout to display the two buttons horizontally when it's possible to do so without truncating either label.

2. When there's not enough space switch the layout to a vertical arrangement.
3. Test your layout on a range of devices from the smallest iPhone to the largest iPad. Try the different slide over and split-screen modes of the iPad to make sure the layout adapts at runtime.

Hints And Tips

1. If you're not sure how to get started first build the horizontal layout of the two buttons without worrying about the truncation. When you have that working think about when and how to change to a vertical layout.
2. Using a stack view is probably the easiest way to do this but feel free to do it without a stack view if you prefer.
3. Create a custom view for the container view and buttons and override its `layoutSubviews` method. Don't forget `super.layoutSubviews`.
4. If you're using a stack view, switch the axis based on the horizontal width.
5. If you're not using a stack view, you need three sets of constraints: a base set of always active constraints and a horizontal and vertical set. Switch between the horizontal and vertical set, deactivating the active set before activating the new set.
6. The buttons should never be smaller than the intrinsic content size of the larger button to avoid truncation.

```
let minButtonWidth =  
    max(noPanicButton.intrinsicContentSize.width,  
        panicButton.intrinsicContentSize.width)
```

7. The minimum width between the margins needed for the horizontal layout is twice the minimum button width plus the 16 point spacing.

```
let minHorizontalWidth = minButtonWidth * 2 + spacing
```

8. You can get the width between the margins from the frame of the margin layout guide:

```
let marginWidth = layoutMarginsGuide.layoutFrame.width
```

9. Switch to the vertical layout when the `minHorizontalWidth` is greater than the `marginWidth` otherwise use the horizontal layout.

Chapter 10

Debugging When It Goes Wrong

Sometimes things go wrong. You think you have your constraints setup fine but when you build and run your views are missing, the console is a mass of unreadable debug logging, and in the worst case, your app may even get stuck in a loop.

In this chapter, we look at some of the tools and techniques you can use to figure out and fix your Auto Layout problems.

Unsatisfiable Constraints

There's nothing to stop you from adding constraints that conflict. When the layout engine is unable to find a working layout that fits all your constraints it does two things:

- Starts to break conflicting constraints until it can find a working solution.
- Logs the conflicting and broken constraints to the console.

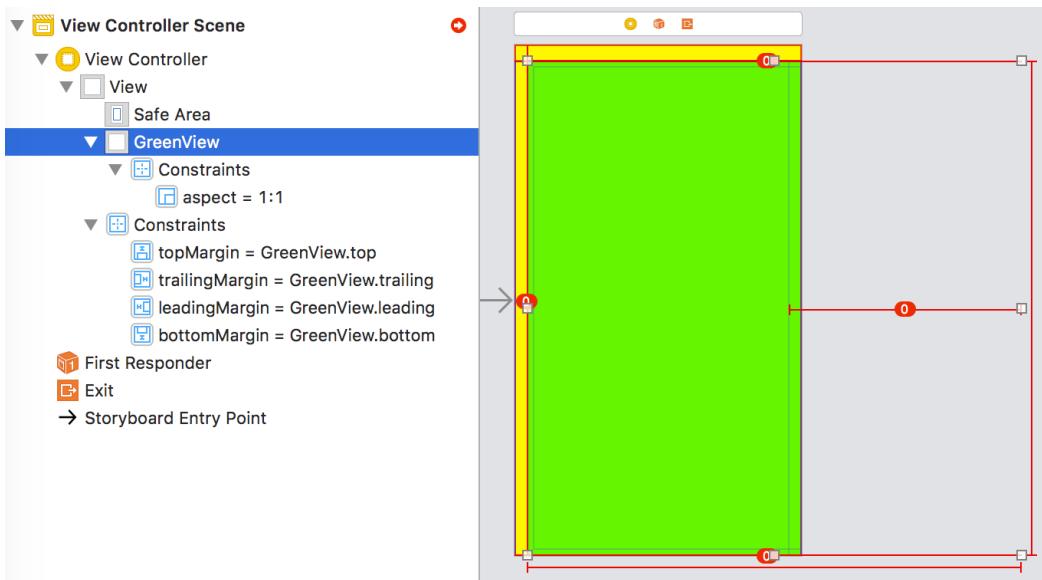
Your app doesn't crash, but your user interface may be a mess. Views may be wrongly sized or positioned or even be offscreen.



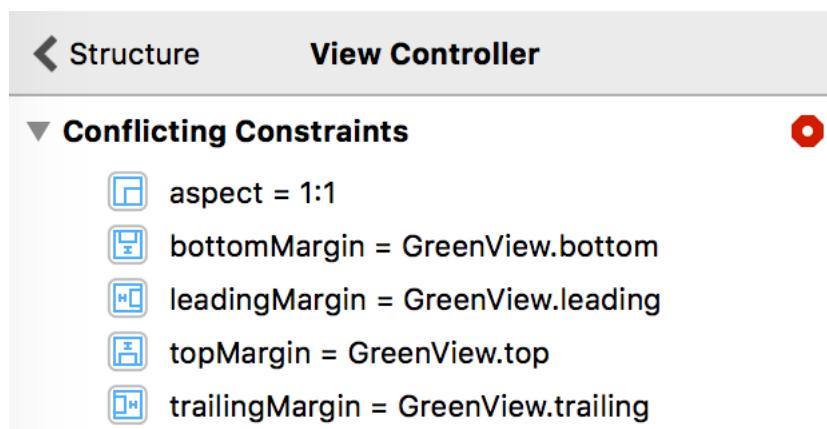
The constraint the layout engine breaks is not necessarily the cause of the problem though it's a good clue when starting your investigation.

Interface Builder Errors

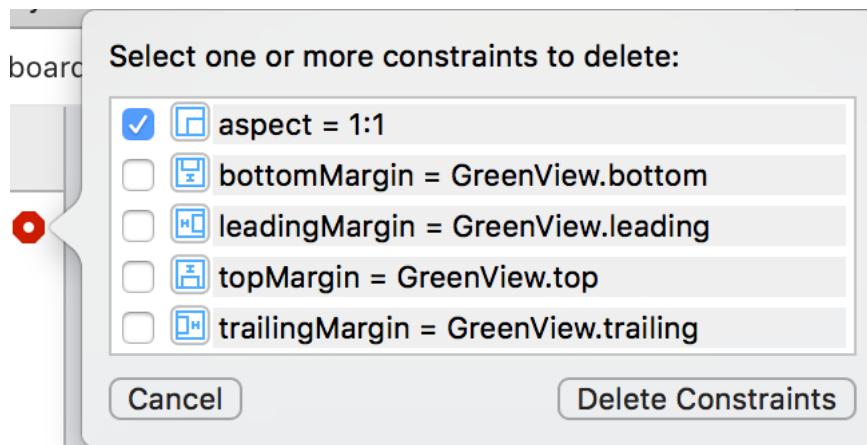
If you're creating your layout in Interface Builder, it does what it can to warn you about conflicting constraints as you add them. For example, I pinned a green view to the margins of a view controller's root view. If I also add a constraint to give it equal width and height (a 1:1 aspect ratio) I have created a conflict:



Clicking the red arrow in the document outline shows a list of the conflicting constraints:



Clicking again on the red error symbol allows you to choose which constraints you want to remove:



Interface Builder can catch the most obvious conflicts, but it cannot catch conflicts that happen at runtime or help when you create your layouts programmatically. Here's a small view controller that shows an OK button in the center of the screen (see sample code: [Conflict-v1](#)):

```
class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
        setupView()
    }

    private func setupView() {
        let okButton = UIButton(type: .system)
        okButton.setTitle("OK", for: .normal)
        view.addSubview(okButton)

        NSLayoutConstraint.activate([
            okButton.centerXAnchor.constraint(equalTo:
view.centerXAnchor),
            okButton.centerYAnchor.constraint(equalTo:
view.centerYAnchor)
        ])
    }
}
```

This builds without error but if you run it the button doesn't show up, and the layout engine logs a *lot* of text to the console:

```

2018-04-19 10:57:06.109610+0100 Conflict-v1[79339:9578302] [LayoutConstraints] Unable to
simultaneously satisfy constraints.
    Probably at least one of the constraints in the following list is one you don't
want.
    Try this:
        (1) look at each constraint and try to figure out which you don't expect;
        (2) find the code that added the unwanted constraint or constraints and fix it.
    (Note: If you're seeing NSAutoresizingMaskLayoutConstraints that you don't
understand, refer to the documentation for the UIView property
translatesAutoresizingMaskIntoConstraints)
(
    "<NSAutoresizingMaskLayoutConstraint:0x60000028b590 h=-& v=-& UIButton:
0x7fe238609dd0'OK'.midY == 0   (active)>",
    "<NSLayoutConstraint:0x60c000284f60 UIButton:0x7fe238609dd0'OK'.centerY == UIView:
0x7fe238613d70.centerY   (active)>",
    "<NSLayoutConstraint:0x60000028c260 'UIView-Encapsulated-Layout-Height' UIView:
0x7fe238613d70.height == 667   (active)>",
    "<NSAutoresizingMaskLayoutConstraint:0x60000028c120 h=-&- v=-&- 'UIView-
Encapsulated-Layout-Top' UIView:0x7fe238613d70.minY == 0   (active, names: '|':UIWindow:
0x7fe238558780 )>"
)

Will attempt to recover by breaking constraint
<NSLayoutConstraint:0x60c000284f60 UIButton:0x7fe238609dd0'OK'.centerY == UIView:
0x7fe238613d70.centerY   (active)>

Make a symbolic breakpoint at UIViewAlertForUnsatisfiableConstraints to catch this in
the debugger.
The methods in the UIConstraintBasedLayoutDebugging category on UIView listed in <UIKit/
UIView.h> may also be helpful.

```

Reading The Log

The log is not easy to read. The first line tells us what sort of a layout problem we have:

Unable to simultaneously satisfy constraints.

The log then shows the conflicting horizontal constraints, followed by the constraint the layout engine chose to break:

Will attempt to recover by breaking constraint
<NSLayoutConstraint:0x60c000284e20
UIButton:0x7fe238609dd0'OK'.centerX ==
UIView:0x7fe238613d70.centerX (active)>

This is our centerX constraint for the button. This pattern repeats for the vertical constraints with the layout engine deciding to break the centerY constraint for the button:

Will attempt to recover by breaking constraint
<NSLayoutConstraint:0x60c000284f60
UIButton:0x7fe238609dd0'OK'.centerY ==
UIView:0x7fe238613d70.centerY (active)>

We only added two constraints and the layout engine decided to break both of them because they were conflicting with other constraints! What

are those other constraints? The list of conflicting constraints gives us a big clue:

```
"<NSAutoresizingMaskLayoutConstraint:0x60000028b590 h=--&  
v=--& UIButton:0x7fe238609dd0'OK'.midY == 0    (active)>",
```

This is an autoresizing mask. The syntax `h=--&` and `v=--&` tells us the horizontal and vertical mask values. The `-` indicates fixed and `&` is flexible so this reads as:

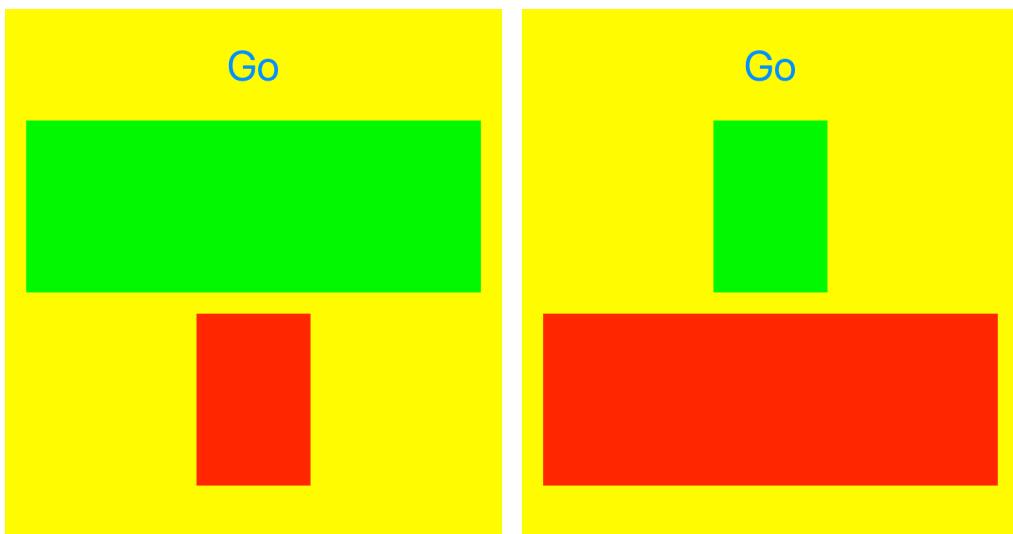
- `h=--&`: fixed leading edge, fixed width, flexible trailing edge
- `v=--&`: fixed top edge, fixed height, flexible bottom edge

This is the default autoresizing mask for an object. The layout engine has translated the autoresizing mask for the button into constraints which conflict with my center constraints. I forgot the golden rule to disable the automatic translation when creating the button! Disable the translation and the layout works as expected:

```
okButton.translatesAutoresizingMaskIntoConstraints = false
```

Setting A Breakpoint

Let's see another example (see sample code: [Conflict-v2](#)). This time I have a layout with two views and a button. When the layout loads the green view has a greater width than the red view (shown on the left). When the user taps the button the widths should switch (shown on the right):



Unfortunately, after tapping the button, both views disappear, and we

get the usual warnings dumped to the console:

```
[LayoutConstraints] Unable to simultaneously satisfy constraints.  
Probably at least one of the constraints in the following list is one  
you don't want.  
Try this:
```

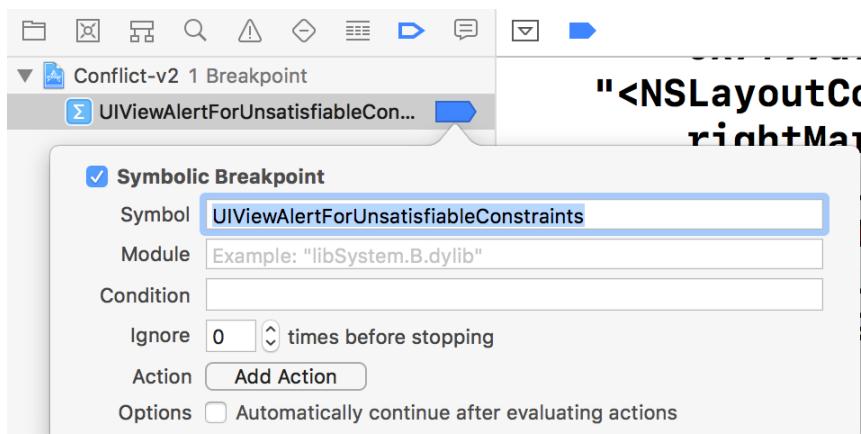
Let's skip over the list of conflicting constraints and look at the hints at the end of the log:

```
Make a symbolic breakpoint at UIViewControllerAnimatedConstraints to  
catch this in the debugger.  
The methods in the UIConstraintBasedLayoutDebugging category on UIView  
listed in <UIKitCore/UIView.h> may also be helpful.
```

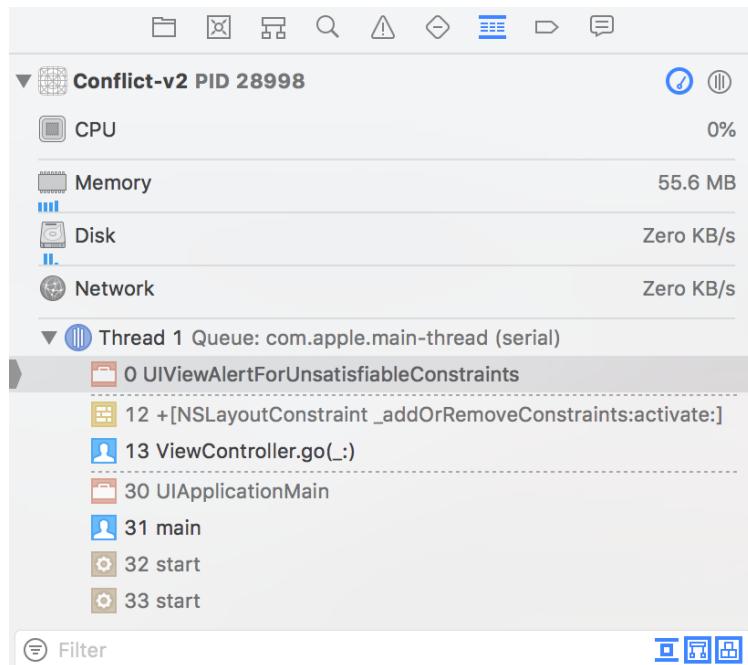
Let's try the first suggestion and add the symbolic breakpoint:

UIViewControllerAnimatedConstraints

Use the breakpoint navigator or Debug > Breakpoints > Create Symbolic Breakpoint to add the breakpoint:



The debugger now breaks when the layout engine detects an unsatisfiable constraint. The debug navigator shows the result when I tap the button:



I've turned on the filters at the bottom-right corner of the debug navigator to reduce the clutter. It shows the method `go(_:_)` in our view controller is activating constraints immediately before the layout engine detects the problem. Clicking on the method in the navigator takes us to the code:

```
@objc private func go(_ sender: UIButton) {
    NSLayoutConstraint.activate(redConstraints)
}
```

This is the target-action method for the button. The red constraints are what make the red view wider than the green view. What I'm missing is anything to deactivate the existing constraints:

```
@objc private func go(_ sender: UIButton) {
    NSLayoutConstraint.deactivate(greenConstraints)
    NSLayoutConstraint.activate(redConstraints)
}
```

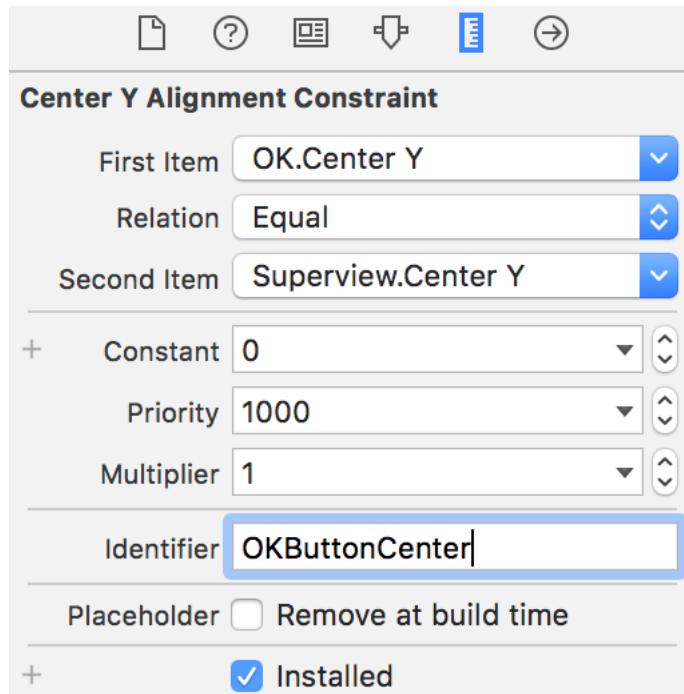
Deactivating the existing width constraints solves the problem. Setting the breakpoint takes you quickly to the area of code that's adding the conflicting constraints.

Adding Identifiers To Views And Constraints

The layout engine logs a considerable amount of detail when it has a problem, mixing our views and constraints in with those created by the

system. It can help to add identifiers to your constraints and views, so they stand out in the log.

All constraints have an `identifier` property that you can set in code or in Interface Builder:

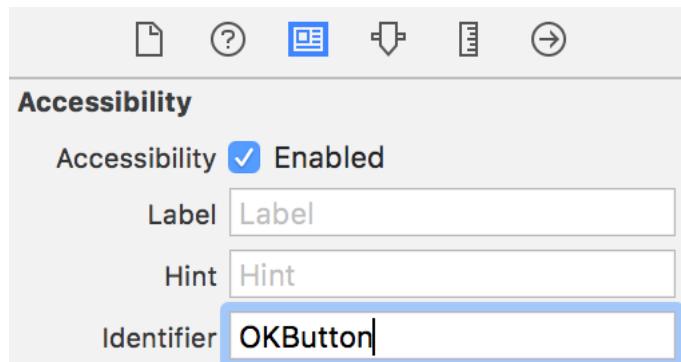


```
okCenterYConstraint.identifier = "OKButtonCenter"
```

When working with batches of constraints, I like to set the same identifier for each logical group of constraints. So for our button example, I might do it like this:

```
let centerConstraints = [
    okButton.centerXAnchor.constraint(equalTo:
        view.centerXAnchor),
    okButton.centerYAnchor.constraint(equalTo:
        view.centerYAnchor)
]
centerConstraints.forEach {
    $0.identifier = "OKCenter"
}
NSLayoutConstraint.activate(centerConstraints)
```

All views and layout guides have an accessibility identifier (don't confuse this with the accessibility label that VoiceOver reads):



With those two changes it becomes easier to spot our views and constraints in the log:

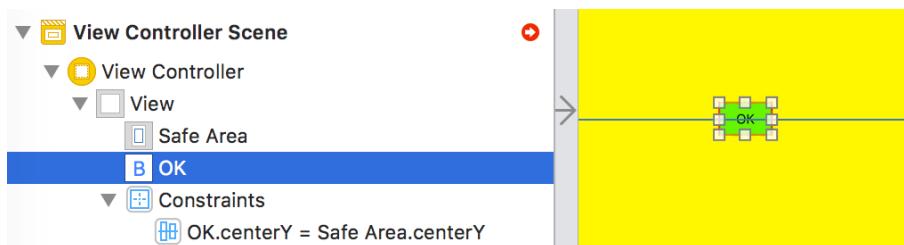
```
<NSLayoutConstraint:0x604000282a30 'OKCenter' OKButton.centerY
== RootView.centerY (active, names: okButton:0x7ff3d7f08020,
RootView:0x7ff3d7f0e9f0 )>
```

Ambiguous Layouts

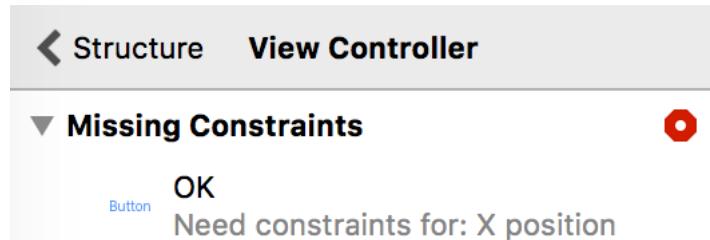
When the layout engine can find more than one solution that fits your constraints you have an ambiguous layout. There are some common causes:

- Missing constraints: You didn't add enough constraints to fix the size and position of every view in the layout.
- Conflicting Priorities: The layout engine needs to stretch or squeeze a view, but the content-hugging or compression-resistance priorities of the views are the same.

Interface Builder usually shows you warnings and errors for missing and conflicting priorities as you build your layout. For example, here's a button that I wanted to center in its superview. I added the constraint to center it vertically, but I'm missing the horizontal constraint:

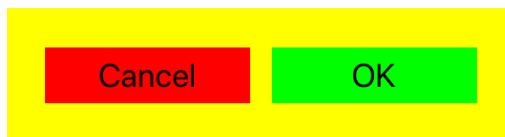


Interface Builder has marked the leading and trailing edges of the button in red in the canvas. There's also a red error indicator in the document outline. Clicking the error indicator shows the full details. We are missing a constraint to fix the x-position of the button:



If you're not using Interface Builder and build your layout in code there are no compile-time warnings or errors about ambiguous layouts. The first sign of a problem is often a missing or misplaced view when you run your app.

Here's a layout with two equal sized buttons that I want to build (see sample code: [Missing-v1](#)):



I decided to build this layout by placing two buttons in a stack view in a custom subview. I'll skip the details here, but this is what I see when I build and run:



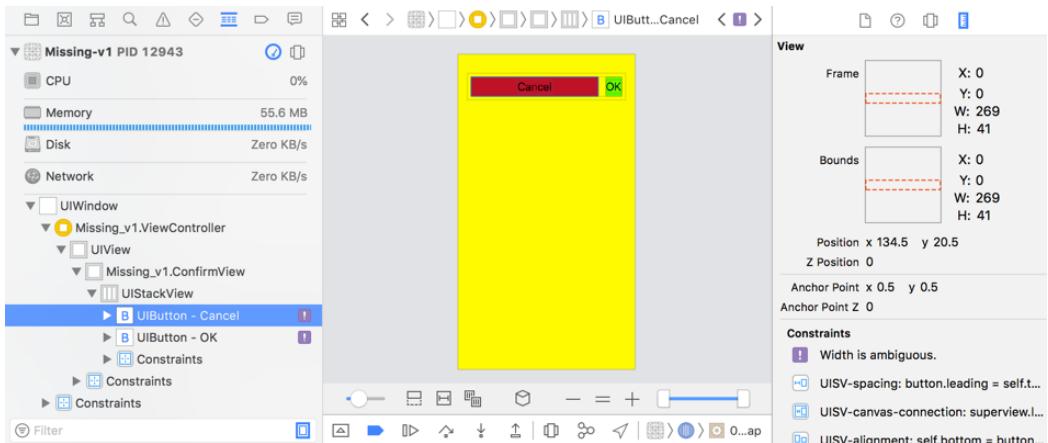
Not what I expected. Let's see if we can debug what's going on.

Using The View Debugger

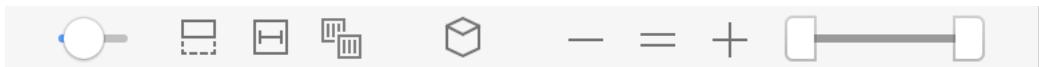
When you're having problems with layouts, the Xcode view debugger is usually a good place to start. Run your app on a device or in the simulator and navigate to the problem layout. Then open the view debugger using the control on the debugger toolbar or from the Xcode Debug menu (:Debug > View Debugging > Capture View Hierarchy):



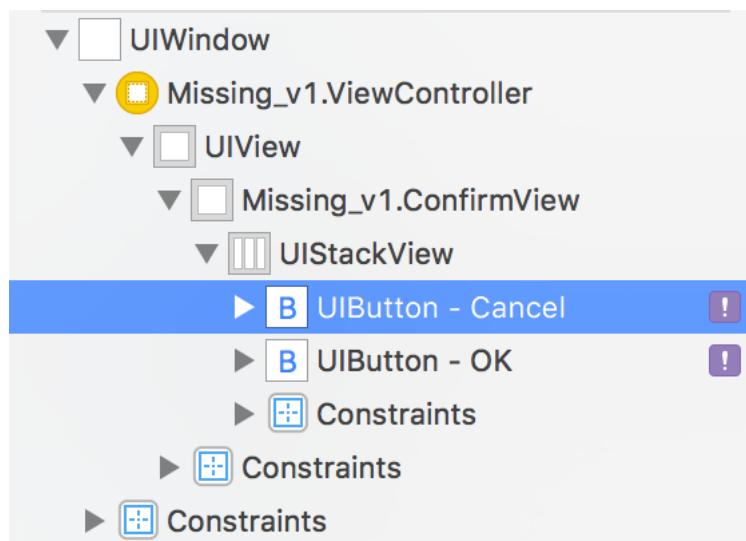
The view debugger pauses your app and shows an exploded 3D-style view of the layout in the canvas. The navigator shows the view hierarchy, and you can inspect any view or constraint with the size and object inspectors:



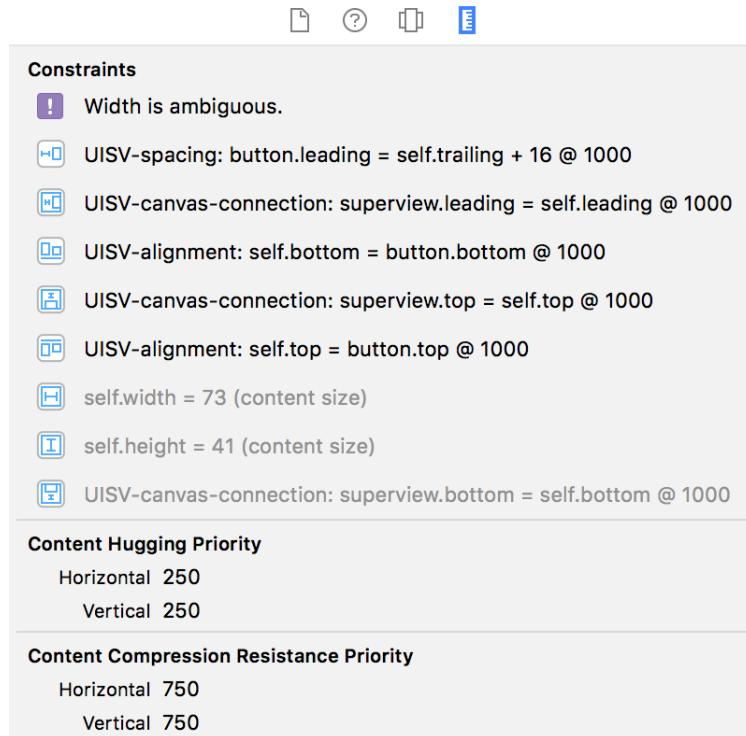
You can rotate and zoom the canvas or use the toolbar below the canvas to filter what you see:



My favorite feature of the view debugger is that it tells you if any views in the view hierarchy have an ambiguous layout. In this case, it has marked both buttons with a purple indicator:



If you click on the cancel button the Size Inspector shows the view frame, bounds, the constraints involving the view and the content priorities:



The Size Inspector makes it clear that the width for the button is ambiguous and lists the constraints involving the view. Only the constraints in bold are affecting the layout of the button. A lot easier to understand than the console log!

I'm using a stack view for the layout so it should be no surprise it has created most of the constraints. The Apple engineers did a great job of labeling their constraints with the UISV prefix. The other two constraints are for the intrinsic content size of the button.

So what's going wrong? Focus on the horizontal constraints. Notice that this button (`self`) has a trailing spacing constraint to the leading edge of the other button and a leading constraint to the leading edge of the superview (the stack view):

```
UISV-spacing: button.leading = self.trailing + 16 @ 1000  
UISV-canvas-connection: superview.leading = self.leading @  
1000
```

The OK button is pinned to the trailing edge of the stack view. What's missing is anything that sets the width of the two buttons. I want the stack view to give the buttons equal width. Here's the code I used to create the stack view:

```
let stackView = UIStackView(arrangedSubviews: [cancelButton,  
    okButton])  
stackView.translatesAutoresizingMaskIntoConstraints = false  
stackView.spacing = 16.0
```

What did I forget? The default distribution for a stack view is `.fill`. To make my two buttons have equal width I need `.fillEqually`. Adding the missing configuration fixes the problem:

```
stackView.distribution = .fillEqually
```

Private Debug Methods

The view debugger is generally the best way to investigate layout problems. Sometimes though you may prefer to work from the command-line in the debugger or you may be prototyping a layout in a playground where the debugger is not available.

There are some private methods and properties you can use in your code or from the debugger that show much the same information as the view debugger.



These are private methods and properties intended for debugging. Don't use any of them in a released version of your application.

Viewing Constraints That Affect The Layout

When debugging a problem, you're often not interested in seeing every constraint involving a view. Also unless you're using an aspect ratio constraint the horizontal and vertical constraints don't interact. You can reduce the clutter by looking at just those constraints impacting the layout along one axis.

The `constraintsAffectingLayout(for:axis:)` method returns the constraints impacting the layout of a view or layout guide for the specified axis. For example, suppose I have two constraints centering a button in the root view of a view controller:

```
okButton.centerXAnchor.constraint(equalTo:  
    view.centerXAnchor),
```

```
okButton.centerYAnchor.constraint(equalTo: view.centerYAnchor)
```

Assuming I set a breakpoint somewhere like `viewDidAppear`. We can see the constraints impacting the horizontal layout of the button in the debugger:

```
(lldb) po okButton.constraintsAffectingLayout(for:  
.horizontal)
```

This shows the constraints that impact the horizontal size and position of the button:

```
(lldb) po okButton.constraintsAffectingLayout(for: .horizontal)  
↳ 3 elements  
- 0 : <NSContentSizeLayoutConstraint:0x60000322b960 okButton.width == 84 Hug:  
    250 CompressionResistance:750 (active, names: okButton:0x7fa6957187e0 )>  
- 1 : <NSLayoutConstraint:0x600003539f40 okButton.centerX == rootView.centerX  
    (active, names: okButton:0x7fa6957187e0, rootView:0x7fa69570b180 )>  
- 2 : <NSLayoutConstraint:0x600003516620 'UIView-Encapsulated-Layout-Width'  
    rootView.width == 375 (active, names: rootView:0x7fa69570b180 )>
```

Let's look at each of the constraints:

0. The intrinsic content size of the button (84 points) sets the button width.
1. My constraint that centers the button in the root view.
2. The system added width constraint (375 points) for the root view.

As before, I added an accessibility identifier to the button and the root view to make them easier to spot in the log.

Checking For Ambiguous Layout

Both `UIView` and `UILayoutGuide` have a boolean instance property that returns true if their layout is ambiguous:

```
// true or false  
okButton.hasAmbiguousLayout
```

This doesn't tell you if a subview of the view has an ambiguous layout. You have to check the full view hierarchy yourself to be sure your layout is without problems:

```
override func viewDidAppear(_ animated: Bool) {  
    super.viewDidAppear(animated)  
    checkAmbiguousLayout(view)  
}
```

```
private func checkAmbiguousLayout(_ view: UIView) {
    for subview in view.subviews {
        _ = subview.hasAmbiguousLayout
        checkAmbiguousLayout(subview)
    }
}
```

When a view has an ambiguous layout checking `hasAmbiguousLayout` logs a description to the console:

```
View has an ambiguous layout. See "Auto Layout Guide: Ambiguous Layouts"
for help debugging. Displaying synopsis from invoking -[UIView
_autolayoutTrace] to provide additional detail.

*c cancelButton:0x7fdb7af12bc0- AMBIGUOUS LAYOUT for cancelButton.Width{id: 50}

Legend:
* - is laid out with auto layout
+ - is laid out manually, but is represented in the layout engine because
    translatesAutoresizingMaskIntoConstraints = YES
• - layout engine host
```

Tracing The View Hierarchy

Instead of calling `hasAmbiguousLayout` on each subview you can use `_autolayoutTrace()`. This dumps the view hierarchy starting from the top window and marks any views with an ambiguous layout in the output.

The only problem is that it's not directly available from Swift. As a workaround you can use `value(forKey:)` in your code:

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    print(view.value(forKey: "_autolayoutTrace")!)
}
```

This prints a detailed dump of the view hierarchy:

```
*UIWindow:0x7fdceec52710 - AMBIGUOUS LAYOUT
|   *UIView:0x7fdcfc110c860
|   |   *UILayoutGuide: 0x6000035180e0 - "UILayoutMarginsGuide", layoutFrame = {{20, 40},
|   |   {335, 607}}, owningView = <UIView: 0x7fdcfc110c860; frame = (0 0; 375 667); autoresize = W+H;
|   |   layer = <CALayer: 0x600000c453a0>>
|   |   *Missing_v1.ConfirmView:0x7fdceef0ba90
|   |   |   *UILayoutGuide: 0x600003569340 - "UILayoutMarginsGuide", layoutFrame = {{8, 8},
|   |   |   {319, 41}}, owningView = <Missing_v1.ConfirmView: 0x7fdceef0ba90; frame = (20 40; 335 57);
|   |   |   layer = <CALayer: 0x600000c5be40>>
|   |   |   *UIStackView:0x7fdceef177f0
|   |   |   |   *cancelButton:0x7fdceef16980- AMBIGUOUS LAYOUT for cancelButton.Width{id: 50}
|   |   |   |   |   UIButtonLabel:0x7fdceef16ca0'Cancel'
|   |   |   |   *okButton:0x7fdceef116a0- AMBIGUOUS LAYOUT for okButton.minX{id: 61},
|   |   |   |   okButton.Width{id: 56}
|   |   |   |   |   UIButtonLabel:0x7fdceef134c0'OK'
```

In the debugger, you can also fall back to using the Objective-C method. Since it doesn't matter where in the view hierarchy we call the method we can start from the key window of the application:

```
expr -l objc -o -- [[[UIApplication sharedApplication]
    keyWindow] _autolayoutTrace]
```

The `-o` prints the Objective-C description of the object (like `po`). You can shorten this even further if you use the private `keyWindow` class method of `UIWindow`:

```
expr -l objc -o -- [[UIWindow keyWindow] _autolayoutTrace]
```

To avoid typing this each time create a command alias in `.lldbinit` in your home directory:

```
command alias aldump expr -l objc -o -- [[UIWindow keyWindow]
    _autolayoutTrace]
```

Then anytime you want to check the view hierarchy in the debugger:

```
(lldb) aldump
$UIWindow:0x7fb63ae23150 - AMBIGUOUS LAYOUT
...
```

Exercising Ambiguity

If it's not already clear why a view has an ambiguous layout you can ask the layout engine to randomly change the frame of a view by calling `exerciseAmbiguityInLayout()` on the view. The layout engine chooses a different frame that still satisfies the constraints.

This can be tricky to use in practice. One way is to use a timer to repeatedly call `exerciseAmbiguityInLayout()` on any view that's ambiguous. Here's a Swift extension on `UIView` that does just that for us:

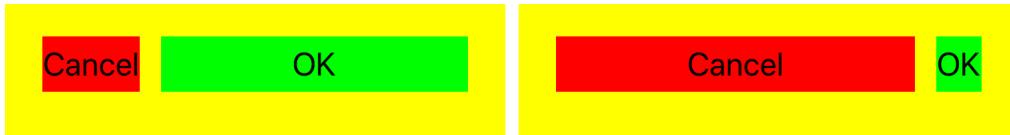
```
import UIKit
extension UIView {
    // From an original idea by Florian Kugler
    // https://www.objc.io/issues/3-views/advanced-auto-layout-
        toolbox/
    class func exerciseAmbiguity(_ view: UIView) {
        #if DEBUG
            if view.hasAmbiguousLayout {
```

```
    Timer.scheduledTimer(withTimeInterval: 0.5, repeats:  
true) { _ in  
    view.exerciseAmbiguityInLayout()  
}  
} else {  
    for subview in view.subviews {  
        UIView.exerciseAmbiguity(subview)  
    }  
}  
#endif  
}  
}
```

Note this is debug only code. We might call this from `viewDidAppear` after the layout engine has done its work:

```
override func viewDidAppear(_ animated: Bool) {  
    super.viewDidAppear(animated)  
    UIView.exerciseAmbiguity(view)  
}
```

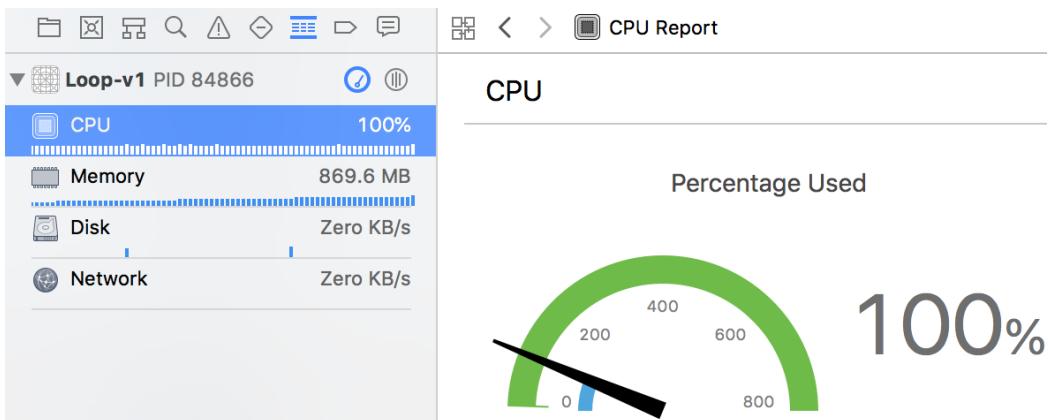
If you try this with the sample code (see [Missing-v1](#)) you'll see the widths of the two buttons bounce between different possible values:



Layout Loops

While much less common than the other layout problems we have seen in this chapter it's possible to get the layout engine into a loop. It's hard to miss when it happens as your user interface locks up, the device CPU utilization goes to 100%, and the memory consumption starts to grow:

(To recreate the problem see sample code: [Loop-v1](#)).

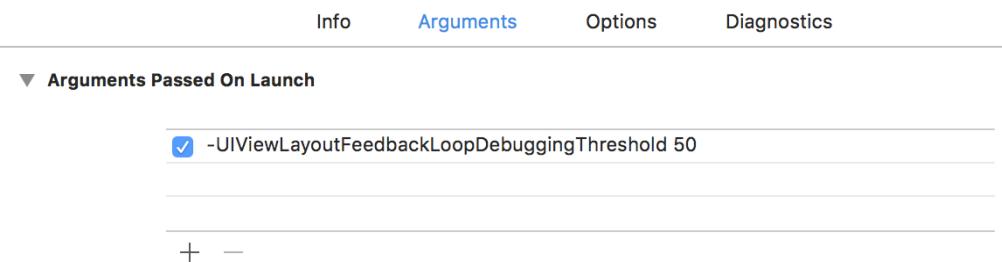


There's a launch argument you can add that turns on layout feedback loop debugging:

```
-UILayoutFeedbackLoopDebuggingThreshold 50
```

The layout engine keeps a count of the calls to `layoutSubviews` during a layout pass. When it exceeds the threshold, it aborts that App and logs the results. The threshold can be in the range 50 to 1000. Choose a value that's higher than the number of subviews in your layout, but 50 is a good starting point.

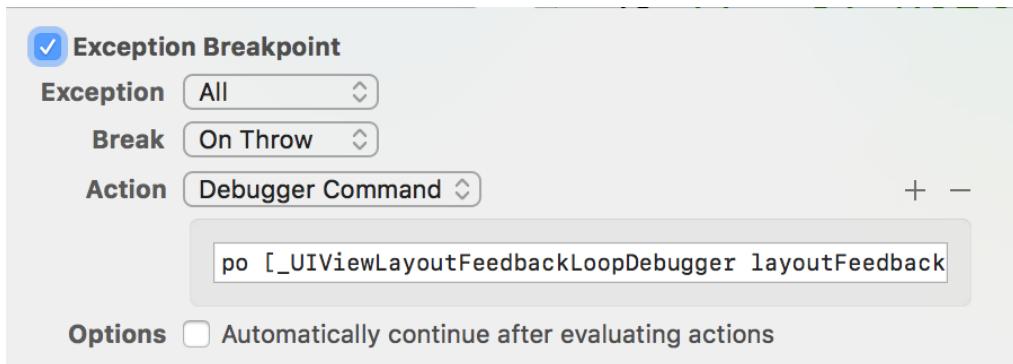
From the Xcode scheme editor (Product > Scheme > Edit Scheme...) add the launch argument to the run action:



To print the debug data in the debugger:

```
po [_UILayoutFeedbackLoopDebugger  
layoutFeedbackLoopDebugger]
```

You can also set an exception breakpoint and have it the layout feedback debug log:



The logs are verbose but here are the key excerpts that show us what's happening. First a warning that we have a `viewDidLayoutSubviews` method that's calling `setNeedsLayout` on a parent view dirtying the layout:

```
[LayoutLoop] >>>UPSTREAM LAYOUT DIRTYING<<<
About to send -setNeedsLayout to layer for
<UIView: 0x7fc47f407f90; f={{0, 0}, {375, 667}}}
> under -viewDidLayoutSubviews for
<UIView: 0x7fc47f407f90; f={{0, 0}, {375, 667}}>
```

Warnings about a layout loop in what turns out to be the root view of our view controller:

```
[LayoutLoop] Degenerate layout!
Layout feedback loop detected in subtree of
<UIView: 0x7fc47f407f90; frame = (0 0; 375 667); >.
```

The call stacks show that the culprit is `viewDidLayoutSubviews` in our view controller:

```
*** Call stacks where -setNeedsLayout is sent
      to the top-level view ***
2 Loop-v1 0x00000001010aa6f1
 _T07Loop_v114ViewControllerC21viewDidLayoutSubviewsyyF
```

The method in the view controller:

```
override func viewDidLayoutSubviews() {
    super.viewDidLayoutSubviews()
    contentView.layer.cornerRadius = contentView.bounds.width /
    25
    view.setNeedsLayout()
}
```

Calling `setNeedsLayout` after the view has finished laying out its subviews causes it to start again in an endless loop. Removing this line removes the layout loop.

Key Points To Remember

Some tips to remember when debugging problems with your layouts:

- Forgetting to disable translation of the autoresizing mask into constraints is a frequent reason for unsatisfiable constraint problems:

```
translatesAutoresizingMaskIntoConstraints = false
```

- Interface Builder can help find a lot of common problems, but it cannot warn you about runtime problems.
- Test your App on as many devices configurations as you can. Some problems only show up when you squeeze a view into the smallest iPhone screen or stretch it to fill the largest iPad Pro.
- Ambiguous layouts happen when you're missing constraints or have conflicting priorities. Use the view debugger to find which views are ambiguous.
- Add identifiers to your constraints and accessibility identifiers to your views to make them stand out in the logs.
- When you can't use the view debugger the private layout debug methods (`_autolayoutTrace`, `hasAmbiguousLayout`) can help.
- If your layout hangs and your App is consuming high CPU you may have created a layout loop. Add a launch argument to set a loop debugging threshold to help find the problem:

```
-UILayoutFeedbackLoopDebuggingThreshold 50
```

Chapter 11

Scroll Views And Auto Layout

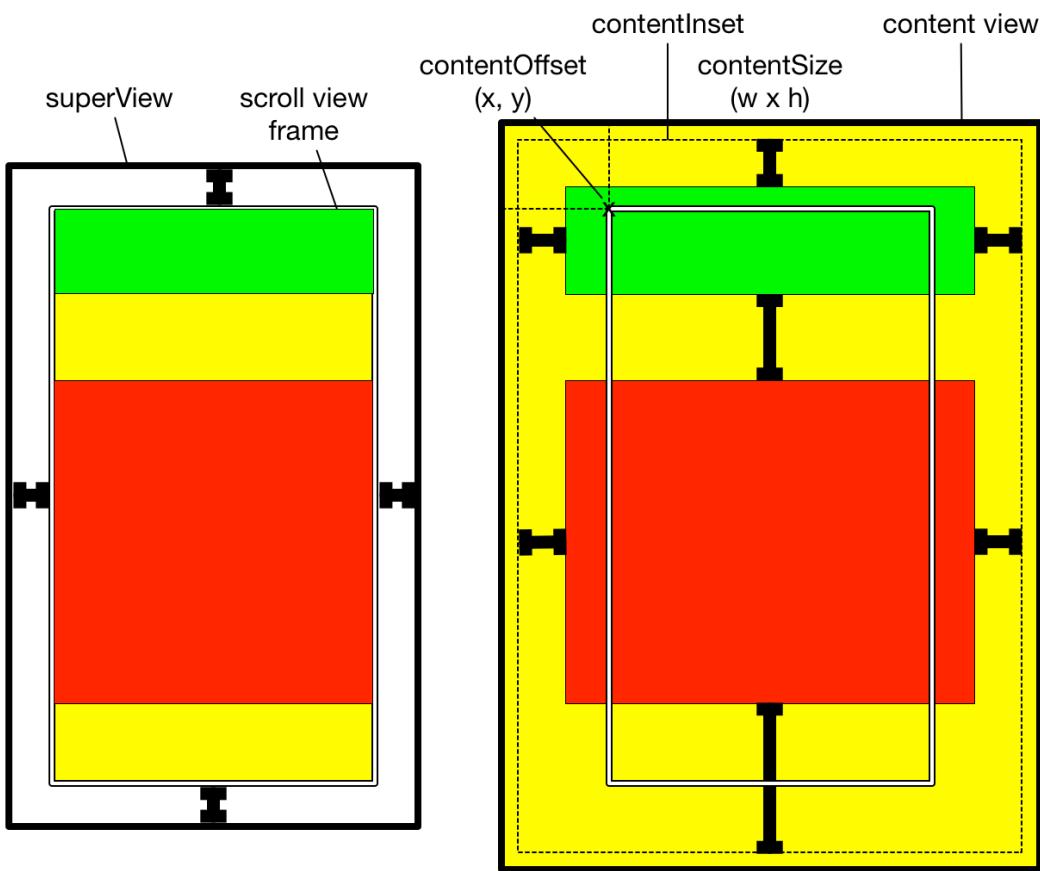
Creating a layout flexible enough to make the best use of the available space is not easy. The size of your content and the available space can both change dramatically at runtime depending on the device, orientation and user preferences such as the localization of preferred text size (we'll see examples of this when we look at [Dynamic Type](#)).

A scroll view helps you build flexible layouts by allowing the user to scroll over content that's bigger than the available space. Unfortunately using a scroll view with Auto Layout can be a confusing process. In this chapter you learn:

- How to set up your constraints when working with a scroll view.
- How to use a stack view (or other container view) with a scroll view.
- A more natural way to think about scroll view constraints with the frame and content layout guides that Apple introduced to scroll views in iOS 11.
- How to use the scroll view content insets so that the keyboard doesn't hide your content.

Creating Constraints For A Scroll View

Creating constraints for a scroll view can seem a little odd at first. You add your content as subviews of the scroll view. The scroll view moves over this potentially much larger content view clipping the content to show only what fits in its bounds:



The scroll view `contentOffset` property is a `CGPoint` setting the origin of the scroll view frame in relation to the origin of the content view. A `contentOffset` of `(0,0)` shows the top-left corner of the content.

The `contentInset` is a `UIEdgeInsets` that allows you to add extra padding between your content and the edge of the content view. We'll see how to use this when we look at [Managing The Keyboard](#). UIKit automatically adds an extra content inset to allow for the safe area. You can change this with the `contentInsetAdjustmentBehavior` property.

The `contentSize` property is a `CGSize` that sets the width and height of the content area controlling how far a scroll view can scroll. If the `contentSize` width is wider than the scroll view bounds, it can scroll horizontally. If the `contentSize` height is larger than the scroll view bounds, it can scroll vertically.

When creating constraints for a scroll view, I find it best to think about two groups of constraints. The first group set the size and position of the scroll view frame in relation to its superview. The second group constrains the content of the scroll view to fill and set the size of the content area.

The second group of constraints is where things can get confusing. A scroll view doesn't have a `contentView` property so how do you constrain the content area? The way Apple solved this was to have a scroll view treat constraints differently:

- A constraint between a scroll view and a view outside the scroll view constrains the frame of the scroll view.
- A constraint between a scroll view margin or edge and a subview constrains the subview to the content area of the scroll view.
- A height or width constraint between the scroll view and a subview constrains the subview with the scroll view's frame.

Notes:

- To fully constrain the content area you need a chain of constraints and views from the top to bottom and leading to trailing edges of the content area. If the views all have an intrinsic content size, this sets the content size of the scroll view. If not, you need extra constraints to set the width or height.
- Use an equal width or height constraint between the scroll view and the content to disable scrolling for that direction. For example, an equal width constraint between the content and the scroll view prevents horizontal scrolling. An equal height constraint prevents vertical scrolling.

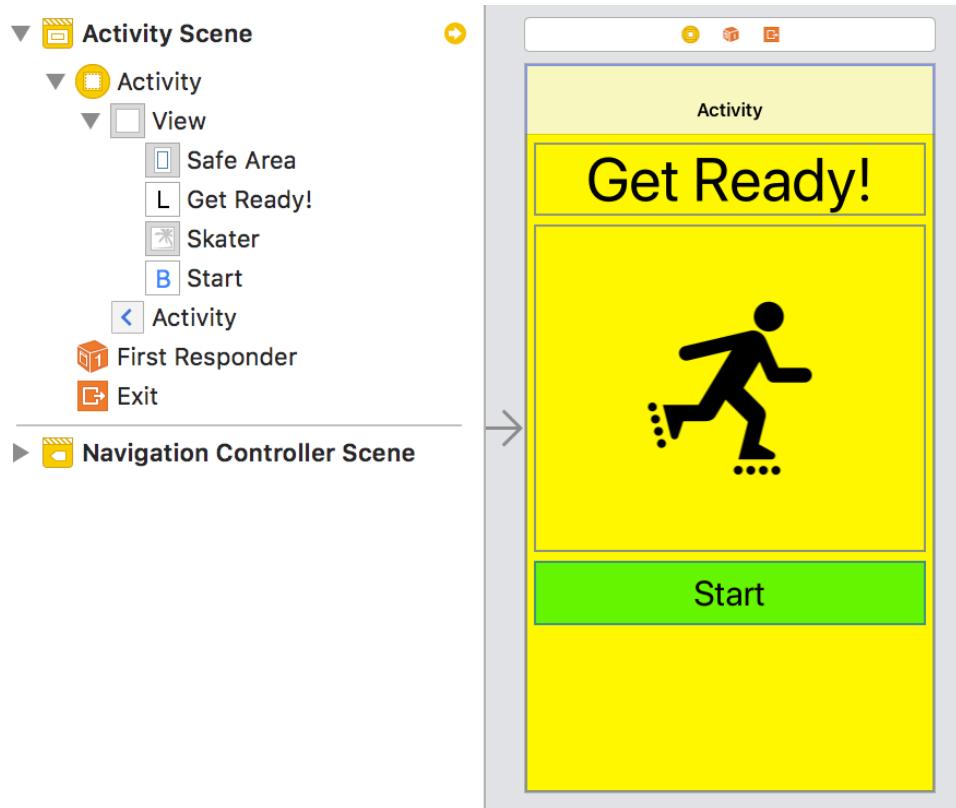
Scroll Views In Interface Builder

Let's see how this works in practice with a real layout shown here on an iPhone 8 in portrait and landscape:



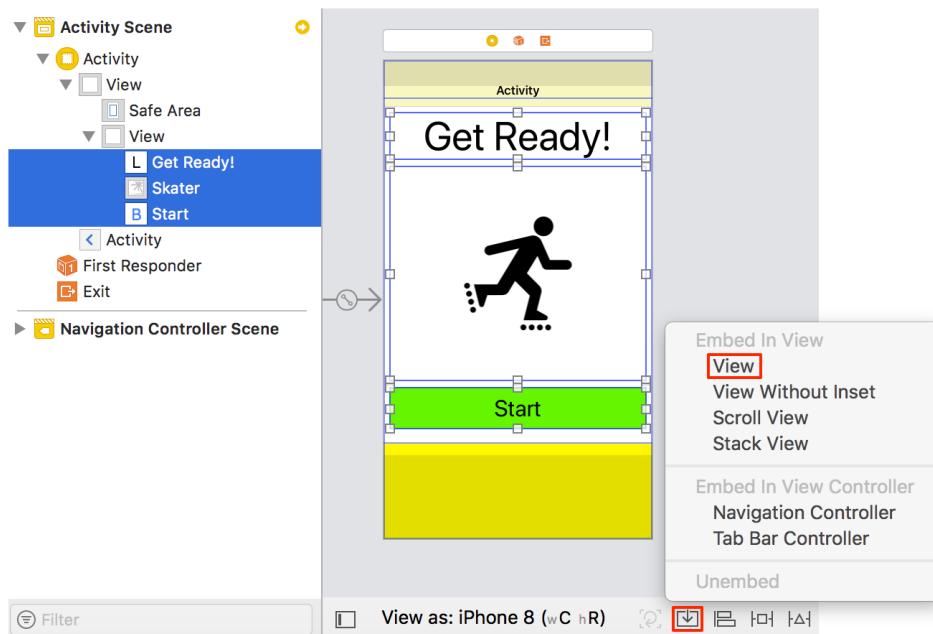
The layout has a label, image and button arranged in a vertical layout that needs to scroll when in landscape (see sample code: [ScrollView-v1](#)):

1. Starting with a view controller embedded in a navigation controller arrange a label, image view, and button in a vertical layout:



My label is center aligned and uses a 56 point system font. The image is a 300 x 300 point vector image. The button title uses a 32 point system font, and I added 10 point top and bottom content insets for some extra padding.

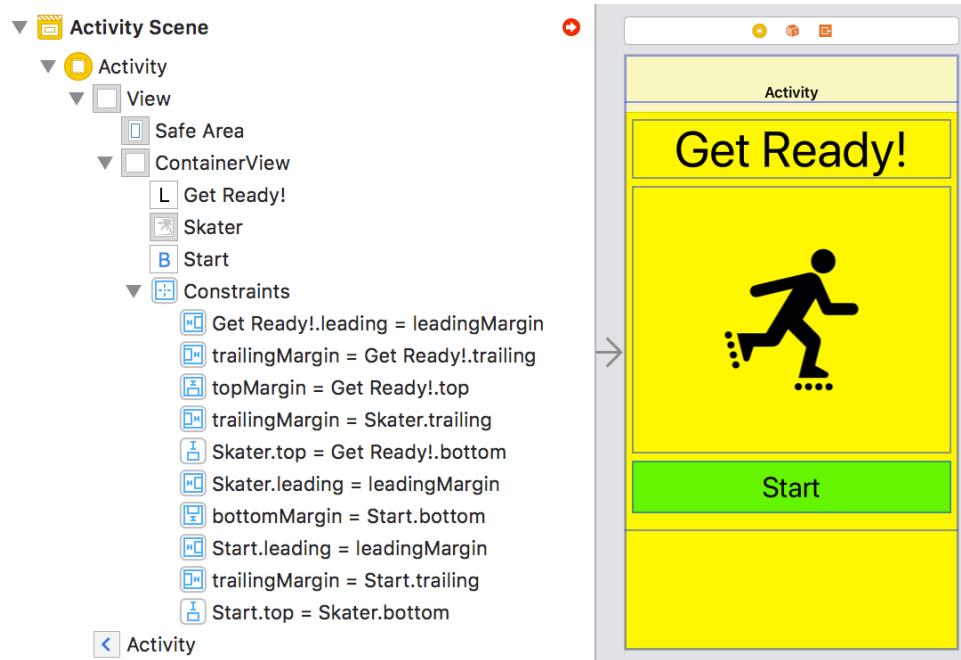
2. When working with a scroll view, I find it easiest to use a containing view for the scroll view content. Select the label, image view and button and then using the **[Embed In]** button embed them in a view:



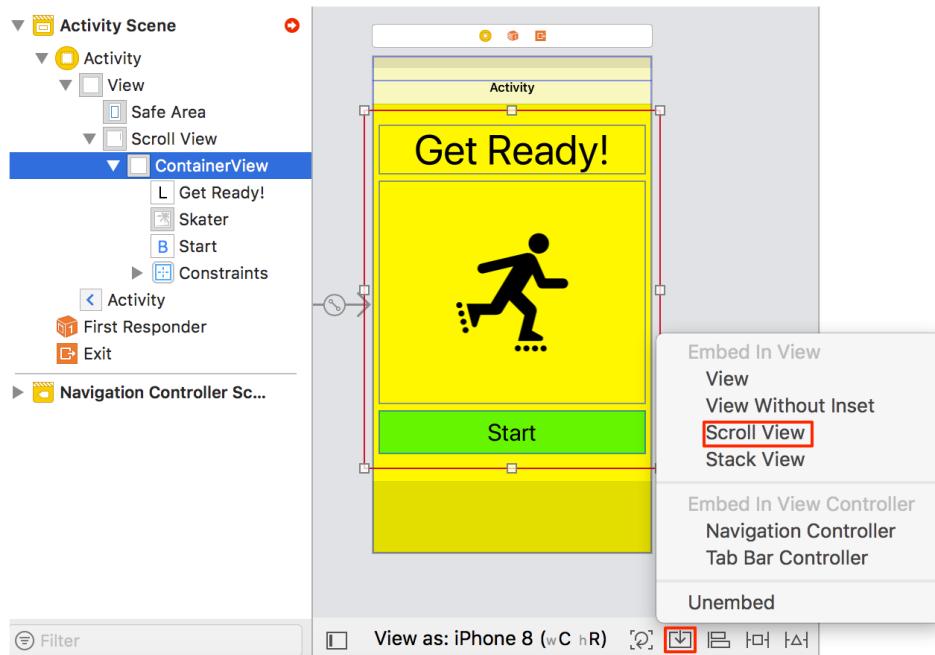
I like to change the label of the view in the document outline to something more meaningful like `ContainerView`.

3. Change the background color of the container view to match the root view and then add the constraints:

- Constrain the label to the leading, top and trailing margins of the container view.
- Constrain the image view to the leading and trailing margins of the container view and add standard vertical spacing to the label.
- Constrain the button to the leading, bottom and trailing margins of the container view and add standard vertical spacing to the image view.

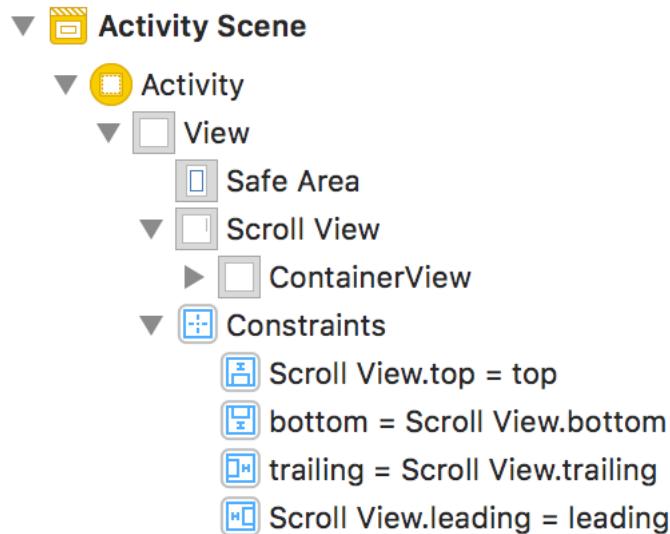


4. With the internal layout of the container view done we can embed it in a scroll view. Select the container view and use the **[Embed In]** button again but this time embed in a scroll view:

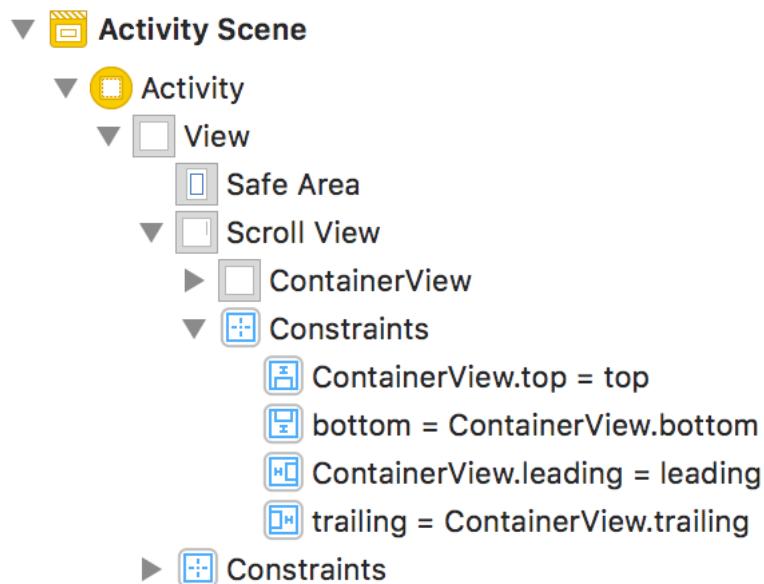


5. Now we can add the constraints for the scroll view. I want the scroll view to fill the root view so that our content scrolls behind the navigation bar. Add four constraints pinning it to the edges of the

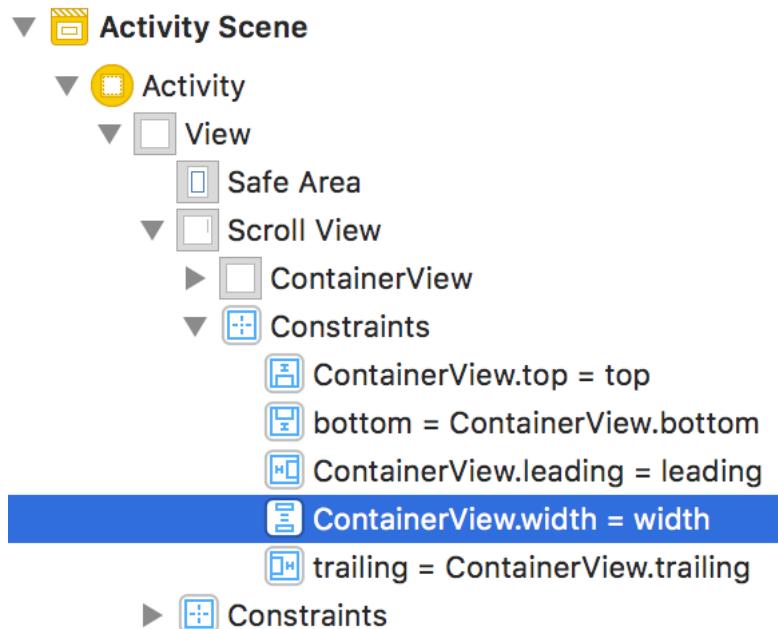
root view:



6. The second set of constraints between the scroll view and the container view set the content area. The content needs to fill the content area of the scroll view, so we need to constrain all four edges of our container view to the edges of the scroll view:



7. Almost done. What about the content size of the scroll view? I want the container view to fill the width of the root view and only scroll vertically. That needs an extra constraint to fix the container view width to the width of the scroll view:



Scroll Views In Code

Let's rebuild this layout in code (see sample code: [ScrollView-v2](#)):

1. I'll avoid repeating the setup of the content view since it follows the same approach we used with Interface Builder. I moved the label, image view, and button to a custom subview of `UIView` that matches the container view we used in Interface Builder. I create an instance of that view in my view controller:

```

private let activityView: ActivityView = {
    let view = ActivityView()
    view.translatesAutoresizingMaskIntoConstraints = false
    view.backgroundColor = .yellow
    return view
}()

```

2. I then build the scroll view and add the activity view as content to the scroll view:

```

private lazy var scrollView: UIScrollView = {
    let scrollView = UIScrollView()
    scrollView.translatesAutoresizingMaskIntoConstraints = false
    scrollView.addSubview(activityView)
    return scrollView
}()

```

3. Then from my `viewDidLoad` method, I call `setupView` to build the view hierarchy and setup constraints:

```
override func viewDidLoad() {
    super.viewDidLoad()
    view.backgroundColor = .yellow
    view.tintColor = .black
    title = NSLocalizedString("Activity", comment: "")
    setupView()
}

private func setupView() {
    view.addSubview(scrollView)
    NSLayoutConstraint.activate([
        // setup constraints
    ])
}
```

4. Let's first add the constraints to fix the scroll view frame to its superview by pinning it to each edge of the root view:

```
scrollView.leadingAnchor.constraint(equalTo:
    view.leadingAnchor),
scrollView.topAnchor.constraint(equalTo: view.topAnchor),
scrollView.trailingAnchor.constraint(equalTo:
    view.trailingAnchor),
scrollView.bottomAnchor.constraint(equalTo:
    view.bottomAnchor),
```

5. Then we add the constraints between the scroll view and the content:

```
scrollView.leadingAnchor.constraint(equalTo:
    activityView.leadingAnchor),
scrollView.topAnchor.constraint(equalTo:
    activityView.topAnchor),
scrollView.trailingAnchor.constraint(equalTo:
    activityView.trailingAnchor),
scrollView.bottomAnchor.constraint(equalTo:
    activityView.bottomAnchor),
```

6. Finally to make sure our content fills the width of the scroll view frame and to prevent any horizontal scrolling we add an equal width constraint:

```
scrollView.widthAnchor.constraint(equalTo:
    activityView.widthAnchor)
```

Container Views

Using a container view for your content helps simplify the scroll view setup and manage things like content margins. I used a subclass of `UIView` as a container for my content in this example, but you don't have to. A stack view is already a container view for content. Let's look next at how to create a scrollable stack view.

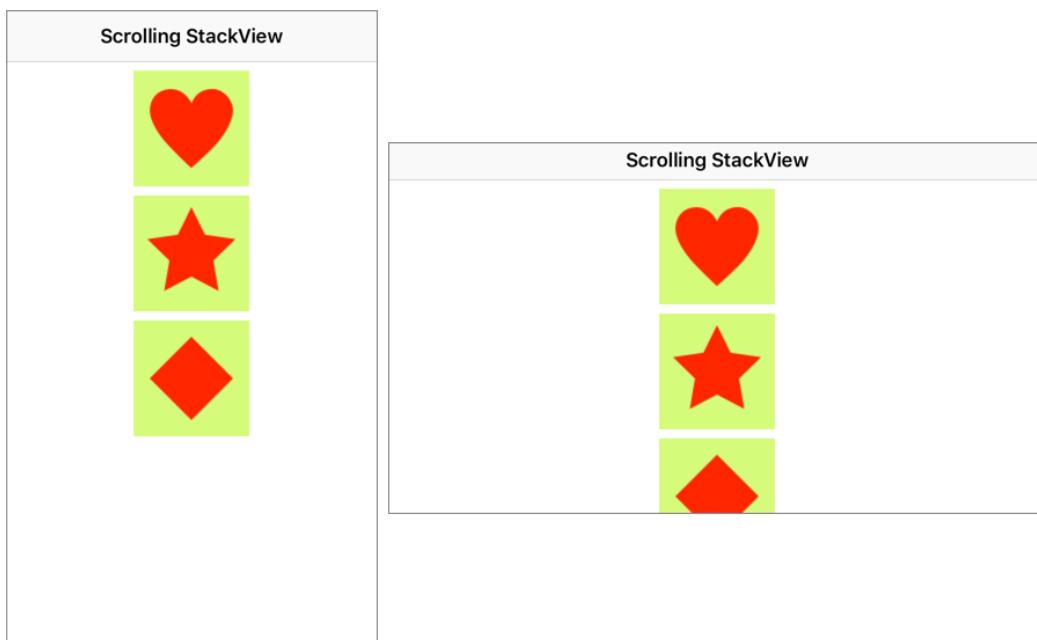
Scrolling A Stack View

A stack view is not a scroll view and will not automatically scroll its contents like a table or collection view. In the next few chapters, we'll see many examples where the contents of a stack view can grow too big for the available space. For those times when you need to scroll nothing is stopping you from embedding a stack view in a scroll view.



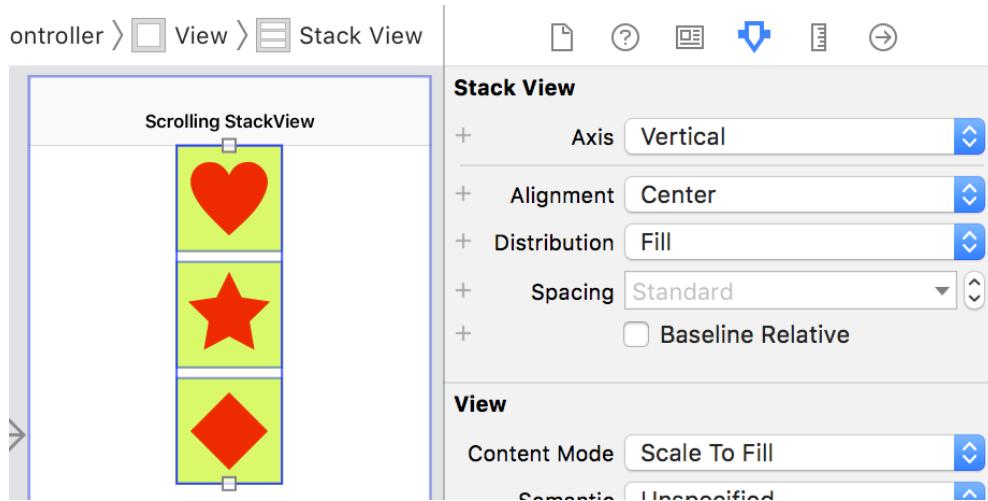
Don't use a scrolling stack view when a table or collection view would make more sense.

Here's an example of a stack view layout shown running on an iPhone SE in both portrait and landscape. The three images views are just too big to fit the available height in landscape:

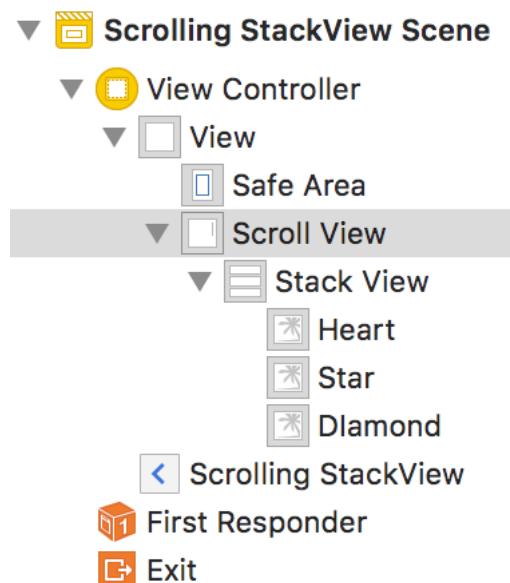


Let's work through an example of embedding this stack view in a scroll view (see sample code: [ScrollStack-v1](#)):

1. Here's our starting stack view with the three image views which are each 100 x 100 points in size. The stack view has the standard amount of spacing, fill distribution and center alignment:

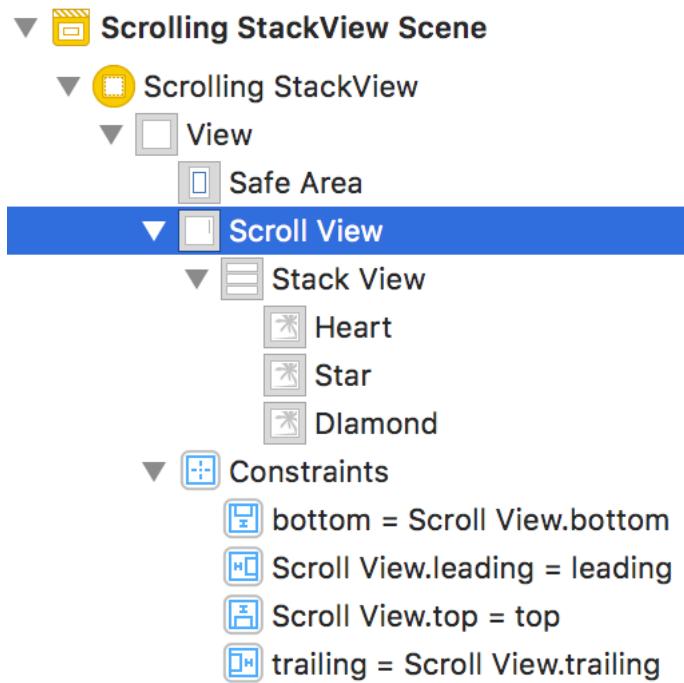


2. Select the stack view and then using either the **[Embed In]** tool from the toolbar at the bottom of the canvas or the Xcode menu Editor > Embed In > Scroll View embed the stack view in a scroll view:

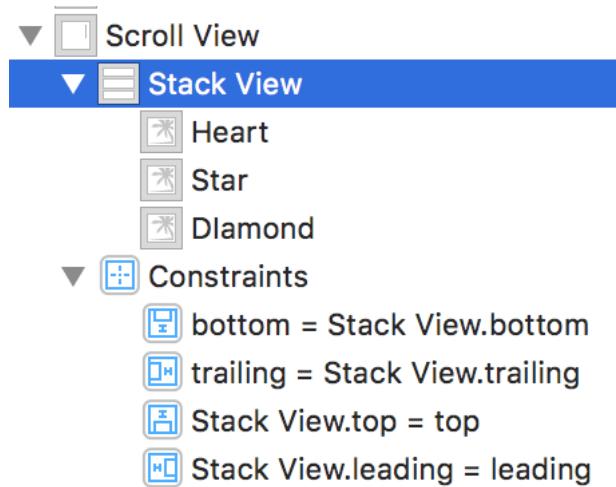


This adds a scroll view to the view hierarchy that contains our stack view.

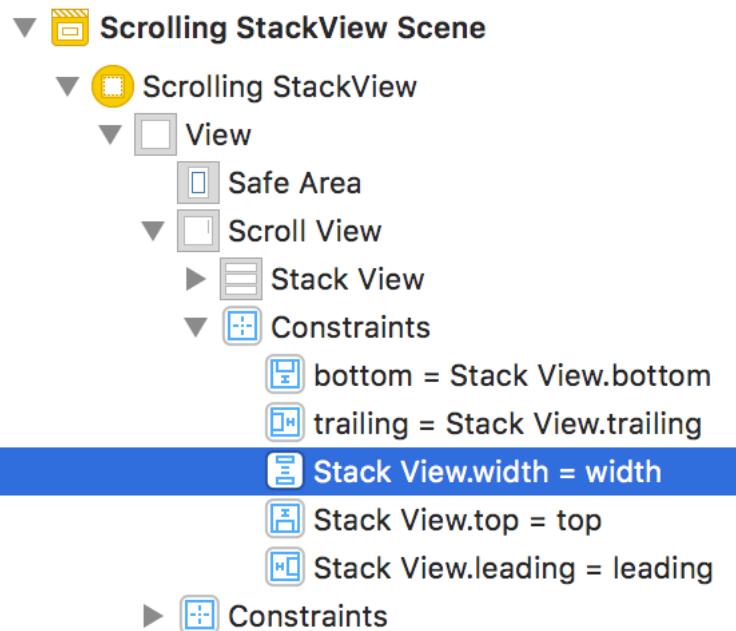
3. Let's start with the first group of constraints to fix the scroll view frame to its superview. Add four constraints to pin the scroll view to the edges of the root view:



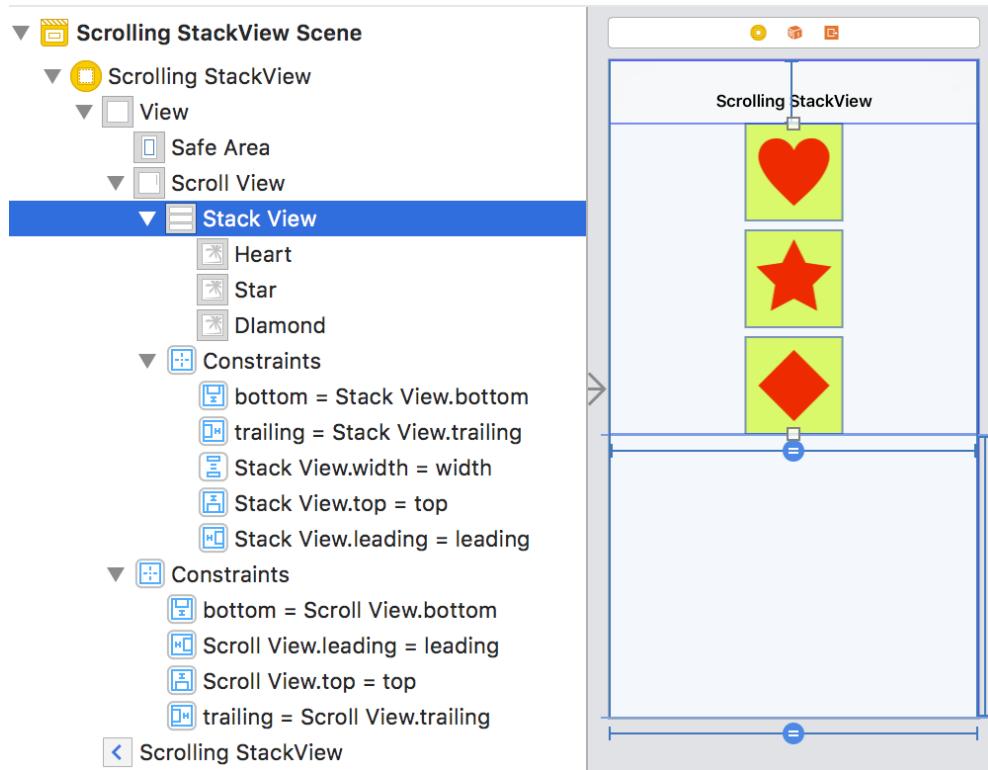
4. The stack view is the content for the scroll view so pin it to the edges of the scroll view with another four constraints:



5. There's one final constraint that's easy to forget when working with scroll views. Add an equal width constraint between the stack view and the scroll view to fix the width of the content area and prevent horizontal scrolling:

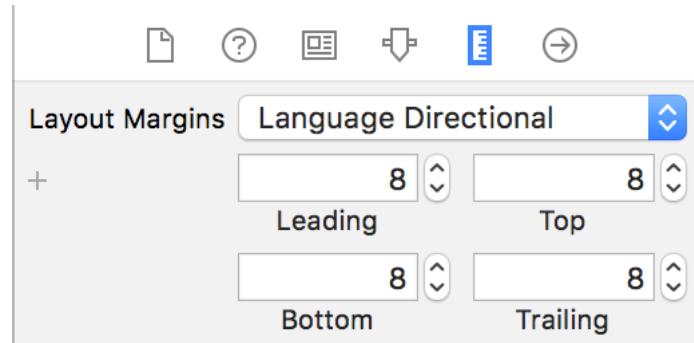


- There should be a total of nine constraints when done. Four to fix the size and position of the scroll view frame and five for the scroll view content:

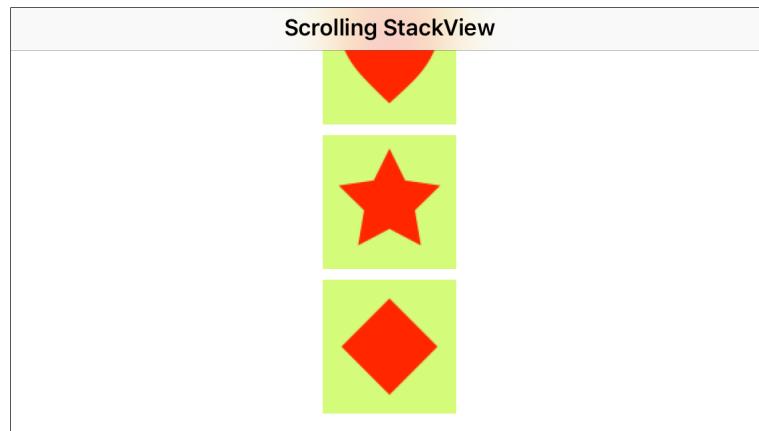


- I want to make one final tweak and add a small margin for the content.

Remember that a stack view doesn't use a margin by default, but we can add one. Select the stack view and use the size inspector to add 8 points of margin on all sides:



8. The stack view contents should now be visible with a little scrolling on the smaller devices:



Frame And Content Layout Guides (iOS 11)

A lot of the confusion around scroll views and Auto Layout comes from the dual nature of the scroll view layout anchors. When constrained to a view external to the scroll view they are acting as the frame of the scroll view. When constrained to a content subview they act to set the content area of the scroll view.

To help remove this confusion Apple added two new layout guide properties to UIScrollView in iOS 11. The guides don't add any new functionality, but they might help you remember how to add your constraints:

```
var frameLayoutGuide: UILayoutGuide { get }
var contentLayoutGuide: UILayoutGuide { get }
```

As the names suggest, the frame layout guide matches the scroll view frame. We can use it when creating constraints from the scroll view to an external view or when we want to fix (float) a subview of the scroll view over the content. The content layout guide matches the content area of the scroll view so we can use it when constraining our scroll view content.

To see how we use them let's rework our scroll view example from the beginning of the chapter (see sample code: [ScrollView-v3](#)):

1. We are only going to rework the scroll view constraints. Everything else stays the same. I'm going to use both the frame layout guide and the content layout guide properties of the scroll view:

```
let frameGuide = scrollView.frameLayoutGuide  
let contentGuide = scrollView.contentLayoutGuide
```

2. Here are the first group of constraints we used to fix the size and position of the scroll view frame:

```
scrollView.leadingAnchor.constraint(equalTo:  
    view.leadingAnchor),  
scrollView.topAnchor.constraint(equalTo: view.topAnchor),  
scrollView.trailingAnchor.constraint(equalTo:  
    view.trailingAnchor),  
scrollView.bottomAnchor.constraint(equalTo:  
    view.bottomAnchor),
```

These are constraints between the scroll view and an external view so they constrain the scroll view frame. We can replace the scroll view with the frame layout guide:

```
frameGuide.leadingAnchor.constraint(equalTo:  
    view.leadingAnchor),  
frameGuide.topAnchor.constraint(equalTo: view.topAnchor),  
frameGuide.trailingAnchor.constraint(equalTo:  
    view.trailingAnchor),  
frameGuide.bottomAnchor.constraint(equalTo:  
    view.bottomAnchor),
```

3. The next group of constraints are the ones I always find confusing. I'm pinning the content container view to the scroll view:

```
scrollView.leadingAnchor.constraint(equalTo:  
    activityView.leadingAnchor),
```

```
scrollView.topAnchor.constraint(equalTo:  
    activityView.topAnchor),  
scrollView.trailingAnchor.constraint(equalTo:  
    activityView.trailingAnchor),  
scrollView.bottomAnchor.constraint(equalTo:  
    activityView.bottomAnchor),
```

These are constraints between the scroll view and a subview so constrain the content area. Constraining the content to the content layout guide makes it clearer:

```
contentGuide.leadingAnchor.constraint(equalTo:  
    activityView.leadingAnchor),  
contentGuide.topAnchor.constraint(equalTo:  
    activityView.topAnchor),  
contentGuide.trailingAnchor.constraint(equalTo:  
    activityView.trailingAnchor),  
contentGuide.bottomAnchor.constraint(equalTo:  
    activityView.bottomAnchor),
```

4. Our final constraint fixes the width of the content area. We did that by making the width of the container view equal to the scroll view width:

```
scrollView.widthAnchor.constraint(equalTo:  
    activityView.widthAnchor)
```

Using the new layout guides, we can write this more clearly by making the widths of the content layout guide and frame layout guide the same:

```
frameGuide.widthAnchor.constraint(equalTo:  
    contentGuide.widthAnchor)
```

Using the new guides takes the same number of constraints, so it's not any less work. However, I think they are easier to understand:

- Pin the frame layout guide to the superview to fix the size and position of the scroll view.
- Pin the content view to the content layout guide.
- If necessary create width or height constraints between the content and frame layout guides to disable horizontal or vertical scrolling.

Floating Content With Layout Guides

You sometimes want a subview of the scroll view to have a fixed position in the scroll view frame, so it appears to float over the scrollable content. You might do this by constraining the subview to some view external to the scroll view. The frame layout guide makes this much more flexible. For example, we could pin a button to the top-left corner of the scroll view.

```
// let frameGuide = scrollView.frameLayoutGuide
button.topAnchor.constraint(equalTo: frameGuide.topAnchor),
button.leadingAnchor.constraint(equalTo:
    frameGuide.leadingAnchor)
```

One problem with this is that the top-left corner of our scroll view frame might be hidden behind a navigation bar. A better choice might be to use the scroll view's layout margin guide. Let's add an info button to our scroll view that sticks in the top left corner when the content scrolls (see sample code: [ScrollView-v4](#)):

1. Create an info button in our view controller:

```
private let infoButton: UIButton = {
    let button = UIButton(type: .infoDark)
    button.translatesAutoresizingMaskIntoConstraints = false
    button.addTarget(self, action: #selector(showInfo(_:)),
        for: .touchUpInside)
    return button
}()
```

2. Add it as a subview when creating the scroll view:

```
scrollView.addSubview(infoButton)
```

3. Then we need two constraints to pin the button to the layout margin guide of the scroll view:

```
infoButton.leadingAnchor.constraint(equalTo:
    scrollView.layoutMarginsGuide.leadingAnchor),
infoButton.topAnchor.constraint(equalTo:
    scrollView.layoutMarginsGuide.topAnchor)
```

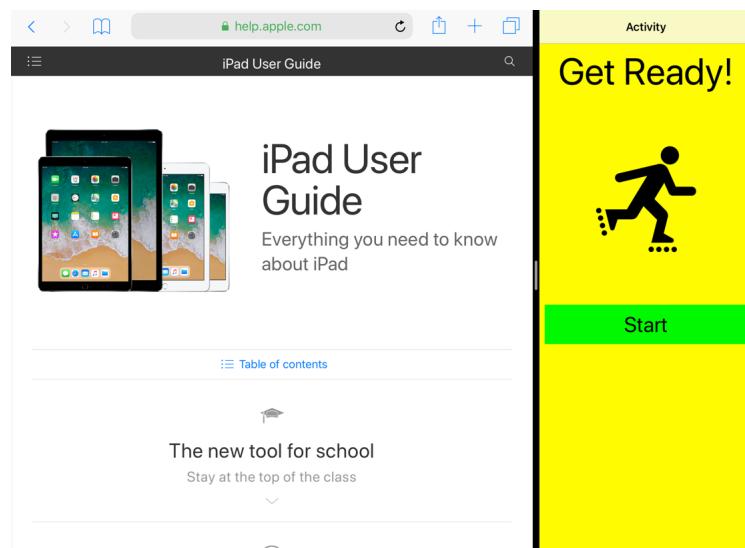
4. Here's how it looks on an iPhone 8 in portrait and landscape. The info button stays in the top-left corner even when the content scrolls:



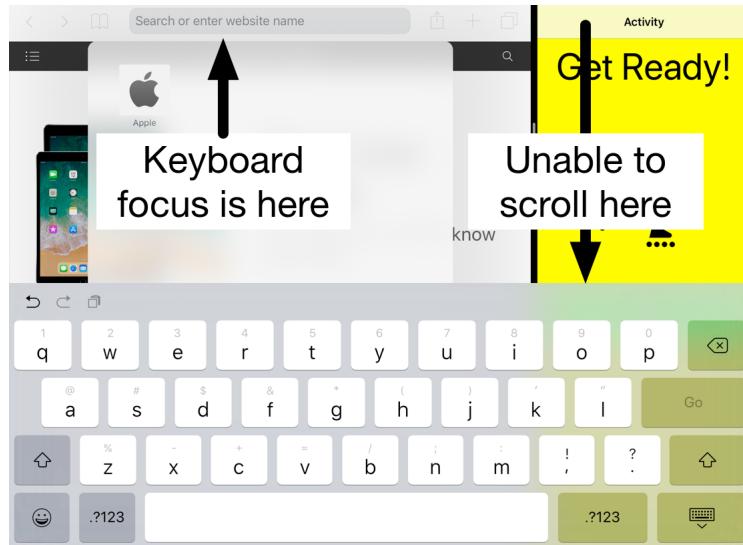
Managing The Keyboard

If you have ever used a layout that needs text input you may have already had to deal with the keyboard. Depending on your layout you may need to scroll the view to move content out of the way of the keyboard.

There's another more unexpected situation where you need to use a scroll view to move content out of the way of the keyboard. We'll look at [Supporting iPad Multitasking](#) in a later chapter but here's my App running on the right-hand side of a split-screen iPad sharing the screen with Safari:



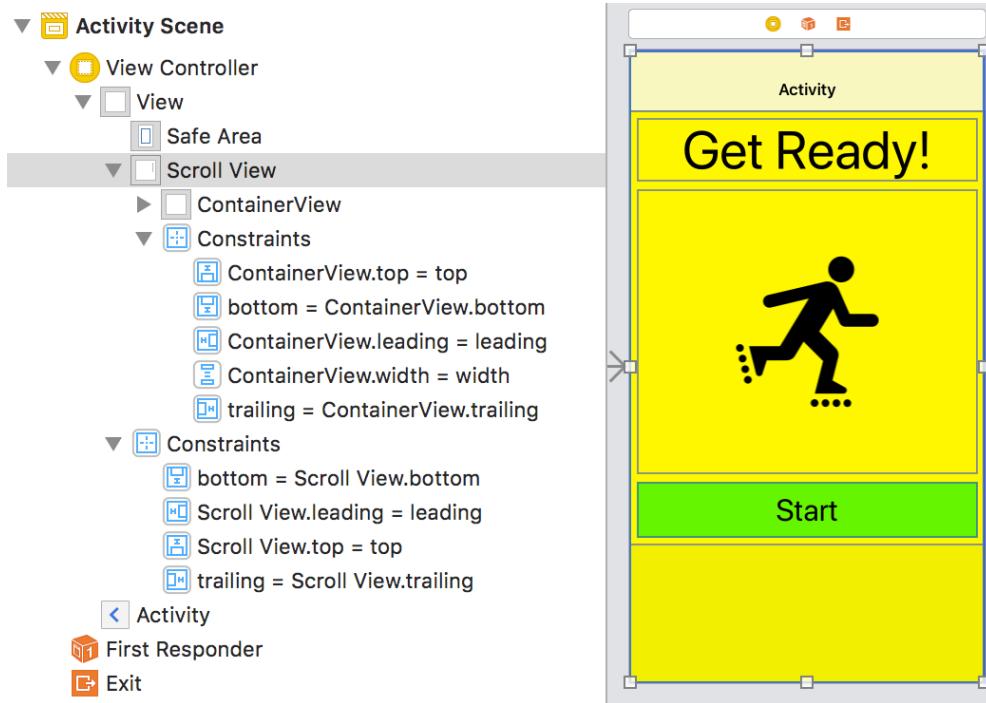
What can be surprising is that our App can end up covered by the keyboard even though it doesn't need text input. Look what happens if I tap on an input field in Safari:



The keyboard appears and covers the bottom part of the screen. My App is not aware of this and does nothing to adjust the content that's now hidden by the keyboard. Even though we are using a scroll view, there's no way to scroll down to view the hidden content without dismissing the keyboard.

The solution is to use the `contentInset` of the scroll view. When the system shows or hides the keyboard, we change the bottom content inset of the scroll view to allow for the height of the keyboard (see sample code: [ScrollView-v5](#)):

1. Here's a reminder of how our layout looks when built with Interface Builder. We embedded the container view for our content in a scroll view pinned to the edges of the root view:



2. We could manage the keyboard in the view controller, but I find it more reusable with a subclass of `UIScrollView`. Here's the template for a scroll view subclass with initializers to work with Interface Builder and programmatic layouts:

```
// AdaptiveScrollView.swift
import UIKit
class AdaptiveScrollView: UIScrollView {
    override init(frame: CGRect) {
        super.init(frame: frame)
        setup()
    }

    required init?(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
        setup()
    }

    private func setup() {
        // setup code here
    }
}
```

3. The setup code adds observers for the two notifications indicating the system has shown or is about to hide the keyboard. First (in `setup()`) the notification for when the keyboard has been shown:

```
NotificationCenter.default.addObserver(self,
    selector: #selector(keyboardDidShow(_:)),
    name: UIResponder.keyboardDidShowNotification,
    object: nil)
```

Then the notification for when the keyboard will hide:

```
NotificationCenter.default.addObserver(self,
    selector: #selector(keyboardWillHide(_:)),
    name: UIResponder.keyboardWillHideNotification,
    object: nil)
```

4. The method called when the system shows the keyboard:

```
@objc private func keyboardDidShow(_ notification:
    Notification) {
    guard let userInfo = notification.userInfo,
        let frame =
            userInfo[UIResponder.keyboardFrameEndUserInfoKey]
        as? NSValue else {
        return
    }

    let keyboardSize = frame.cgRectValue.size
    let contentInsets = UIEdgeInsets(top: 0.0, left: 0.0,
        bottom: keyboardSize.height, right: 0.0)
    adjustContentInsets(contentInsets)
}
```

We check the userInfo dictionary of the notification for a key and if present get the value. It's an NSValue containing a CGRect for the keyboard frame. We only need the height from the frame which we use to create extra padding at the bottom of the scroll view.

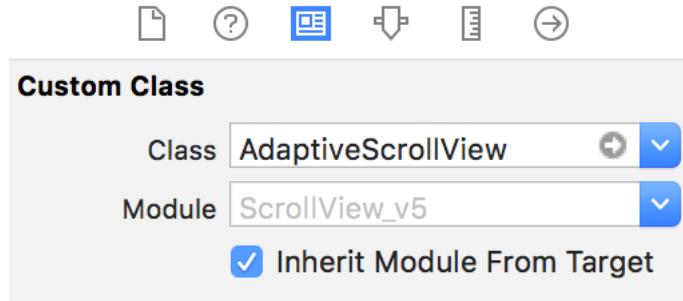
5. The `adjustContentInsets` function sets the content inset of the scroll view. It also sets the scroll indicator insets, so the scrolling indicators don't get hidden by the keyboard:

```
private func adjustContentInsets(_ contentInsets:
    UIEdgeInsets) {
    contentInset = contentInsets
    scrollIndicatorInsets = contentInsets
}
```

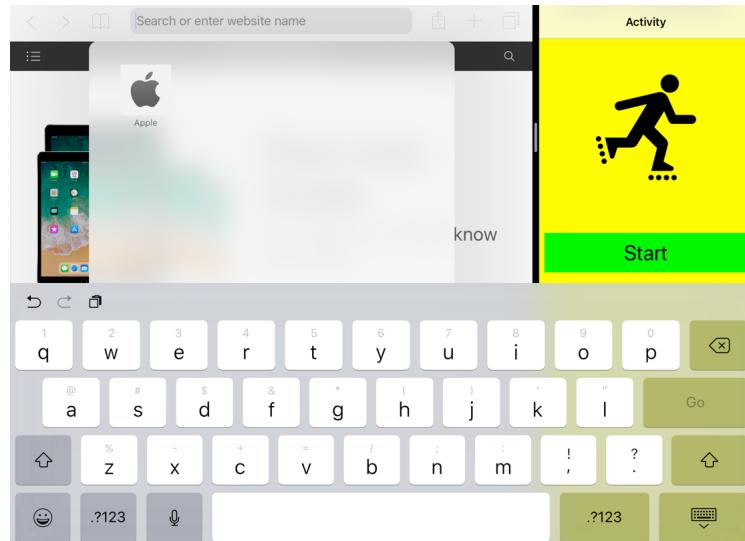
6. When the system hides the keyboard we use the notification to reset the content insets to zero:

```
@objc private func keyboardWillHide(_ notification:  
    Notification) {  
    adjustContentInsets(.zero)  
}
```

- Finally, we need to change the class of the scroll view used in the storyboard to use our custom adaptive scroll view:



- If we set things up right we should now be able to scroll our content all the way to the bottom when the split view keyboard is showing:



You don't need to worry about managing the keyboard if you're using the UIKit table and collection views. They are both scroll views but already take care of adjusting their content to allow for the keyboard.

Key Points To Remember

A scroll view is a useful tool for building flexible layouts. Take the time to master it as we make good use of it in the rest of this book:

- Add your content as a subview of the scroll view. If you have more than one subview use a container view or a stack view.
- A scroll view treats constraints differently:
 - A constraint between a scroll view and a view outside the scroll view constrains the frame of the scroll view.
 - A constraint between a scroll view margin or edge and a subview constrains the subview to the content area of the scroll view.
 - A height or width constraint between the scroll view and a subview constrains the subview with the scroll view's frame.
- Constrain your scroll view to the superview to set its frame. Use the frame layout guide if available (iOS 11)
- Constrain your content view to the scroll view to set the content area. Use the content layout guide if available (iOS 11).
- Use an equal width or height constraint between the content view and the scroll view to disable scrolling horizontally or vertically. Create these constraints between the content layout guide and frame layout guide if available (iOS 11).
- Remember the keyboard even if you're not expecting keyboard input. Adjust the `contentInset` of the scroll view when the keyboard appears or disappears.

Test Your Knowledge

Challenge 11.1 Scrolling A Stack View

Can you rebuild the activity layout I showed in this chapter with a stack view? Here's a reminder of the layout:



1. Build this layout using Interface Builder or in code. Use the following view metrics:
 - The label is center aligned and uses a 56 point system font.
 - The image is a 330 x 300 point vector image. Use your own image or grab mine from the sample code for this chapter.
 - The button title uses a 32 point system font with 10 point top and bottom content insets.
 - There's a standard amount of vertical spacing between the views and an 8 point margin on each side.
2. Test the layout on iPhones and iPads in both portrait and landscape. Make sure that the view scrolls to show the button in landscape.

Hints And Tips

1. There's no need to use an extra container view. Add the label, image view, and button to a stack view and embed the stack view directly in a scroll view.
2. If you're building the layout in code, there's no need to create the custom subclass of `UIView`. Build the layout in the view controller.
3. If you keep the target to at least iOS 11 feel free to use the frame and content layout guides in your constraints.
4. With the stack view embedded directly in the scroll view, you need to have the stack view provide the 8 point margins for the content. See [Stack View Margins](#) for a reminder.

Challenge 11.2 Floating Content

Can you add a floating button to the last layout?

1. Add an info button to the scroll view of the last layout.
2. The info button should stay fixed to the top-left corner margin of the view below the navigation bar.
3. Scrolling the content should not change the position of the info button.

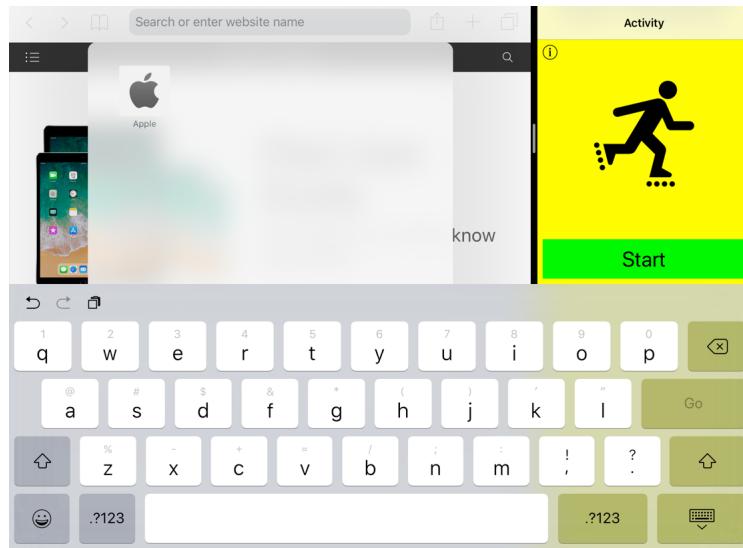


Hints And Tips

1. This is not easy to do with Interface Builder. If you add the button to the scroll view and pin it to the scroll view margins, the scroll view treats it as content, and it scrolls with the scroll view. You can add it to the root view, but I prefer to handle this type of addition in code.
2. If you're building the main layout in Interface Builder but adding the button in code, you need an outlet for the scroll view in the view controller.
3. Add the button as a subview of the scroll view.
4. Constrain the button to the leading and top margins of the scroll view.

Challenge 11.3 Managing The Keyboard

Fix the layout so that it's aware of the keyboard when used in a split screen on an iPad. With the keyboard visible the layout should scroll vertically to allow the button to be visible:



1. Change your solution to the last challenge to make the scroll view aware of the keyboard.
2. Test the layout on an iPad in split screen and trigger the keyboard in the other App.
3. It must be possible to scroll the layout when the keyboard appears so that you can still see the bottom button.

Hints And Tips

1. You need to listen for the keyboard notifications and adjust the scroll view content insets when the system shows and hides the keyboard.
2. Feel free to copy the `AdaptiveScrollView` class we used in this chapter. See [Managing The Keyboard](#) for a recap.
3. If you use the `AdaptiveScrollView` class, you need to change the class of the scroll view in Interface Builder or where you create the scroll view in code.

Chapter 12

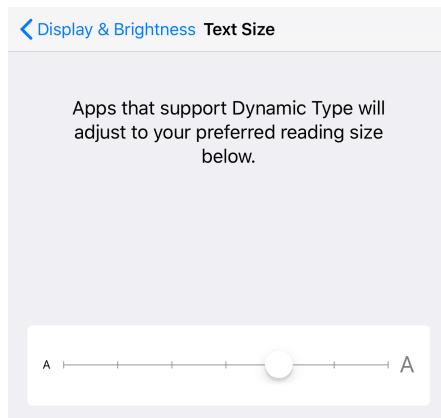
Dynamic Type

Many people have trouble reading text if the font size is too small. The definition of “too small” is also different for different people. Dynamic type puts the user in control by allowing them to change their preferred text size. In this chapter we look at how you add support for dynamic type to your Apps:

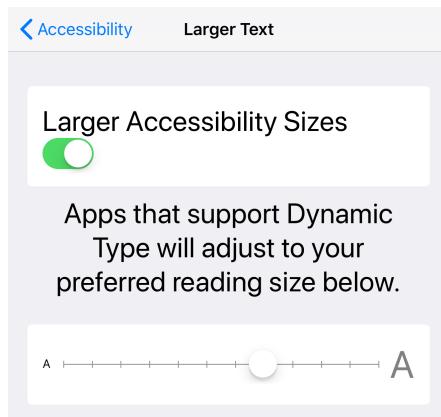
- How to adopt dynamic type by using text styles in Interface Builder or when building your layout in code.
- How new features in iOS 10 make supporting dynamic type less work (and what to do if you still need to support iOS 9).
- How readable content guides work with dynamic type to keep long lines of text to a readable width.
- How to scale and use custom fonts with dynamic type.

Using Dynamic Type

Dynamic type, introduced by Apple back in iOS 7, puts the user in control of the size of the text in your application. The user sets their preferred text size in the device settings and applications are then expected to adjust to the preferred size. There are seven standard sizes from extra-small to extra-extra-extra-large:



The user can also choose from five extra large accessibility sizes (General > Accessibility > Larger Text):



To support dynamic type you set the font for your labels, text fields or text views using a text style. There were six choices when introduced with iOS 7:

```
.headline    .subheadline  .body  
.footnote   .caption1    .caption2
```

In iOS 9 Apple added four more styles:

```
.title1     .title2      .title3  
.callout
```

Then in iOS 11, Apple added the large title style:

```
.largeTitle
```

When you use a text style, you no longer choose the font family, weight or size. Since iOS 9 the font is the Apple San Francisco system typeface

with a size and weight adjusted for the user size preference and the text style. This is how the various text styles look at extra-small, extra-large and accessibility extra-extra-extra-large sizes on iOS 11:

Title 1

Title 2

Title 3

Headline

Subhead

Body

Callout

Footnote

Caption 1

Caption 2

Title 1

Title 2

Title 3

Headline

Subhead

Body

Callout

Footnote

Caption 1

Caption 2

Title 1

Title 2

Title 3

Headline

Subhead

Body

Callout

Footnote

Caption 1

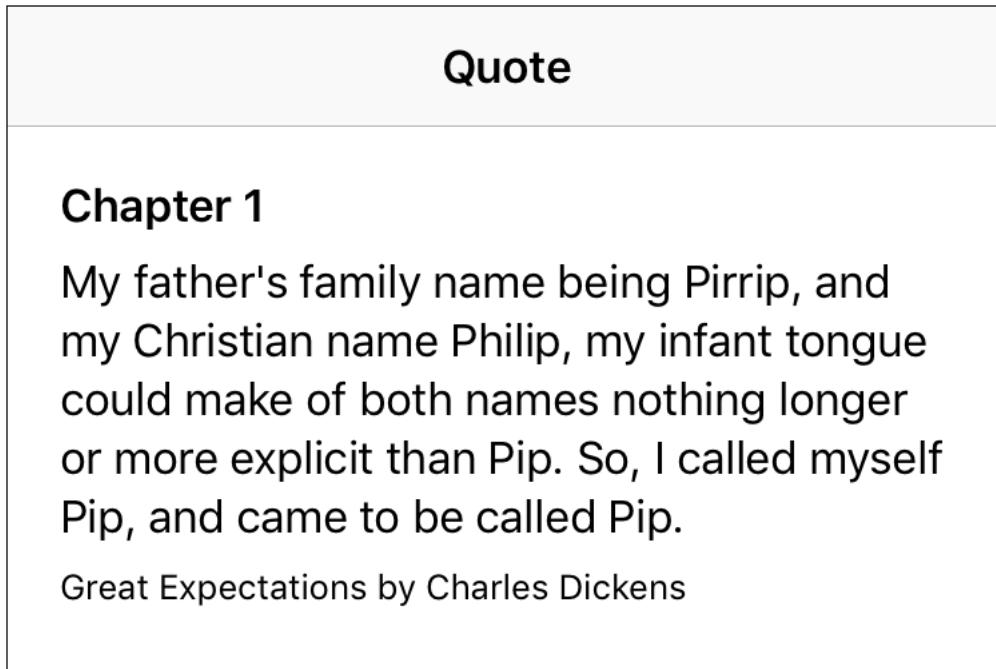
Caption 2



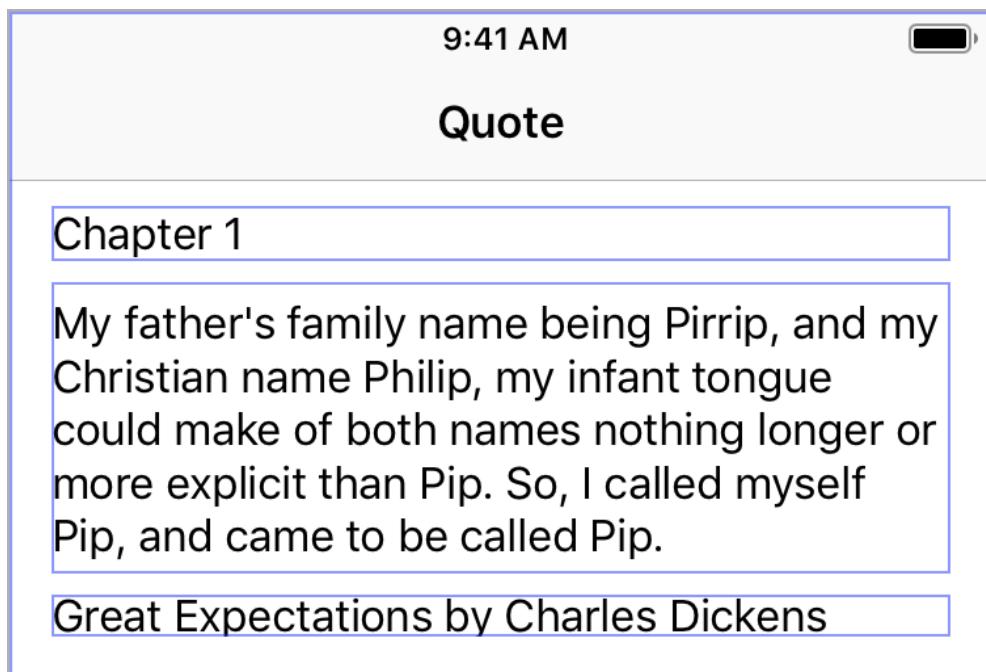
Before iOS 11 the extra large accessibility sizes only affected the `.body` style.

Text Styles In Interface Builder

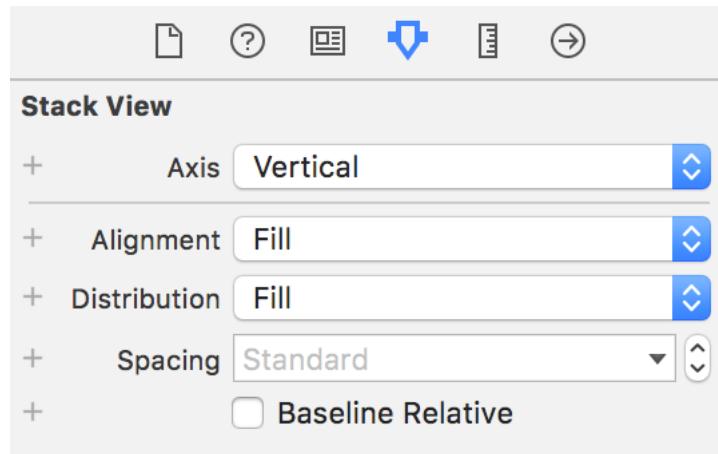
Let's use Interface Builder to build this layout using the headline, body and footnote text styles (see sample code: [DynamicType-v1](#)):



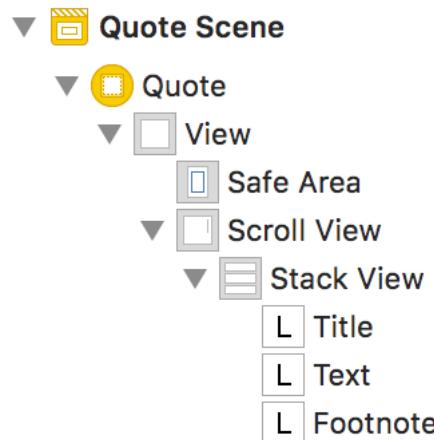
1. Starting from the Xcode Single View App iOS application template. Embed a view controller in a navigation controller and drag three labels into the canvas. Set the number of lines for each label to zero and add some text:



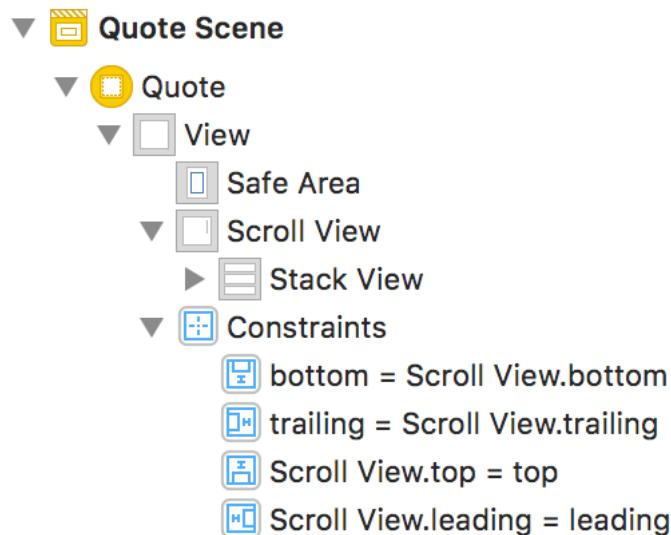
2. Select the three labels and embed them in a vertical stack view (Editor > Embed In > Stack View or use the **[Embed In]** tool). Configure the stack view to use a fill alignment, fill distribution and standard spacing:



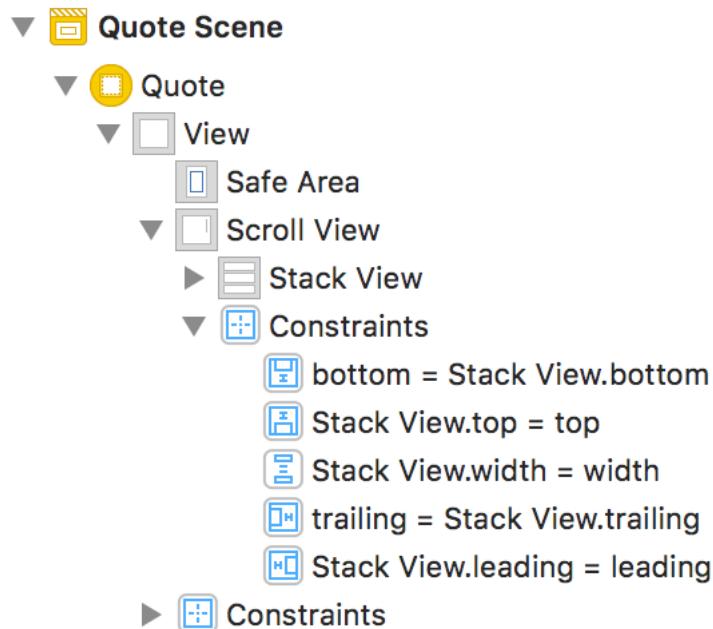
3. Since the vertical height of our text can grow significantly, we need to allow it to scroll. Let's use the technique we learned in the last chapter. See [Scrolling A Stack View](#) for a recap. Select the stack view and embed it in a scroll view (Editor > Embed In > Scroll View or use the **[Embed In]** tool):



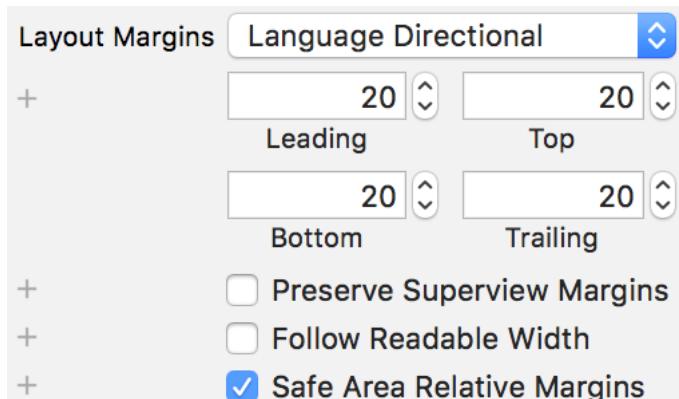
4. Add four constraints to pin the scroll view to the edges of the root view:



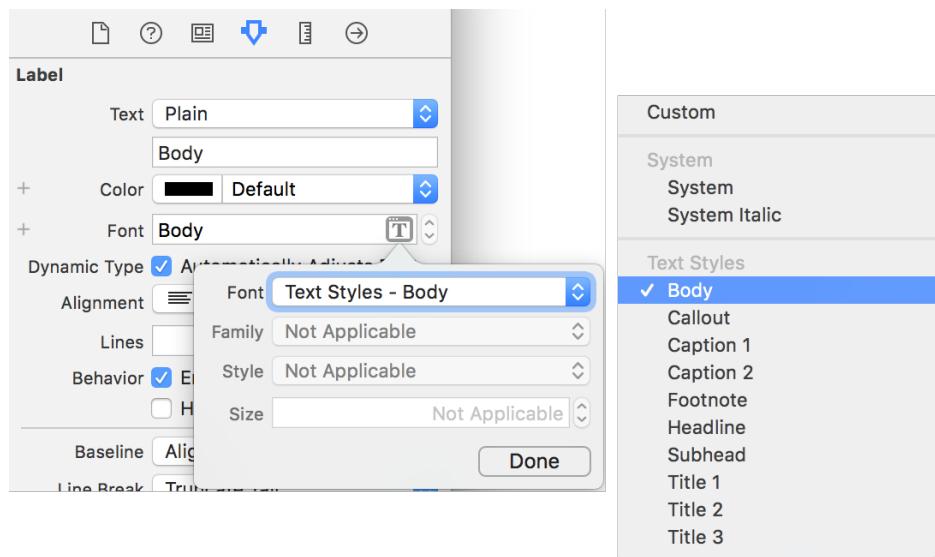
5. Add four constraints to pin the stack view to the scroll view content area and then add the final constraint to give the scroll view and stack view equal widths:



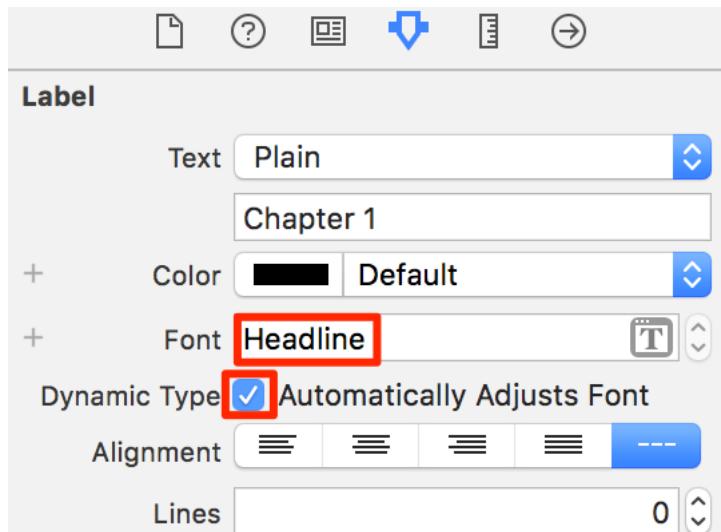
6. We have the stack view pinned to the edges of the scroll view which is pinned to the edges of the root view. To keep our content inside the safe area, with some extra padding, add a 20pt margin to the stack view using the size inspector:



7. We have the layout done, but we are still using fixed fonts for our labels. In Interface Builder you access the text styles from the font drop-down menu:

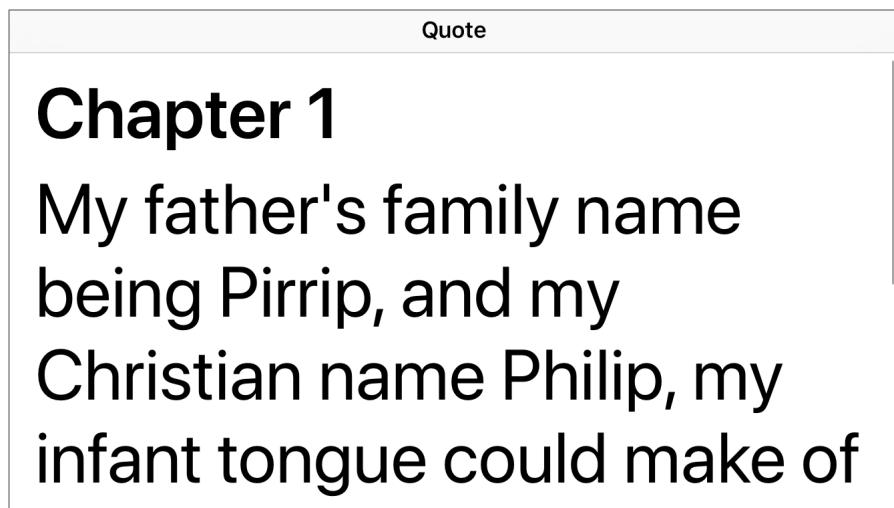


8. Select the first label and using the attributes inspector change the font to "Headline":



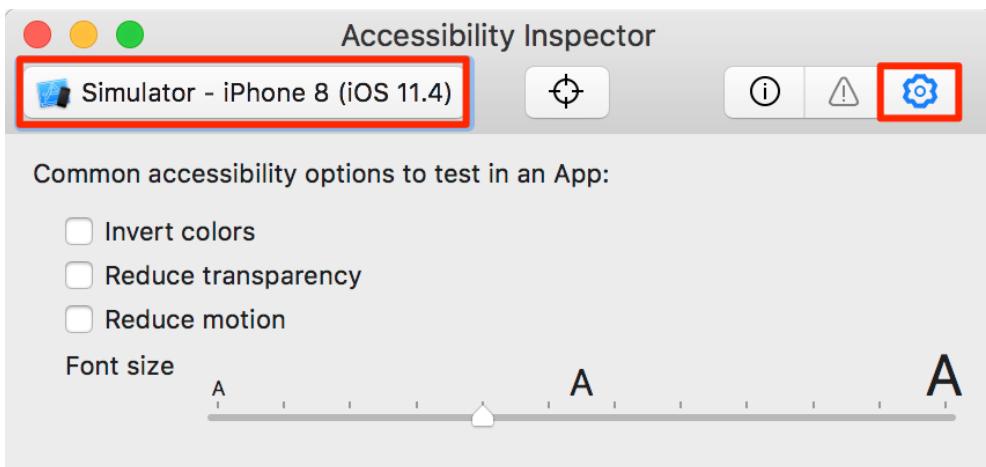
Make sure to select the "Automatically Adjusts Font" checkbox so that the system updates the font size anytime the user changes their preferred text size.

9. Repeat for the other two labels using the Body and Footnote text styles. Don't forget to select the "Automatically Adjusts Font" checkbox for both labels.
10. Build and run. Here's how it looks on an iPhone 8 in landscape at the largest accessibility size, so you need to scroll to see all of the text:



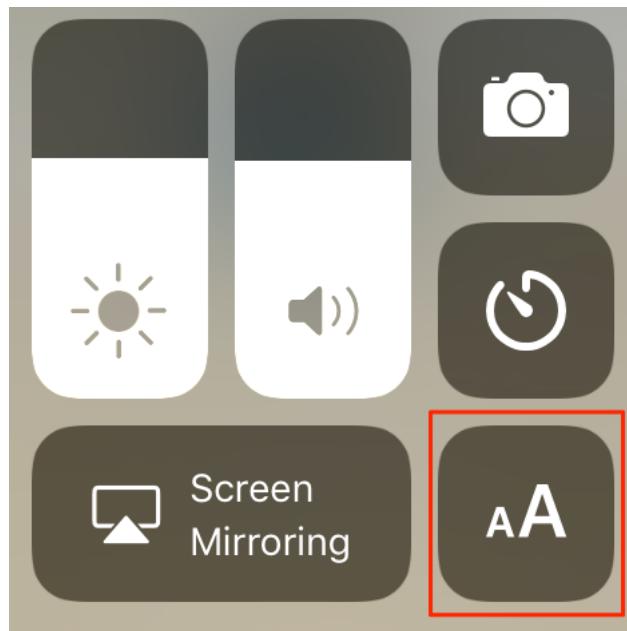
Testing Dynamic Type With The Accessibility Inspector

Changing the preferred content size in the settings app can be painful when you're developing a layout with dynamic type. The Accessibility Inspector included with the Xcode developer tools is quicker. See Xcode > Open Developer Tool > Accessibility Inspector:



Select the simulator or device running your App (top left) and then select the Settings tool (top right). You can then slide the font size control left and right to change the preferred content size of your running App.

To quickly change the text size on a device you can also customize the Control Centre (Settings > Control Centre > Customize Controls) to include the Text Size control:



Text Styles In Code

When setting the font in code use the `UIFont` class method. For example, to set a label to use the Body style:

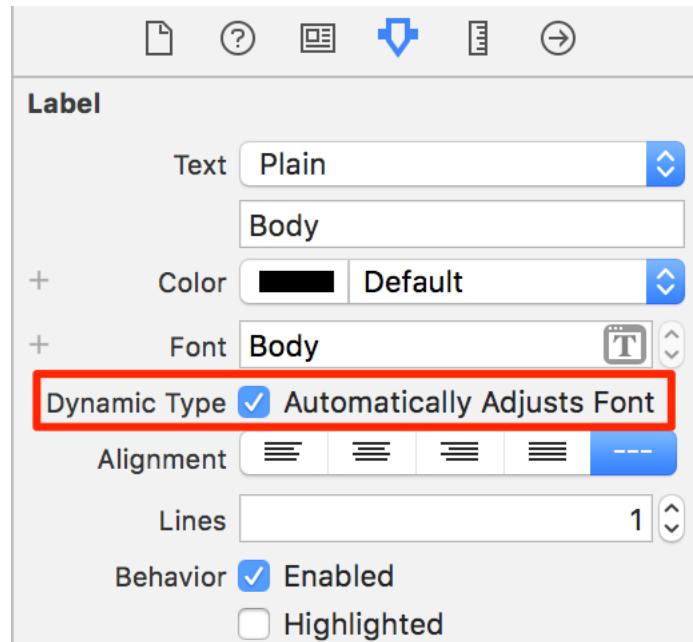
```
label.font = UIFont.preferredFont(forTextStyle: .body)  
label.adjustsFontForContentSizeCategory = true
```

Note the `adjustsFontForContentSizeCategory` property. Apple added this property to labels, text views and text fields in iOS 10. It's the same as selecting the "Automatically Adjusts Font" checkbox in Interface Builder. To rebuild the last example in code I created a small extension to configure the label (see sample code: [DynamicType-v2](#)):

```
// UILabel+Style.swift  
import UIKit  
extension UILabel {  
    static func makeLabel(style: UIFont.TextStyle) -> UILabel {  
        let label = UILabel()  
        label.font = UIFont.preferredFont(forTextStyle: style)  
        label.adjustsFontForContentSizeCategory = true  
        label.numberOfLines = 0  
        return label  
    }  
}
```

Supporting Dynamic Type With iOS 9

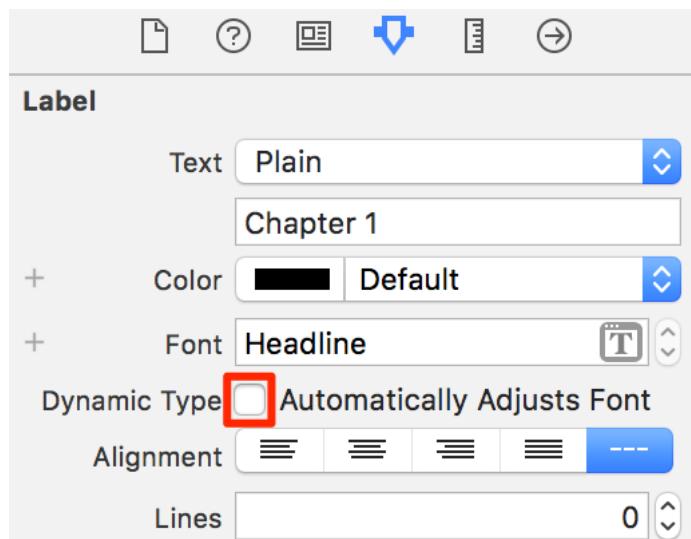
To support dynamic type you need to update the preferred font of your text any time the user changes the size. Apple made this automatic in iOS 10 when you select “Automatically Adjusts Font” in Interface Builder (or set the property in code):



You get a warning if you select this when not using text styles or if your deployment target is earlier than iOS 10. To use dynamic type with iOS 9 you must observe the content size category did change notification and manually update the fonts of any labels, text fields and text views that use dynamic type.

Let's rework our Interface Builder example to support iOS 9 (see sample code: [DynamicType-v3](#)):

1. Once we change the deployment target of the project to iOS 9 we can no longer select the “Automatically Adjusts Font” checkbox in Interface Builder for any of the labels:



- Instead, we check the iOS availability in the `viewDidLoad` method of the view controller. If we are at iOS 10 or later, we can set the property in code for each of the labels. Otherwise, we need to create an observer for the notification:

```
override func viewDidLoad() {
    super.viewDidLoad()
    configureView()

    if #available(iOS 10, *) {
        [headline, body, footnote].forEach {
            $0?.adjustsFontForContentSizeCategory = true
        }
    } else {
        NotificationCenter.default.addObserver(self,
            selector: #selector(updateTextStyles(_:)),
            name: UIContentSizeCategory.didChangeNotification,
            object: nil)
    }
}
```

- Then in the method that handles the notification, we update the font of each of the labels:

```
@objc private func updateTextStyles(_ notification:
    Notification) {
    titleLabel.font = UIFont.preferredFont(forTextStyle:
        .headline)
    textLabel.font = UIFont.preferredFont(forTextStyle:
        .body)
```

```
footnoteLabel.font = UIFont.preferredFont(forTextStyle:  
    .footnote)  
}
```

The lack of auto-adjusting font sizes makes supporting dynamic type on iOS 9 and earlier a bit of a pain. Remember to go back and clean up this code when you drop support for iOS 9.



There's an annoying bug with the iOS 9 simulator (other iOS releases are not affected) that means the observer method for the content size category did change notification is never called. Test iOS 9 on a device when working with dynamic type.

Readable Content Guides

Long lines of text are hard to read. Opinions vary on the ideal length from as low as 45 characters up to as much as 100 characters. Here's our dynamic type example when viewed on a 10.5" iPad Pro in landscape using the default text content size (squeezed a little to fit here):

Quote
<p>Chapter 1 My father's family name being Pirrip, and my Christian name Philip, my infant tongue could make of both names nothing longer or more explicit than Pip. So, I called myself Pip, and came to be called Pip. Great Expectations by Charles Dickens</p>

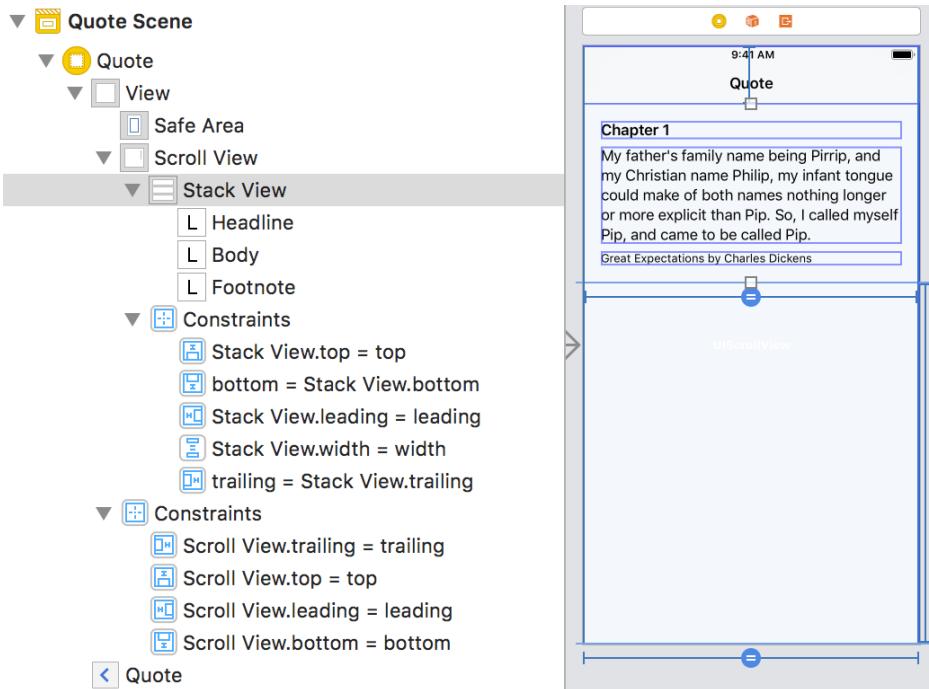
This layout might be fine on an iPhone, but on an iPad, it leads to uncomfortably long lines. The line length is over 130 characters which is not much fun to read.

Apple thought of this when they added layout guides in iOS 9 and included a readable content guide. This is a layout guide that creates a readable width within a view taking into account the user's preferred text content size.

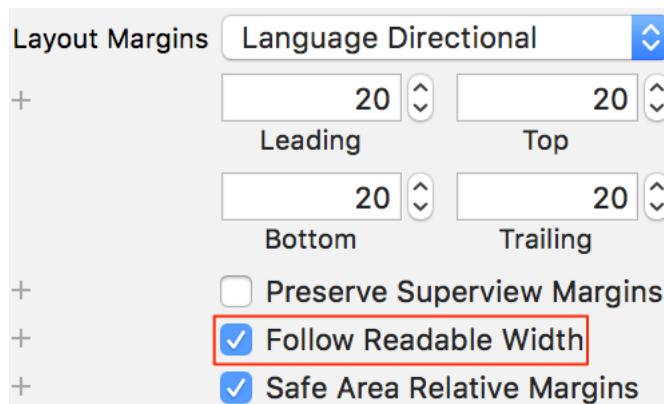
Readable Content Guide In Interface Builder

It's possible to use the readable content guide in Interface Builder, but the feature is not easy to find. Let's change our dynamic type example to see how it works (see sample code: [DynamicType-v4](#)):

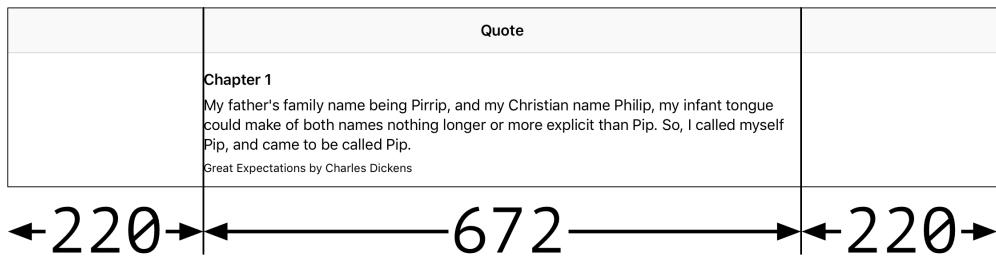
- To recap, the layout we built in Interface Builder has three labels embedded in a vertical stack view which I then embedded in a scroll view. We are using the layout margins of the stack view to inset the labels from the edges:



- This is a working layout it just doesn't work well on a wide device. The solution is to constrain the labels to the readable content guide instead of the margins of the stack view. In Interface Builder select the stack view and using the size inspector select "Follow Readable Width". You'll find it below the layout margins:

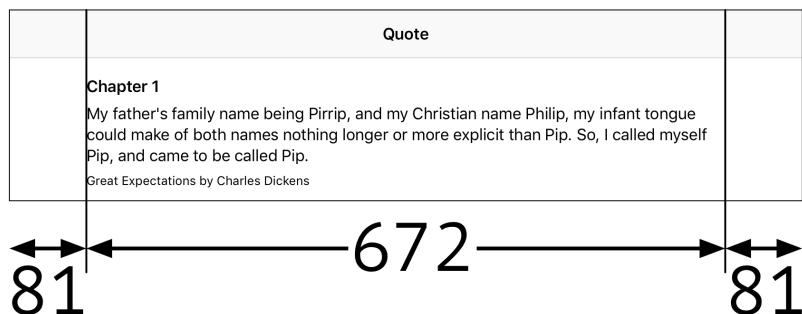


- If you build and run now you should see a much more readable line length. I added some size guides to make it easier to see what's happening:



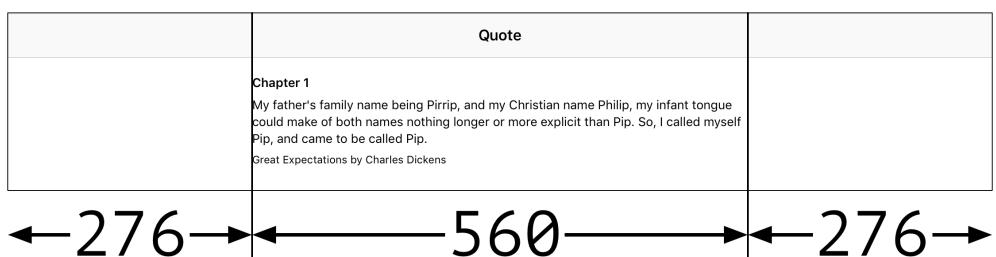
The readable content guide, in this case, defines an area that's 672 points wide and centered in the superview. The line length at the default dynamic type content size is now a much more readable 87 characters.

4. The system maintains a readable line length for us as the view width and font size changes. Rotating the iPad to portrait keeps the line length the same, albeit with smaller margins:



The readable content guide never goes beyond the layout margin guide and is always centered within it.

5. Changing the dynamic type size changes the readable content guide width to maintain the line width. If I choose the smallest dynamic type content size the content guide width shrinks from 672 points to 560 points:

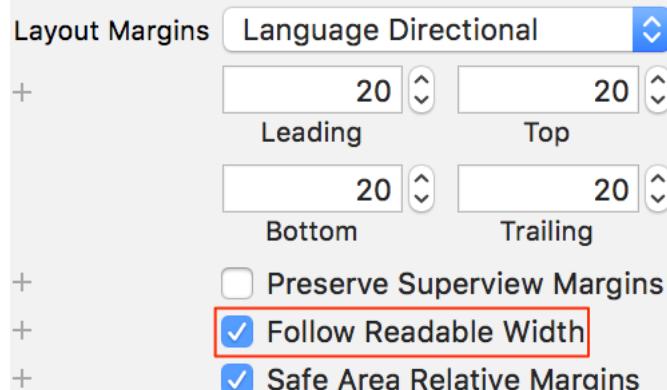


Readable Content Guide In Code

The `readableContentGuide` is a property of `UIView`. It's a layout guide so you use it the same way you use `layoutMarginsGuide` when creating constraints in code.:

```
let marginGuide = view.readableContentGuide
NSLayoutConstraint.activate([
   .textLabel.leadingAnchor.constraint(equalTo:
        marginGuide.leadingAnchor),
   .textLabel.trailingAnchor.constraint(equalTo:
        marginGuide.trailingAnchor),
    ...
])
```

This works fine when using the readable content guide of a normal view. If you're using a stack view it's a bit awkward. A stack view has a readable content guide but you don't use it directly. In the last example, we configured the stack view in Interface Builder by selecting the "Follow Readable Width" option in the size inspector:



Unfortunately that option doesn't exist when working with a stack view in a programmatic layout. We can give the stack view a margin and tell it to use it:

```
stackView.directionalLayoutMargins =
    NSDirectionalEdgeInsets(top: padding, leading: padding,
                           bottom: padding, trailing: padding)
stackView.isLayoutMarginsRelativeArrangement = true
```

What we're missing is a way to configure the stack view to use the readable content guide instead of the layout margin guide.

One workaround is to embed the stack view in a container view and use its readable content guide to set the width of the stack view. Let's see

how we can modify our programmatic example to do that (see sample code: [DynamicType-v5](#)):

1. Here's a reminder of how we created our stack view in code. We no longer need the stack view margin as the container view will take care of the padding:

```
private lazy var stackView: UIStackView = {
    let stackView = UIStackView(arrangedSubviews:
        [titleLabel,.textLabel, footnoteLabel])
    stackView.translatesAutoresizingMaskIntoConstraints =
        false
    stackView.axis = .vertical
    stackView.spacing = UIStackView.spacingUseSystem
    return stackView
}()
```

2. We need to create a container view, give it the margin and add the stack view to it:

```
private lazy var containerView: UIView = {
    let view = UIView()
    view.translatesAutoresizingMaskIntoConstraints =
        false
    view.directionalLayoutMargins =
        NSDirectionalEdgeInsets(top: padding, leading: padding,
            bottom: padding, trailing: padding)
    view.addSubview(stackView)
```

3. We can now constrain the stack view to the **readable content guide** of the container view:

```
let readableGuide = view.readableContentGuide
NSLayoutConstraint.activate([
    stackView.leadingAnchor.constraint(equalTo:
        readableGuide.leadingAnchor),
    stackView.topAnchor.constraint(equalTo:
        readableGuide.topAnchor),
    stackView.trailingAnchor.constraint(equalTo:
        readableGuide.trailingAnchor),
    stackView.bottomAnchor.constraint(equalTo:
        readableGuide.bottomAnchor)
])
return view
}()
```

4. We embed the container view, not the stack view in the scroll view:

```
scrollView.addSubview(containerView)
```

5. The scroll view constraints also need to be modified. The scroll view content guide is now constrained to the container view not the stack view:

```
contentGuide.topAnchor.constraint(equalTo:  
    containerView.topAnchor),  
contentGuide.bottomAnchor.constraint(equalTo:  
    containerView.bottomAnchor),  
contentGuide.leadingAnchor.constraint(equalTo:  
    containerView.leadingAnchor),  
contentGuide.trailingAnchor.constraint(equalTo:  
    containerView.trailingAnchor),
```

Having to embed the stack view in a container view just to get a readable width is extra work. We could skip the stack view and add the labels directly to the container view. Alternatively we could have constrained our scroll view to the readable content guide of the root view but that prevents us from having content that stretches to the full width. Hopefully Apple will improve the stack view API in the future to make such workarounds unnecessary.

Text Views

I've only used text labels so far in this chapter, but dynamic type also works with text fields and text views. (You can also use dynamic type for the title label of a button but, I prefer to keep the button title at a fixed size.)

Text View, Text Field or Text Label?

If you're new to UIKit the differences between a label, text field or text view can be confusing. Here are some guidelines to help you choose between these text controls:

- **UITextField:** For input of a single line of text. Use secure entry for sensitive data such as passwords.
- **UILabel:** For displaying single or multiple lines of text. Selection and scrolling are not possible.
- **UITextView:** For display of multiple lines of text. Editing,

scrolling, and selection can each be enabled/disabled.

One reason to use a text view instead of a text label is to allow the user to select, copy and lookup words:

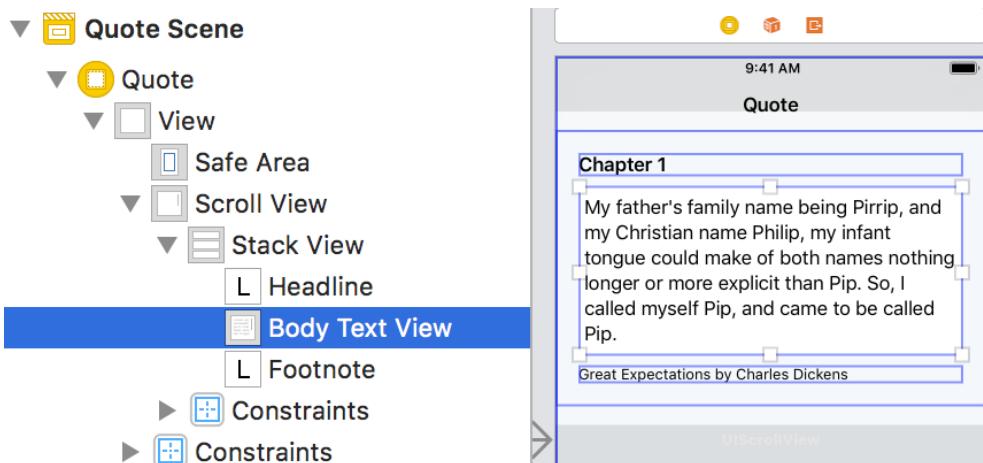
Chapter 1

My father's family being Pirrip, and
my Christian name Philip, my infant tongue
could make of both names nothing longer
or more explicit than Pip. So, I called myself
Pip, and came to be called Pip.

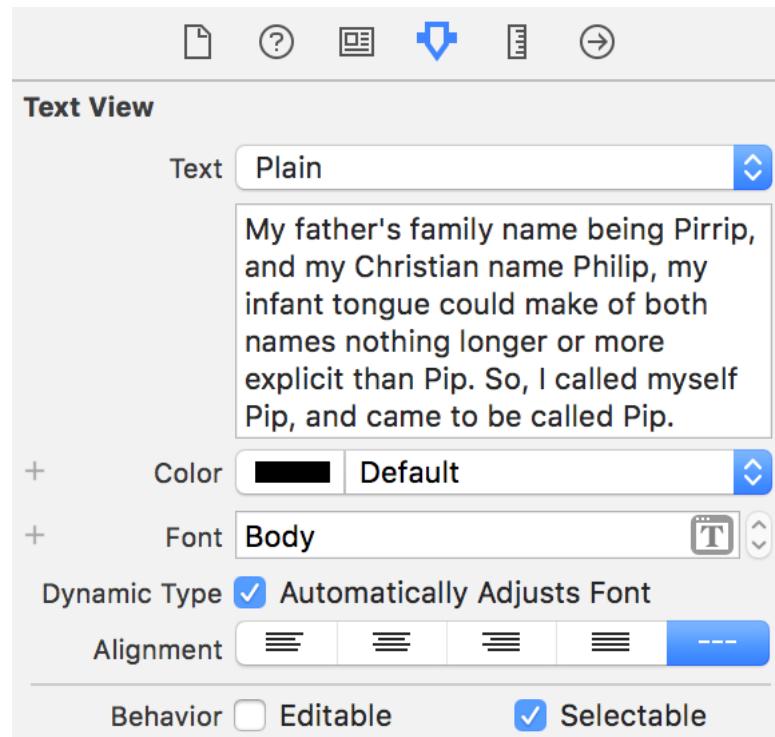
Great Expectations by Charles Dickens

Let's revisit our example, replacing the body label with a text view (see sample code: [TextView-v1](#)):

1. I'll avoid repeating the general setup. The only difference is that our stack view now contains a text view for the body text:

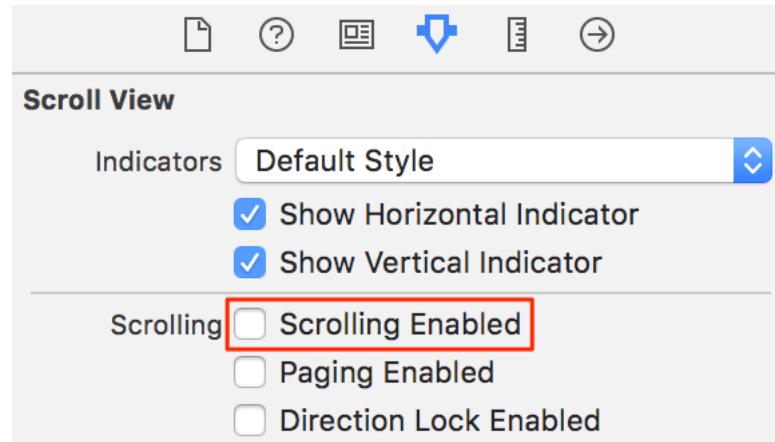


2. You configure a text view to use dynamic type in the same way as a text label. Choose the text style for the font in the attributes inspector and remember to select “Automatically Adjusts Font”:



We don't want the user to edit the text so deselect the "Editable" behavior. Leave "Selectable" enabled.

3. A text view is, by default, scrollable. We have embedded our stack view in a scroll view so that all three text items scroll together. To prevent the text view from scrolling on its own disable its scrolling behavior:



4. Build and run. You should have a working layout using dynamic type with the added ability for the user to select, copy and lookup text in the text view.
5. I should mention one feature of text views that annoys me. By default,

they include some extra padding when laying out lines. You can see this if you compare the leading edge of the body text view to the chapter and footnote labels:

Chapter 1

My father's family name being Pirrip, and my Christian name Philip, my infant tongue could make of both names nothing longer or more explicit than Pip. So, I called myself Pip, and came to be called Pip.

Great Expectations by Charles Dickens

6. You can remove this extra padding by changing the default value of the `lineFragmentPadding` property of the text view's text container. This property is not available in Interface Builder, so you need to do it in the view controller:

```
@IBOutlet private var textView: UITextView!  
  
override func viewDidLoad() {  
    super.viewDidLoad()  
    ...  
    textView.textContainer.lineFragmentPadding = 0  
}
```

Scaling Dynamic Type

When you use dynamic type, Apple decides the default typeface, font size and weight for each text style and content size. For example, the Body text style uses the San Francisco typeface at regular weight and is 17 points at the default (Large) content size.

You can use a custom typeface with dynamic type (we'll see how shortly) but what if you only want to change the weight, style or size of the text? You can change an attribute for a font with a font descriptor. A `UIFontDescriptor` describes a font with a dictionary of attributes.

To get a font descriptor for a given text style at the users preferred content size:

```
let descriptor =
    UIFontDescriptor.preferredFontDescriptor(withTextStyle:
        style)
```

When using font descriptors for one of the built-in text styles you can change the symbolic traits but not the font family. For example, using a descriptor to get a bold version of the Subhead text style:

```
let descriptor =
    UIFontDescriptor.preferredFontDescriptor(withTextStyle:
        .subheadline)
if let boldDescriptor =
    descriptor.withSymbolicTraits(.traitBold) {
    label.font = UIFont(descriptor: boldDescriptor, size: 0)
}
```

The `size` parameter allows you to scale a font when creating it from a font descriptor. If you only want to scale the font, you can also skip the descriptor. For convenience we can define a `UIFont` extension that returns a scaled version of a given text style:

```
// UIFont+scaleFactor.swift
import UIKit

public extension UIFont {
    static func preferredFont(forTextStyle style:
        UIFont.TextStyle, scaleFactor: CGFloat) -> UIFont {
        let font = UIFont.preferredFont(forTextStyle: style)
        return font.withSize(font.pointSize * scaleFactor)
    }
}
```

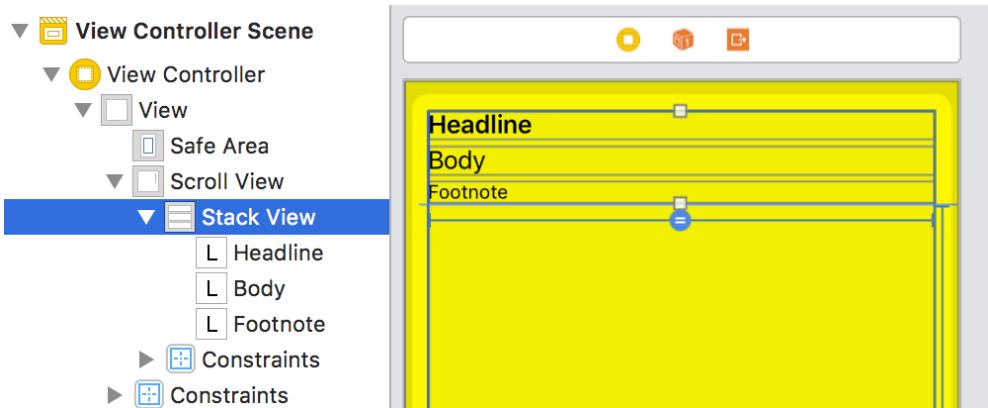


I recommend not making the font smaller than the sizes chosen by Apple. This is likely to annoy your users who expect a larger preferred content size to improve readability.

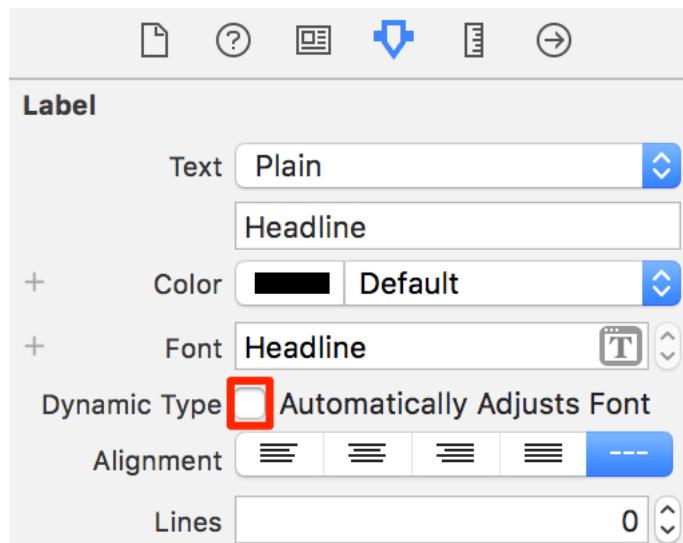
Let's use this in a layout built with Interface Builder (see sample code: [ScaledFont-v1](#)):

1. I have three labels in a vertical stack view. The labels use the Headline, Body and Footnote styles. To allow for the dynamic layout

size I set the number of lines of each label to zero, and I embedded the stack view in a scroll view:



2. Since we want to control the font used don't select the "Automatically Adjusts Font" option for the labels in Interface Builder:



3. Connect three outlets in the view controller to the three labels in the storyboard:

```
@IBOutlet var headline: UILabel!
@IBOutlet var body: UILabel!
@IBOutlet var footnote: UILabel!
```

4. We can then use our convenience method to configure the font for each label:

```
private func configureView() {
    headline?.font = UIFont.preferredFont(forTextStyle:
        .headline, scaleFactor: scaleFactor)
    body?.font = UIFont.preferredFont(forTextStyle: .body,
        scaleFactor: scaleFactor)
    footnote?.font = UIFont.preferredFont(forTextStyle:
        .footnote, scaleFactor: scaleFactor)
}
```

Note the optional chaining when accessing the labels in case the Interface Builder outlets are not yet connected.

5. The scale factor is a property of the view controller. I gave it a default value of 2 and call the `configureView` method if we ever set a new value:

```
var scaleFactor: CGFloat = 2.0 {
    didSet {
        configureView()
    }
}
```

6. We then make sure to configure our labels from `viewDidLoad`:

```
override func viewDidLoad() {
    super.viewDidLoad()
    configureView()
}
```

7. Finally we need to observe the content size category did change notification so we can reconfigure our labels if the user changes their preferred font size. Add the observer in `viewDidLoad`:

```
NotificationCenter.default.addObserver(self, selector:
    #selector(didChangePreferredFont(_:)), name:
    UIContentSizeCategory.didChangeNotification, object:
    nil)
```

8. Then when the notification arrives we call `configureView` to update the fonts:

```
@objc private func didChangePreferredFont(_ notification:
    Notification) {
    configureView()
}
```

9. Here's how the labels look at the default (Large) content size with a scale factor of 1.0:



Here's how they look with a scale factor of 2.0 (twice normal size):



Custom Fonts With Dynamic Type

What if you want to use dynamic type but don't want to use the Apple default San Francisco system font?

Before iOS 11 to use a custom font with dynamic type, you had to decide the font details (like the font face and size) for each of the text styles and then decide how to scale those font choices for each of the content size categories. You then listen for the content size category change notification and update your user interface with the right fonts.

A good starting point when making your font choices is to copy Apple. They publish the font metrics they use for the San Francisco typeface in the iOS Human Interface Guidelines:

- <https://developer.apple.com/design/human-interface-guidelines/ios/visual-design/typography/>

For example, at the default Large content size, the Headline style uses a Semi-Bold face of 17 points increasing to 23 points at the XXXLarge size and dropping to 14 points at the XSmall size.

Font Metrics

Apple introduced `UIFontMetrics` in iOS 11 to make it easier to use a custom font with dynamic type. To use a custom font with one of the text styles you first get a font metrics object for that style and then use it to scale your custom font (sized for the default Large content size) for the users preferred content size.

```
let fontMetrics = UIFontMetrics(forTextStyle: .body)
let myFont = UIFont(name: fontName, size: largeFontSize)
label.font = fontMetrics.scaledFont(for: myFont)
label.adjustsFontForContentSizeCategory = true
```

As long as you remember to set the `adjusts font` property the custom font scales automatically just like the system font.

Creating A Style Dictionary

The `UIFontMetrics` class takes care of scaling your custom font for each of the twelve content size categories. You do still need to decide on a font for each style at the default content size. This font size is then scaled by the font metrics when the user changes the content size.

To avoid having font face names and sizes scattered through the code, I like to use a style dictionary that has the face name and size to use for each of the text styles at the Large content size. To make it easy to customize and even change typefaces, I keep this style dictionary in a plist file.

Here's how it looks for the Noteworthy typeface which Apple bundles with iOS. It has both a bold and a light face:

Key	Type	Value
▼ Root	Dictionary	(11 items)
► UICTFontTextStyleTitle0	Dictionary	(2 items)
► UICTFontTextStyleTitle1	Dictionary	(2 items)
► UICTFontTextStyleTitle2	Dictionary	(2 items)
► UICTFontTextStyleTitle3	Dictionary	(2 items)
▼ UICTFontTextStyleHeadline	Dictionary	(2 items)
fontName	String	Noteworthy-Bold
fontSize	Number	17
► UICTFontTextStyleSubhead	Dictionary	(2 items)
▼ UICTFontTextStyleBody	Dictionary	(2 items)
fontName	String	Noteworthy-Light
fontSize	Number	17
► UICTFontTextStyleCallout	Dictionary	(2 items)
► UICTFontTextStyleFootnote	Dictionary	(2 items)
► UICTFontTextStyleCaption1	Dictionary	(2 items)
► UICTFontTextStyleCaption2	Dictionary	(2 items)

I followed the font sizes that Apple uses for the Large text size for each of the text styles. For example, I used a 17 point Noteworthy-Bold font for the Headline style and a 17 point Noteworthy-Light for the Body style.

To apply the fonts, I wrap the style dictionary in a `ScaledFont` struct that you initialize with the name, and optionally the bundle, of the `plist` file without the extension. The `font(forTextStyle:)` method returns the scaled font for a text style:

```
public struct ScaledFont {
    public init(fontName: String, in bundle: Bundle = default)
    public func font(forTextStyle textStyle: UIFont.TextStyle)
        -> UIFont
}
```

I use the Swift Decodable protocol and `PropertyListDecoder` class to read the dictionary from the `plist` file:

```
public struct ScaledFont {
    private struct FontDescription: Decodable {
        let fontSize: CGFloat
        let fontName: String
    }
}
```

```

private typealias StyleDictionary =
    [UIFont.TextStyle.rawValue: FontDescription]
private var styleDictionary: StyleDictionary?

public init(fontName: String, in bundle: Bundle =
    Bundle.main) {
    if let url = bundle.url(forResource: fontName,
        withExtension: "plist"),
        let data = try? Data(contentsOf: url) {
        let decoder = PropertyListDecoder()
        styleDictionary = try?
            decoder.decode(StyleDictionary.self, from: data)
    }
}
}

```

The method that returns the scaled font for a text style:

```

public func font(forTextStyle textStyle: UIFont.TextStyle) ->
    UIFont {
    guard let fontDescription =
        styleDictionary?[textStyle.rawValue],
        let font = UIFont(name: fontDescription.fontName, size:
            fontDescription.fontSize) else {
            return UIFont.preferredFont(forTextStyle: textStyle)
        }

    let fontMetrics = UIFontMetrics(forTextStyle: textStyle)
    return fontMetrics.scaledFont(for: font)
}

```

This method gets the font name and size for the given text style from the style dictionary and creates the font object. It falls back to the system font if the text style is missing from the dictionary. Then it uses the font metrics for the text style to scale the custom font for the content category size.

An example of using this to set the font of a label:

```

let scaledFont = ScaledFont(fontName: "Noteworthy")
bodyLabel.font = scaledFont.font(forTextStyle: .body)

```

A Practical Example

Let's use the style dictionary I created for the Noteworthy typeface to recreate our three label text example from earlier in the chapter (see

sample code: [CustomFont-v1](#)):

1. I'm going to create my layout in code so I added three properties for the labels to the view controller:

```
private let titleLabel = UILabel.makeLabel(style:  
    .headline)  
private let textLabel = UILabel.makeLabel(style: .body)  
private let footnoteLabel = UILabel.makeLabel(style:  
    .footnote)
```

2. I'm using a small extension on `UILabel` to create the multi-line labels with a text style:

```
// UILabel+Style.swift  
import UIKit  
extension UILabel {  
    static func makeLabel(style: UIFont.TextStyle) ->  
        UILabel {  
            let label = UILabel()  
            label.font = UIFont.preferredFont(forTextStyle: style)  
            label.adjustsFontForContentSizeCategory = true  
            label.numberOfLines = 0  
            return label  
        }  
}
```

3. I'm using the now familiar layout with my three labels in a vertical stack view which I then embed in a scroll view. See the earlier examples in this chapter for the details.
4. I added a font name property to the view controller that when set creates a new scaled font from the plist file which I added to the application target bundle:

```
var fontName = "Noteworthy" {  
    didSet {  
        scaledFont = ScaledFont(fontName: fontName)  
        configureFont()  
    }  
}  
  
private lazy var scaledFont: ScaledFont = {  
    return ScaledFont(fontName: fontName)  
}()
```

5. I call `configureFont` method from `viewDidLoad` and when the font

name changes to set the font for each of the three labels:

```
private func configureFont() {  
    titleLabel.font = scaledFont.font(forTextStyle:  
        .headline)  
    textLabel.font = scaledFont.font(forTextStyle: .body)  
    footnoteLabel.font = scaledFont.font(forTextStyle:  
        .footnote)  
}
```

6. Here's how it looks at the XXXLarge content size:

Chapter I

My father's family name being Pirrip,
and my Christian name Philip, my infant
tongue could make of both names
nothing longer or more explicit than Pip.
So, I called myself Pip, and came to be
called Pip.

Great Expectations by Charles Dickens

Adding A Custom Font To Your App

We don't have to restrict our font choices to those included by Apple with iOS. The <https://fonts.google.com> directory has a wide choice of open source fonts. For example, this is the Noto Serif font:

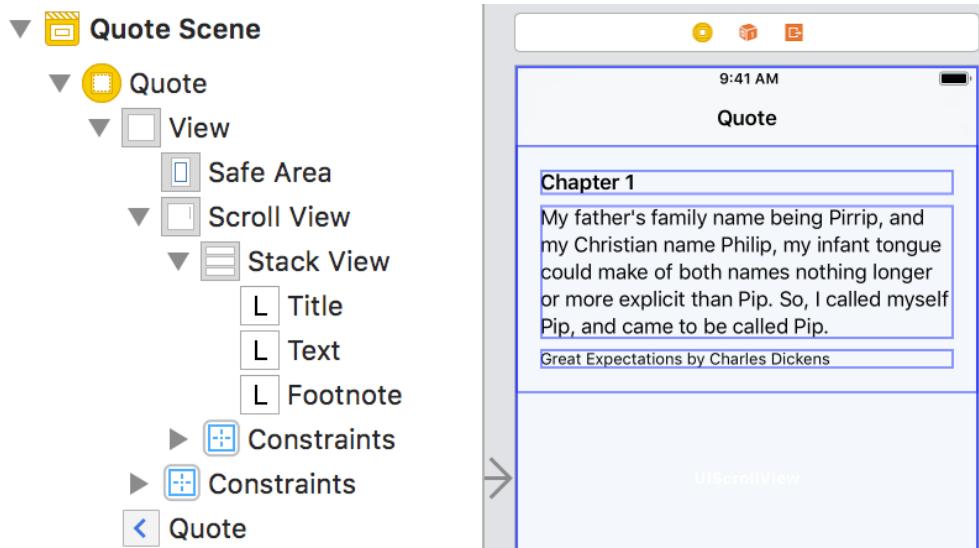
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
1 2 3 4 5 6 7 8 9 0



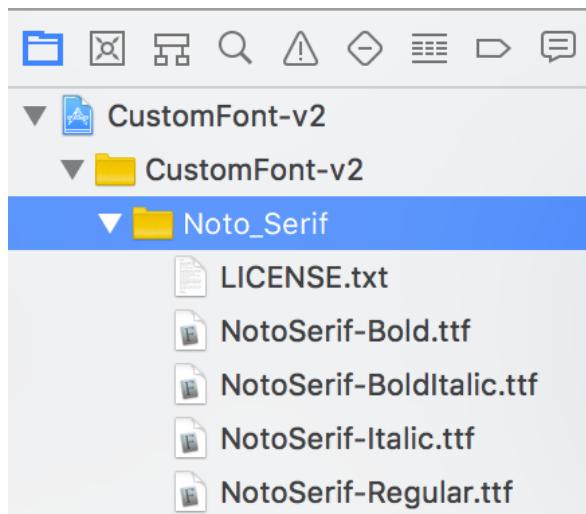
Check the license of any fonts you download to use with your App. When you use commercial fonts from sources other than Google Fonts you usually need to buy a license to embed them in your App.

Let's rebuild our dynamic type example using the Noto Serif font (see sample code: [CustomFont-v2](#)):

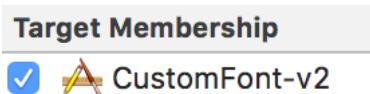
1. This time I'm going to start with the Interface Builder version of our layout. See the earlier examples in this chapter for the details:



2. Download the Noto Serif font from Google fonts and copy the folder into the Xcode project:



3. Select each of the files and use the File Inspector to make sure you added it to the project target:



4. Add entries for each of the font files to the `UIAppFonts` key ("Fonts provided by application") in the `Info.plist` file for the target:

▼ Fonts provided by application		Array	▼ (4 items)
Item 0	String	NotoSerif-Regular.ttf	
Item 1	String	NotoSerif-Bold.ttf	
Item 2	String	NotoSerif-Italic.ttf	
Item 3	String	NotoSerif-BoldItalic.ttf	

5. When creating a font object for our custom font, we need to know the font name. Unfortunately, this doesn't always match the file name. To get the list of loaded fonts and find their font names run this code snippet from the application delegate to dump all available fonts:

```
let families = UIFont.familyNames
families.sorted().forEach {
    print("\($0)")
    let names = UIFont.fontNames(forFamilyName: $0)
    print(names)
}

...
Noto Serif
["NotoSerif", "NotoSerif-BoldItalic", "NotoSerif-Bold",
 "NotoSerif-Italic"]
...
```

6. We can now create our style dictionary starting from the template we used for the Noteworthy font:

Key	Type	Value
▼ Root	Dictionary	+(11 items)
▶ UICTFontTextStyleTitle0	Dictionary	(2 items)
▶ UICTFontTextStyleTitle1	Dictionary	(2 items)
▶ UICTFontTextStyleTitle2	Dictionary	(2 items)
▶ UICTFontTextStyleTitle3	Dictionary	(2 items)
▼ UICTFontTextStyleHeadline	Dictionary	(2 items)
fontName	String	NotoSerif-Bold
fontSize	Number	17
▶ UICTFontTextStyleSubhead	Dictionary	(2 items)
▼ UICTFontTextStyleBody	Dictionary	(2 items)
fontName	String	NotoSerif
fontSize	Number	17
▶ UICTFontTextStyleCallout	Dictionary	(2 items)
▶ UICTFontTextStyleFootnote	Dictionary	(2 items)
▶ UICTFontTextStyleCaption1	Dictionary	(2 items)
▶ UICTFontTextStyleCaption2	Dictionary	(2 items)

7. In our view controller I have three outlets connected to the labels created in Interface Builder:

```
@IBOutlet private var titleLabel: UILabel!
@IBOutlet private var textLabel: UILabel!
@IBOutlet private var footnoteLabel: UILabel!
```

8. I added my ScaledFont struct to the project and created an instance in the view controller together with the font name (I left the default as Noteworthy):

```
var fontName = "Noteworthy" {
    didSet {
        scaledFont = ScaledFont(fontName: fontName)
        configureFont()
    }
}

private lazy var scaledFont: ScaledFont = {
    return ScaledFont(fontName: fontName)
}()
```

9. I change the default font of the view controller to “NotoSerif” in the application delegate:

```
quoteViewController.fontName = "NotoSerif"
```

- Finally, we configure the fonts for the three labels when the view is first loaded:

```
override func viewDidLoad() {
    super.viewDidLoad()
    ...
    configureFont()
}

private func configureFont() {
    titleLabel?.font = scaledFont.font(forTextStyle:
        .headline)
    textLabel?.font = scaledFont.font(forTextStyle: .body)
    footnoteLabel?.font = scaledFont.font(forTextStyle:
        .footnote)
}
```

- Here's how the layout looks using NotoSerif at the XXXLarge content size:

Chapter 1

My father's family name being Pirrip, and my Christian name Philip, my infant tongue could make of both names nothing longer or more explicit than Pip. So, I called myself Pip, and came to be called Pip.

Great Expectations by Charles Dickens

Key Points To Remember

Please consider adding support for dynamic type to your App:

- Supporting dynamic type puts the user in control of the size of the text in your application. Make your users happy to use your App by respecting their preferred text size.
- Be prepared. Using Auto Layout helps but the size of your text content can change dramatically when the user chooses the larger text sizes. When testing your layout with dynamic type don't forget about the five extra large accessibility sizes.
- Use multi-line text labels or text views when using a dynamic type text style. Avoid truncating text or shrinking font sizes to make content fit. Allow the user to scroll the view if the content can grow larger than the screen.
- Starting in iOS 10 select the "Automatically Adjusts Font" checkbox in Interface Builder or set the property in code to have dynamic type automatically adjust when the user changes their preferred text size:

```
label.adjustsFontForContentSizeCategory = true
```

- If you use dynamic type text styles with iOS 9 or earlier you have to observe the content size category did change notification and use it to reset the fonts for your text labels.
- Long lines of text are hard to read. Constrain text to the readable content guide to avoid them.
- Starting in iOS 11 the `UIFontMetrics` class makes it easier to use and scale a custom font with dynamic type.

Test Your Knowledge

Get into the habit of using dynamic type with these challenges:

Challenge 12.1 Using Dynamic Type

In this first challenge build this text layout using dynamic type and readable content guides:

Book

Great Expectations

Charles Dickens

My father's family name being Pirrip, and my Christian name Philip, my infant tongue could make of both names nothing longer or more explicit than Pip. So, I called myself Pip, and came to be called Pip.

1. The layout is embedded in a navigation controller with a standard amount of vertical spacing between the views and a 20 point margin around the views.
2. Use a multi-line text label with the “Title 1” text style for the title.
3. Use a multi-line text label with the “Headline” text style for the author label.
4. Use a text view with the “Body” text style for the text to allow word selection (but not editing).
5. The text should respect the readable content guide so that the line lengths stay readable even on larger devices.
6. The text sizes should automatically update when the user changes their preferred text content sizes. The text should scroll if it becomes too big to fit on the screen.

Hints And Tips

1. The approach we have seen several times in this chapter of a stack view embedded in a scroll view would work well for this layout. You can also do it without the stack view if you prefer.
2. Remember to set the number of lines to zero for the labels and disable scrolling and editing for the text view.

3. Remember to configure the labels and text view to automatically adjust their fonts when the user changes their preferred text content size.
4. If you're building the layout with Interface Builder, you need to set "Follow Readable Width" on the stack view. If you're building the layout in code, constrain the stack view to the readable content guide of a container view.
5. See [Text Views](#) for a reminder on how to remove the extra padding used by the text view.

Challenge 12.2 Changing A Text Style

Here's how the default Subheadline style looks in comparison to the Headline style:

Headline text style

Subheadline text style

I sometimes like to make the Subheadline italic to make it more distinctive. Your challenge is to figure out how to change the default text styles so that the Subheadline style appear italic:

Headline text style

Subheadline text style

1. Build a layout with two multi-line labels that use the Headline and Subheadline text styles. It doesn't matter if you use Interface Builder or build the layout in code.
2. Using the techniques we saw in [Scaling Dynamic Type](#) change the font for the Subheadline style to make it appear italic.
3. Don't change the Headline style and make sure the text styles still respond to changes in the user's preferred content size.

Hints And Tips

1. Don't select "Automatically Adjusts Font" for the Subheadline label.
2. Add an observer to the view controller for the content size category did change notification.
3. Use the method that handles the notification to configure the font of the Subheadline label.
4. You also need to configure the font when the view controller loads its view (from viewDidLoad).
5. To configure the font, you need the preferred font descriptor for the Subheadline text style.
6. Create a new descriptor based on the original preferred descriptor with the added symbolic trait for italic text.
7. Create a new italic version of the Subheadline font using the new descriptor. Don't change the font size.

Challenge 12.3 Using A Custom Font

Use this challenge to experiment with using a custom font with dynamic type. I'm using the first few paragraphs from "The Wonderful Wizard of Oz" available from Project Gutenberg (<https://www.gutenberg.org>), but you can use any text you want:

Book

The Wonderful Wizard of Oz

by L. Frank Baum

1. The Cyclone

Dorothy lived in the midst of the great Kansas prairies, with Uncle Henry, who was a farmer, and Aunt Em, who was the farmer's wife. Their house was small, for the lumber to build it had to be carried by wagon many miles. There were four walls, a floor and a roof, which made one room; and this room contained a rusty looking cookstove, a cupboard for the dishes, a table, three or four chairs, and the beds. Uncle Henry and Aunt Em had a big bed in one corner, and Dorothy a little bed in another corner. There was no garret at all, and no cellar—except a small hole dug in the ground, called a cyclone cellar, where the family could go in case one of those great whirlwinds arose, mighty enough to crush any building in its path. It was reached by a trap door in the middle of the floor, from which a ladder led down into the small, dark hole.

The text styles and fonts I used:

- Title 1 for the book title (Roboto-Bold)
- Subheadline for the author label (Roboto-LightItalic)
- Headline for the chapter label (Roboto-Bold)
- Body for the text (Roboto-Light)

Feel free to get creative and use different styles.

1. Download a font you like from Google Fonts (<https://fonts.google.com>) and add it to the App.
2. Your text should use dynamic type and resize when the user changes their preferred content size.
3. Your text should respect the readable content guides to avoid long lines that are hard to read.
4. The body text should be selectable.
5. The content should scroll vertically if necessary when it becomes too large to fit on the screen.

Hints And Tips

1. You can select any font you like, but I recommend finding one that has bold and italic styles. I used Roboto which has 12 styles though I only used three of them (Roboto-Light, Roboto-LightItalic, and Roboto-Bold).
2. Remember to add a custom font to your app you need to add the font files (.ttf) to the project target and list them in the targets Info.plist file. See [Adding A Custom Font To Your App](#) for details.
3. Use the ScaledFont struct I showed in [Custom Fonts With Dynamic Type](#) to define the font weight and size to use for each of the text styles. Start with the Noteworthy.plist style dictionary and change it for the custom font you're using.
4. You can build the user interface in code or with Interface Builder. Use a combination of multi-line labels and text views embedded in a stack view embedded in a scroll view. Make sure to disable editing and scrolling for the text view.
5. Remember to set the “Automatically Adjusts Font” property.

Chapter 13

Working With Table Views

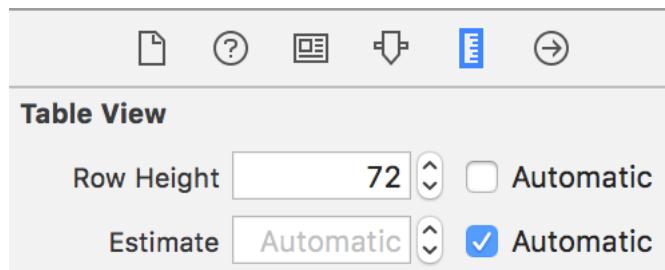
To learn how to create table view based user interfaces see the prerequisite resources in the [Introduction](#). In this chapter I'm going to focus on a couple of topics essential for building adaptive layouts with table views:

- How to create self-sizing table view cells
- How to use readable content guides with table views.

Self-Sizing Table View Cells

When you create the layout for a table view cell, you're working with a view that has its width set by the table view. You have several ways to set the row height of the cell:

- Set a default row height for all cells in the table view. You can do this in Interface Builder at the table view level using the size inspector:



If you build your layout in code you set the property directly on the table view:

```
tableView.rowHeight = 72.0
```

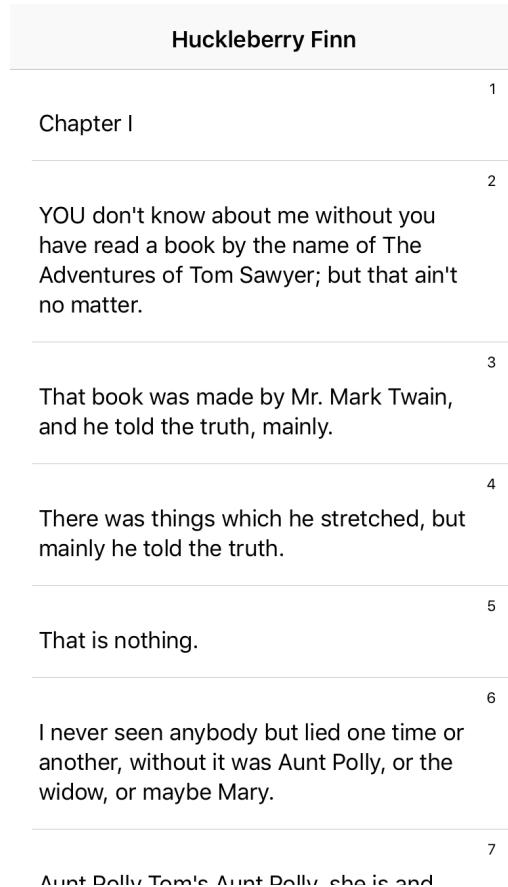
- You can use the `heightForRowAt` table view delegate method to return a row height:

```
override func tableView(_ tableView: UITableView,  
    heightForRowAt indexPath: IndexPath) -> CGFloat {  
    // Calculate row height  
    let height: CGFloat = ...  
    return height  
}
```

This allows you to return a different height for each row but at the expense of the table view having to call the method for each of its rows every time you display or reload the table view.

- You can use Auto Layout to create a self-sizing cell that adjusts its height to fit the content.

All three approaches have their uses, but when it comes to building an adaptive layout, the self-sizing cell is the most flexible. By sizing the cell height at runtime to fit the content, it also copes with dynamic type content size changes and different device sizes. For example, here's the table view layout I'm going to build:



The cell layout has a label showing the line number and a text view showing some text from the book Huckleberry Finn. The line number is using the Caption 2 text style, and the text view is using the Body text style. Note how the heights of each cell vary to fit the content. Here's how the cells look when I increase the dynamic text size:

Huckleberry Finn	
1	Chapter I
2	YOU don't know about me without you have read a book by the name of The Adventures of Tom Sawyer; but that ain't no matter.
3	That book was made by Mr. Mark Twain, and he told the truth, mainly.
4	There was things which he stretched, but mainly he <small>told the truth</small>

There are three steps to creating a self-sizing table view cell:

1. Any views you add to the table view cell should be subviews of the content view of the cell. Use Auto Layout to fully constrain your views to the content view, not the cell itself.
2. Set the table view row height to automatic.
3. Set a non-zero value for the estimated row height (or use automatic).

The value to use for the estimate is not always obvious. If you have cells that are all roughly the same height you can use an average but what if your cell height varies dramatically? What value should you use with dynamic type? If there's no obvious estimate, use the default value of automatic and have the table view choose a value for you.

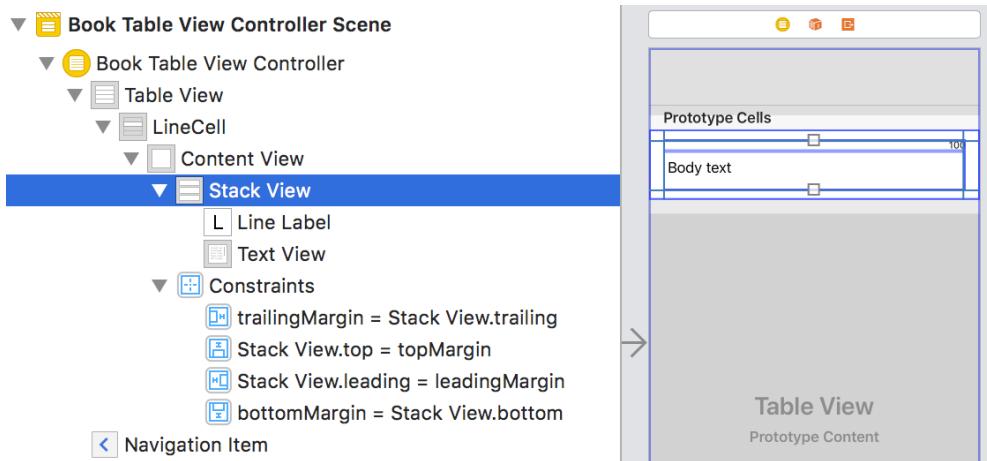
Let's look at three different ways to create our self-sizing table view cell:

- Using a storyboard
- Using a NIB file for the cell
- Creating a custom cell layout in code

Creating Table View Cells With A Storyboard

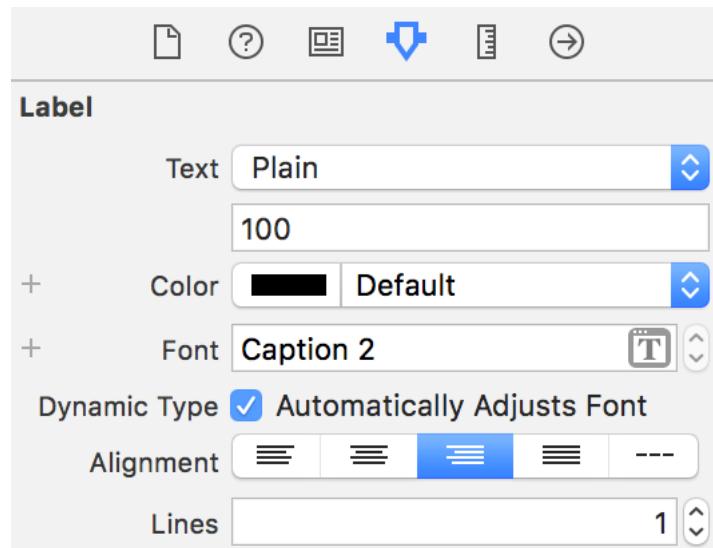
First let's create our table view cell with Interface Builder in a storyboard (see sample code: [SelfSizing-v1](#)):

1. I'm starting with a new Xcode project using a storyboard containing a table view controller embedded in a navigation controller. The table view uses a single custom cell.
2. Here's the table view controller in my storyboard showing the layout of my custom table view cell:

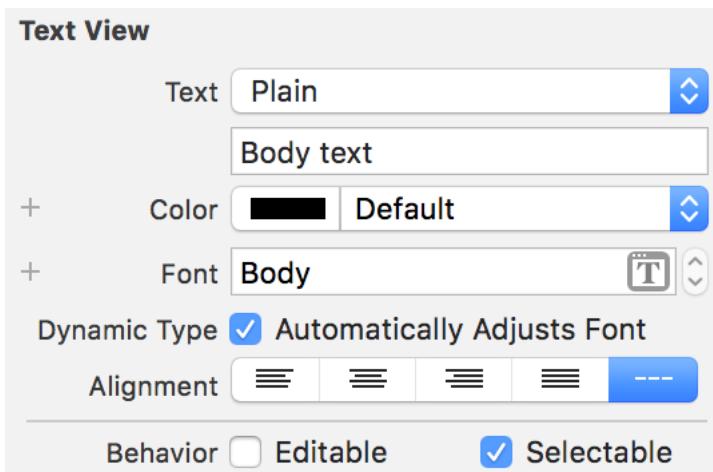


I embedded a label and text view in a vertical stack view that's a subview of the content view of the table view cell.

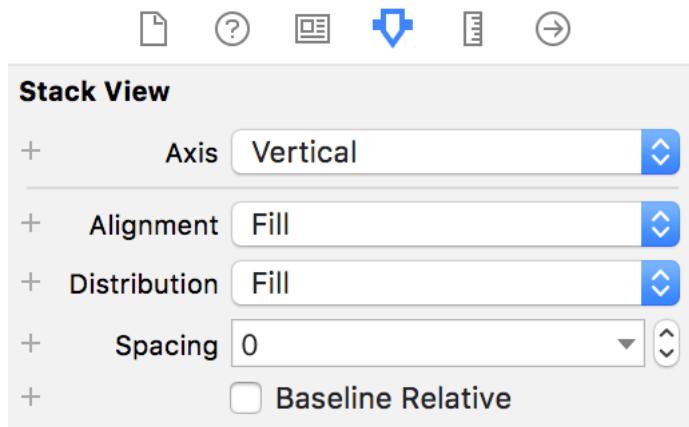
3. The line label is right aligned and uses the Caption 2 text style:



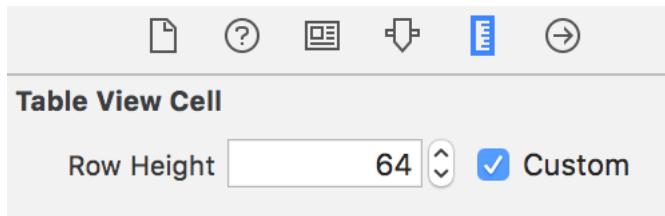
4. The text view uses the Body text style. I also disabled editing and scrolling (not shown) which are enabled by default:



5. The stack view is vertical with .fill alignment and distribution and zero spacing. The constraints are pinning the stack view to the margins of the **content view**, not the table view cell.



- To make it easier to layout the cell I increased the default height of the custom cell to 64 points:



Depending on how big you made the table view cell and the size of the content you may see errors from Interface Builder suggesting you change the content priorities of the label or text view:

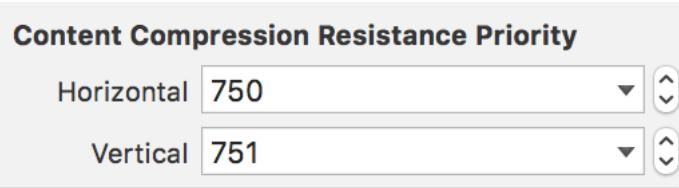
▼ Content Priority Ambiguity ✖

Text View Set vertical compression resistance priority to 749

Label Line Label Set vertical compression resistance priority to 751

This is confusing. Interface Builder doesn't know that this is a self-sizing cell so is warning us that the row height set in Interface Builder is squeezing the content.

- The easiest way to remove the error is to increase the row height of the custom cell, so it's no longer squeezing the contents. You can also follow the Interface Builder suggestion to increase the vertical compression priority of either the label or text view to remove the ambiguity. For example, increase the text view priority to 751:

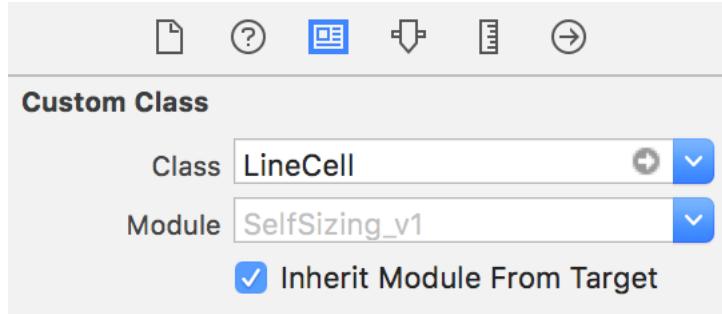


It doesn't matter which of these workarounds you use to stop Interface Builder complaining. The row height is set at runtime to fit the cell contents without stretching or squeezing either view.

8. I created a subclass of UITableViewCell named LineCell which contains outlets to the label and text view in the storyboard:

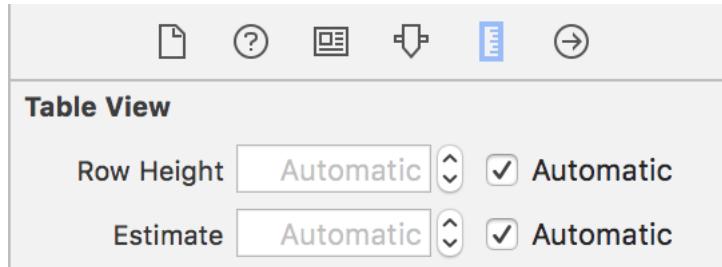
```
// LineCell.swift
import UIKit
final class LineCell: UITableViewCell {
    @IBOutlet var lineLabel: UILabel!
    @IBOutlet var textView: UITextView!
}
```

Remember to set the class of the table view cell in the storyboard:



See the sample code for details on how the table view data source populates the table view cell.

9. To make the table view self-sizing we need to set the row height to automatic and set an estimated row height. You can set both properties using the size inspector on the table view:



Both properties default to automatic. Since I'm using dynamic type and I don't have a better guess for the row height, I'll leave the estimate at automatic.

10. The Apple documentation for row height tells you to explicitly set the row height to automatic in code when creating the table view cell in Interface Builder. This is to work around a problem where setting a custom row height in Interface Builder would override the default automatic value. I don't find this to be necessary with the latest versions of Xcode but to be sure you can do this from viewDidLoad in the table view controller:

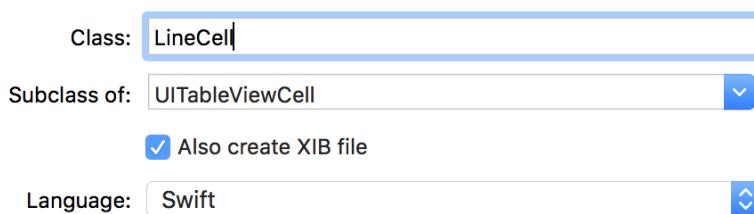
```
override func viewDidLoad() {  
    super.viewDidLoad()  
    tableView.rowHeight = UITableView.automaticDimension  
}
```

11. Build and run. You should have a table view with cells that automatically adjust their height to fit the content.

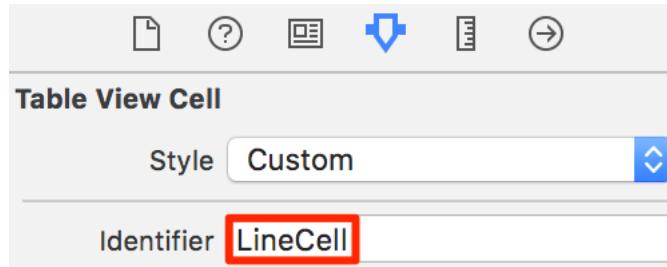
Creating Table View Cells With A NIB

Using a NIB to create the layout for your table view cell allows you to reuse it in more than one table view. The steps to create a self-sizing table view cell with a NIB are more or less the same as for using a storyboard (see sample code: [SelfSizing-v2](#)):

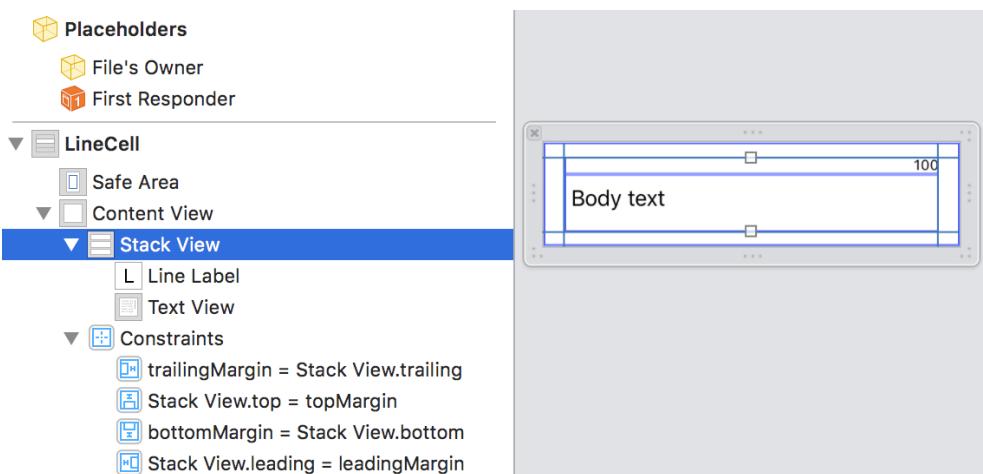
1. Start with an Xcode project containing a storyboard with a table view controller embedded in a navigation controller. This time don't create a prototype cell in the table view as we'll create it in a separate NIB file.
2. Add a new file to your Xcode project File > New > File... and choose "Cocoa Touch Class" from the iOS templates. Make the new class a subclass of UITableViewCell, name the class and select "Also create XIB file":



3. Open the XIB file in Interface Builder and set the cell identifier:



4. As with the storyboard example, you may find it easier to increase the default size of the table view cell. Drag the bottom of the cell down to increase the height of the cell in the canvas until you have enough room for the label and text view. The exact height doesn't matter as the size is set at runtime so use whatever size you like to make it easy to layout the cell.
5. Drag a label and text view from the object library into the table view content view. Configure them as we did for storyboard example. The label uses the Caption 2 text style and is right aligned. The text view uses the Body text style.
6. Don't forget to disable scrolling for the text view and enable Automatically Adjusts Font for both items. Embed them in a vertical stack view with .fill distribution and alignment and zero spacing. Add four constraints to pin the stack view to the margins of the content view:



7. As with the storyboard example create connections from the label and text view in Interface Builder to outlets in the table view cell subclass:

```
// LineCell.swift
import UIKit
final class LineCell: UITableViewCell {
    @IBOutlet var lineLabel: UILabel!
    @IBOutlet var textView: UITextView!
}
```

- Finally, in the `viewDidLoad` method of the table view controller we register our NIB file with the table view for the cell reuse identifier we used in the NIB file:

```
override func viewDidLoad() {
    super.viewDidLoad()
    let nib = UINib(nibName: "LineCell", bundle: nil)
    tableView.register(nib, forCellReuseIdentifier:
        "LineCell")
    tableView.rowHeight = UITableView.automaticDimension
}
```

Creating Table View Cells In Code

You don't have to create your table view cells with Interface Builder. You can create a table view cell subclass entirely in code (see sample code: [SelfSizing-v3](#)):

- Follow the same setup as in the last two examples but this time don't create the table view cell with Interface Builder. Add a new Swift file to your Xcode project for the `UITableViewCell` subclass:

```
// LineCell.swift
import UIKit
class LineCell: UITableViewCell {
```

- Add a property for the line label and configure it with the `Caption 2` style:

```
let lineLabel: UILabel = {
    let label = UILabel()
    label.font = UIFont.preferredFont(forTextStyle:
        .caption2)
    label.adjustsFontForContentSizeCategory = true
    label.textAlignment = .right
    return label
}()
```

3. Add a second property for the text view configured to use the Body text style with editing and scrolling disabled:

```
let textView: UITextView = {
    let view = UITextView()
    view.font = UIFont.preferredFont(forTextStyle: .body)
    view.adjustsFontForContentSizeCategory = true
    view.setEditable = false
    view.isScrollEnabled = false
    return view
}()
```

4. Create and configure a stack view to manage the layout of the label and text view:

```
private lazy var stackView: UIStackView = {
    let stackView = UIStackView(arrangedSubviews:
        [lineLabel, textView])
    stackView.translatesAutoresizingMaskIntoConstraints =
        false
    stackView.axis = .vertical
    return stackView
}()
```

5. We need to implement the UITableViewCell designated initializer and the required initializer for UIView:

```
override init(style: UITableViewCell.CellStyle,
    reuseIdentifier: String?) {
    super.init(style: style, reuseIdentifier:
        reuseIdentifier)
    setupView()
}

required init?(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
    setupView()
}
```

I follow the usual approach of moving the view setup to a private `setupView` method.

6. In `setupView` add the stack view to the content view of the cell:

```
private func setupView() {
    contentView.addSubview(stackView)
```

7. Then add constraints between the stack view and the layout margin guide of the content view:

```
let margins = contentView.layoutMarginsGuide
NSLayoutConstraint.activate([
    margins.leadingAnchor.constraint(equalTo:
        stackView.leadingAnchor),
    margins.trailingAnchor.constraint(equalTo:
        stackView.trailingAnchor),
    margins.topAnchor.constraint(equalTo:
        stackView.topAnchor),
    margins.bottomAnchor.constraint(equalTo:
        stackView.bottomAnchor)
])
}
```

8. The setup of the table view controller is mostly the same as before except this time we need to register our custom subclass with the table view:

```
override func viewDidLoad() {
    super.viewDidLoad()
    tableView.register(LineCell.self,
        forCellReuseIdentifier: "LineCell")
    tableView.rowHeight = UITableView.automaticDimension
}
```

Readable Table Views

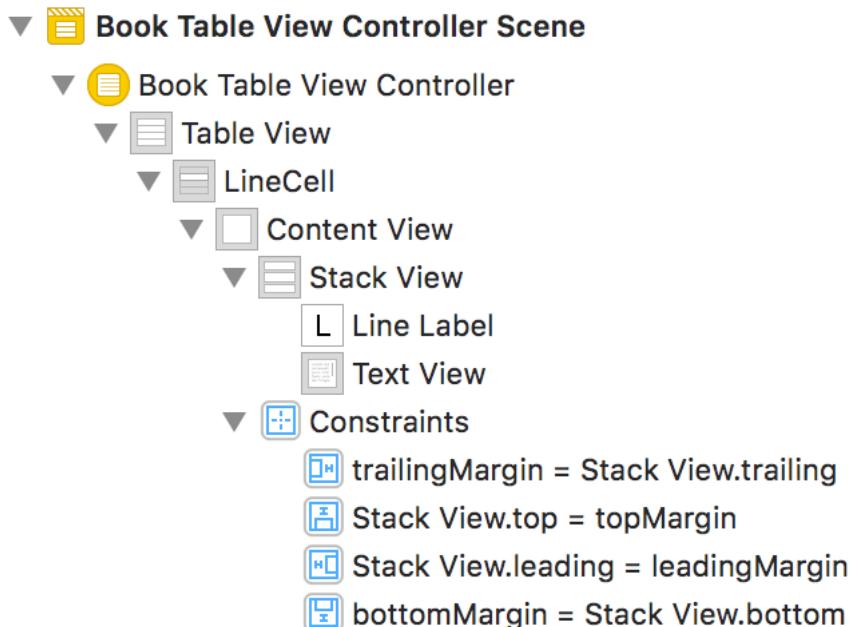
Take a look at our table view layout when running on a 10.5" iPad Pro:

Huckleberry Finn	
1	
Chapter I	
2	YOU don't know about me without you have read a book by the name of The Adventures of Tom Sawyer; but that ain't no matter.
3	That book was made by Mr. Mark Twain, and he told the truth, mainly.
4	There was things which he stretched, but mainly he told the truth.
5	That is nothing.
6	I never seen anybody but lied one time or another, without it was Aunt Polly, or the widow, or maybe Mary.
7	

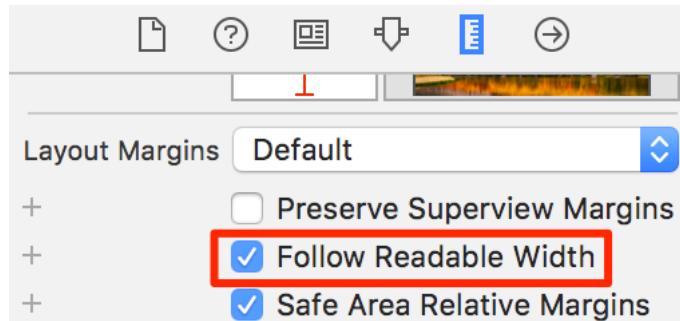
In portrait, our lines are over 100 characters long at the default dynamic type size. As we saw when looking at [Readable Content Guides](#) long lines of text are difficult to read. Luckily we can adjust our table view layouts to make use of readable content guides.

Readable Width With Custom Table View Cells

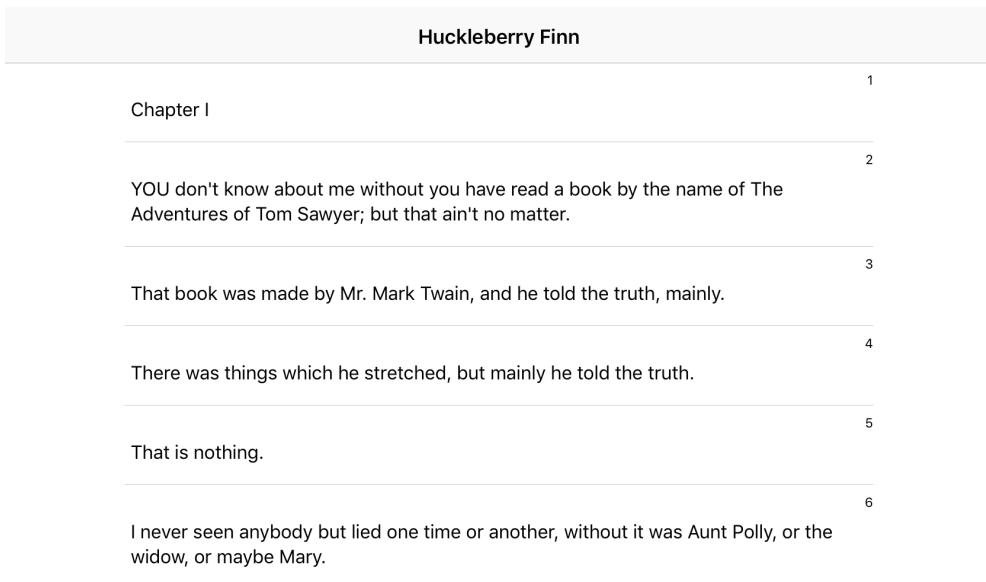
When using a custom table view cell with Interface Builder, you create constraints between the subviews and the cell content view. Here's the layout from the line cell we used earlier:



Note that I constrained the stack view to the margins of the cell content view. To make these margins follow the readable width select the content view and use the size inspector to select the “Follow Readable Width” option, Interface Builder disables it by default (see sample code: [ReadableTable-v1](#)):



You can, if you prefer, also set the option on the table view where it applies to all cell types defined in the storyboard scene. Here's how the cell looks when using the readable width:



I prefer to avoid the built-in table view styles (Basic, Right Detail, Left Detail, Subtitle) and create a subclass of `UITableViewCell`. If you do use one of the built-in styles be aware that setting the “Follow Readable Width” option on the content view of the cell has no effect. You must set the option on the table view.

If you're creating your cell layout in code, you can set the table view property to have all cell layout margins for that table follow the readable width:

```
// default is false for iOS 12  
// default is true for iOS 9, 10, 11  
tableView.cellLayoutMarginsFollowReadableWidth = true
```

In iOS 12 this property matches the Interface Builder behavior by defaulting to false. Confusingly it was `true` by default in iOS 11 and earlier releases. Let's revisit our self-sizing table view example where we created the cell layout in code and fix it to follow the readable width (see sample code: [ReadableTable-v2](#)).

When creating the constraints between the stack view and the content view of the cell we used the layout margins guide:

```
let margins = contentView.layoutMarginsGuide  
NSLayoutConstraint.activate([  
    margins.leadingAnchor.constraint(equalTo:  
        stackView.leadingAnchor),  
    ...  
])
```

As with Interface Builder, there are two ways we can change this layout so that it follows the readable width.

1. Set the table view property to `true`. Remember this defaults to `true` in iOS 11 and earlier, but is `false` in iOS 12:

```
override func viewDidLoad() {  
    ...  
    tableView.cellLayoutMarginsFollowReadableWidth = true  
}
```

2. Alternatively, change the cell layout so that we use the readable content guide of the content view instead of the layout margin guide:

```
let margins = contentView.readableContentGuide  
NSLayoutConstraint.activate([  
    margins.leadingAnchor.constraint(equalTo:  
        stackView.leadingAnchor),  
    ...  
])
```

I prefer the second of these two options as it keeps the layout configura-

tion for the cell together.

Key Points To Remember

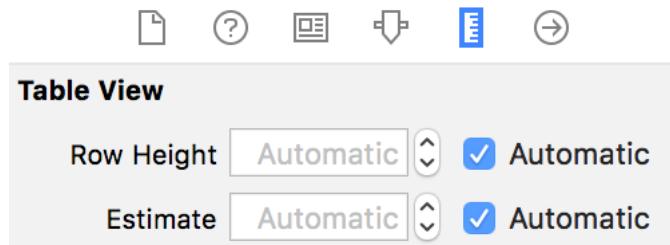
To create a self-sizing table view cell requires you to do three things:

1. Use Auto Layout to fully constrain your subviews to the content view of the cell and not the cell itself.
2. Set the table view row height to automatic in Interface Builder or in code:

```
tableView.rowHeight = UITableView.automaticDimension
```

3. Set a non-zero value for the estimated row height in Interface Builder or in code. If you're not sure what value to use keep the default value of automatic:

```
tableView.estimatedRowHeight =  
    UITableView.automaticDimension
```



If you're using Interface Builder and want your table view cells to follow the readable width make sure to layout your cells using the margins of the content view. You then have two options to have the cell follow the readable width:

1. The first option is to select the “Follow Readable Width” option for the table view to change the setting for all table view cells defined in that storyboard scene.
2. The second option is to select “Follow Readable Width” on the content view of the cell to change the setting just for that cell type (doesn't work for the built-in cell styles).

If you're creating your table view in code, you also have two options depending on how you layout your table view cell:

1. If you layout your cell using the layout margin guide of the content view set the table view property to true.

```
tableView.cellLayoutMarginsFollowReadableWidth = true
```

2. A second option is to layout your cell using the readable content guide of the content view.

Remember that the default for the table view property changed in iOS 12. It's now `false` by default. Be careful if you target iOS 11 or earlier where the default is `true`:

```
// default is false for iOS 12  
// true for earlier releases  
tableView.cellLayoutMarginsFollowReadableWidth = true
```

Test Your Knowledge

Practice building some self-sizing and readable table view cell layouts.

Challenge 13.1 Self-Sizing Table View Cell

Here's a layout for a simple messaging App. Shown on the left at the default text size and at the XXL size on the right:

Messages
 Joe Hello World!
 Peter Piper Mr. Watson I presume
 Harry Housefield Once upon a time, in a land far far away
 Sir Roger Martin-Grayson III I cannot wait to see what happens next. I do hope they all lived happily ever after.

Messages
 Joe Hello World!
 Peter Piper Mr. Watson I presume
 Harry Housefield Once upon a time, in a land far far away
 Sir Roger Martin-Grayson III I cannot wait to see what happens next. I do hope they all lived happily ever after.

1. Build this layout using a self-sizing table view cell.

2. Some points to note about the layout:
 - The profile placeholder image is 60x60 points.
 - The username label is using the Headline text style.
 - The message text is using the Body text style.
 - There's a standard amount of spacing between the labels and image and I pinned all three views to the default cell content margins.
3. You can build the cell layout using a storyboard, NIB or entirely in code.
4. The table-view cells should size to fit the content without clipping or truncation.
5. The text size (and the table view cell size) should update if the user changes their preferred text size.
6. As a bonus can you change your project to deploy back to iOS 9?

Hints And Tips

1. Feel free to populate the table view with your data. To keep it simple I hardcoded the message data in the data source:

```
struct Message {  
    let username: String  
    let text: String  
}  
  
private let messages: [Message] = [  
    Message(username: "Joe", text: "Hello World!"),  
    Message(username: "Peter Piper", text: "Mr. Watson I  
    presume"),  
    Message(username: "Harry Housefield", text: "Once upon a  
    time, in a land far far away"),  
    Message(username: "Sir Roger Martin-Grayson III", text:  
        "I cannot wait to see what happens next. I do hope they  
        all lived happily ever after.")  
]
```

2. You can build the table view cell layout with two stack views or with individual constraints. Either way, you need to adjust the content priorities for the views.
3. Don't forget to set the number of lines for the two labels to zero so they can wrap onto multiple lines as needed.

4. I'm using a placeholder template vector image that I added to the asset catalog. Create your own or grab mine from the solution code.
5. Use the content mode of the image view to keep the placeholder image at the top of the cell.
6. To support back to iOS 9 you cannot set "Automatically Adjusts Font" in Interface Builder. You need to listen for the content size change notification and reload the table view so you can update the font of the two labels in the data source method. See [Supporting Dynamic Type With iOS 9](#) for the details.
7. A reminder that dynamic type notifications don't work in the iOS 9 simulator.

Challenge 13.2 Readable Width Table View Cell

Take some lines of text from a project Gutenberg book (<https://www.gutenberg.org>) and show them in a table view one row per line. Here are the first few rows on a 10.5" iPad Pro in portrait:

Great Expectations
Chapter I
My father's family name being Pirrip, and my Christian name Philip, my infant tongue could make of both names nothing longer or more explicit than Pip.
So, I called myself Pip, and came to be called Pip.
I give Pirrip as my father's family name, on the authority of his tombstone and my sister,--Mrs. Joe Gargery, who married the blacksmith.
As I never saw my father or my mother, and never saw any likeness of either of them (for their days were long before the days of photographs), my first fancies regarding what they were like were unreasonably derived from their tombstones.
The shape of the letters on my father's, gave me an odd idea that he was a square, stout, dark man, with curly black hair.

1. Build this layout using a self-sizing table view cell that limits the text length to the readable width.
2. The text should use the Body text style and adjust its size when the user changes their preferred text size.
3. Use a text view for the text so that the user can select and look up words.

Hints And Tips

1. You can build this layout using a storyboard or in code as you prefer.
2. If you use Interface Builder remember to constrain the text view to the margins of the content view and set the “Follow Readable Width” option on the content view of your custom cell.
3. Remember to disable scrolling and editing for the text view.
4. Remember to set “Automatically Adjusts Font” for the text view.
5. If building the table view cell layout in code constrain the text view to the readable content guide of the content view. Make sure you add the text view to the content view and not directly to the table view cell.

Chapter 14

Adapting For Size

Apple introduced the concept of adaptive user interfaces in iOS 8 using a combination of Auto Layout, size classes, and adaptive view controller presentations. The aim is to make it easier to develop universal interfaces that scale from the smallest iPhone to the largest iPad.

Building user interfaces that adjust to changes in screen size became even more critical when Apple added the multi-tasking slide over and split-screen modes to the iPad in iOS 9.

In this chapter you learn some techniques for building layouts that adapt to make the best use of the available screen size:

- What interface trait collections and size classes are and how they vary for the different iPhone and iPad screen sizes.
- How to use Interface Builder to create variations of a view property, constraint or even a whole layout for different size classes.
- How to use the `UITraitEnvironment` protocol to build adaptive layouts in code and use the `UIContentContainer` protocol to animate changes to your layout.
- How to use the asset catalog to create images for specific size class variations.
- How to use a localization strings dictionary to show different strings based on the screen width.
- What to do when size classes are not enough.

Trait Collections

Before iOS 8 if you wanted to have a different user interface for the iPad version of an App you could check the `userInterfaceIdiom` of the device running your App:

```
if UIDevice.current.userInterfaceIdiom == .pad {  
    // iPad specific code  
}
```

That was fine when Apps always ran full-screen on either an iPhone or iPad. We now have more devices, different screen resolutions and color gamuts and properties like dynamic type and split-screen modes that can change at runtime.

Starting in iOS 8 Apple introduced the idea of a trait collection to describe the environmental properties (traits) of a user interface. You can check the trait collection of views, view controllers, and other classes that adopt the trait environment protocol (`UITraitEnvironment`) using their `traitCollection` property:

```
// UIScreen, UIWindow, UIViewController  
// UIView, UIPresentationController  
var traitCollection: UITraitCollection { get }
```

The `traitCollection` property is of type `UITraitCollection` and has the following trait properties:

- `horizontalSizeClass`: A `UIUserInterfaceSizeClass`. Possible values are `unspecified`, `compact`, `regular`. Default is `unspecified`.
- `verticalSizeClass`: A `UIUserInterfaceSizeClass`. Possible values are `unspecified`, `compact`, `regular`. Default is `unspecified`.
- `displayScale`: A `CGFloat` indicating the display scale where 0.0 is `unspecified`, 1.0 is non-Retina and 2.0 is retina.
- `displayGamut`: A `UIDisplayGamut` value. Possible values are `unspecified`, `SRGB` and `P3`. Available since iOS 10.
- `userInterfaceIdiom`: A `UIUserInterfaceIdiom`. Possible values are `unspecified`, `phone`, `pad`, `tv`, `carPlay`. Default is `unspecified`.
- `forceTouchCapability`: A `UIForceTouchCapability`. Possible values are `unknown`, `available`, `unavailable`. Note a user can disable force touch in the accessibility settings.

- `layoutDirection`: `UITraitEnvironmentLayoutDirection`. Possible values are `unspecified`, `leftToRight` and `rightToLeft`. Available since iOS 10.
- `userInterfaceStyle`: A `UIUserInterfaceStyle`. Possible values are `unspecified`, `light`, `dark`. Available for CarPlay and tvOS devices in iOS 12.
- `preferredContentSizeCategory`: A `UIContentSizeCategory`. See [Dynamic Type](#) for possible values. Available since iOS 10.



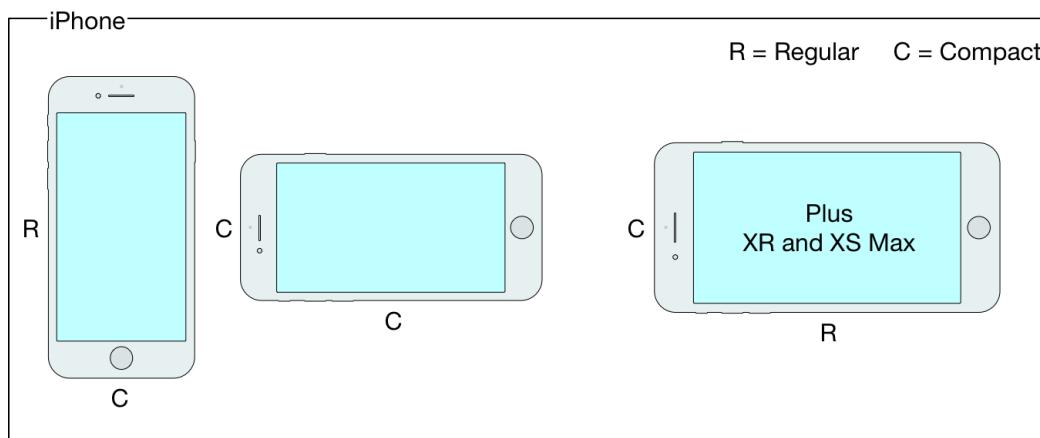
Many traits have a default value of `unspecified` when the system has not set a value because the object is not yet in the view hierarchy.

Size Classes

The two user interface traits you probably use the most when creating adaptive layouts are the horizontal and vertical size classes. There are two size classes that can apply to either the horizontal (width) or vertical (height) dimension of a user interface:

- `regular`: when the interface has *lots of space*.
- `compact`: when the interface has *limited space*.

That's a little vague. Here's how the size classes apply to the iPhone:



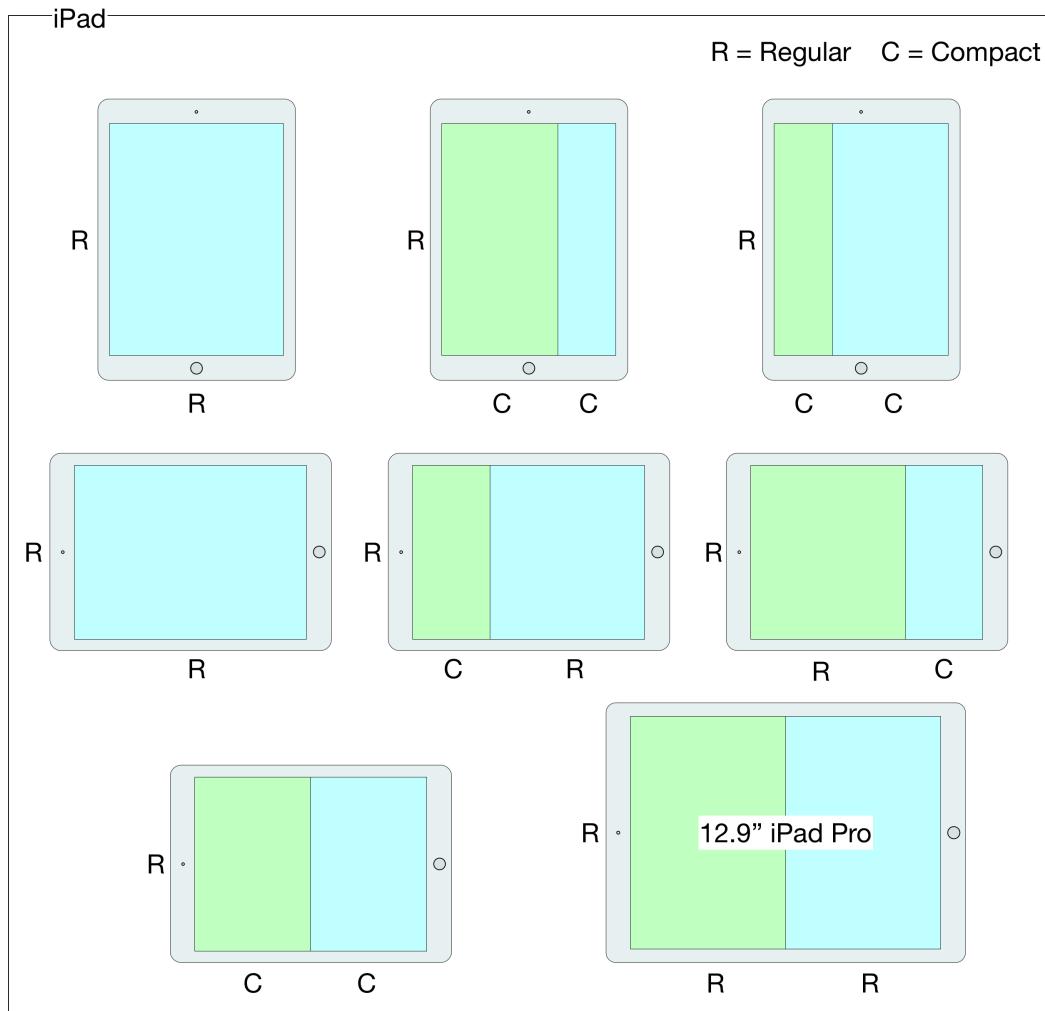
Notes:

- The iPhone 7/8/X/XS and iPhone SE all have a regular height and compact width in portrait but in landscape both dimensions are

compact.

- The iPhone Plus, XR and XS Max models differ in that they also have a regular width in landscape.

The iPad has more possibilities due to the multitasking split views introduced in iOS 9. These allow the user to split the screen to show two apps side-by-side with an app having 1/3, 1/2 or 2/3 of the screen width:



Notes:

- A full-screen iPad application always has regular height and regular width size classes regardless of orientation or device.
- The height/vertical size class is always regular even with a split view.
- In portrait the apps have different widths (1/3 and 2/3), but both have a compact horizontal size class.

- In landscape the 2/3 width gives a regular horizontal size class.
- In landscape the 50:50 split view gives both apps a compact horizontal size class except for the larger 12.9" iPad Pro where both apps have a regular horizontal size class.

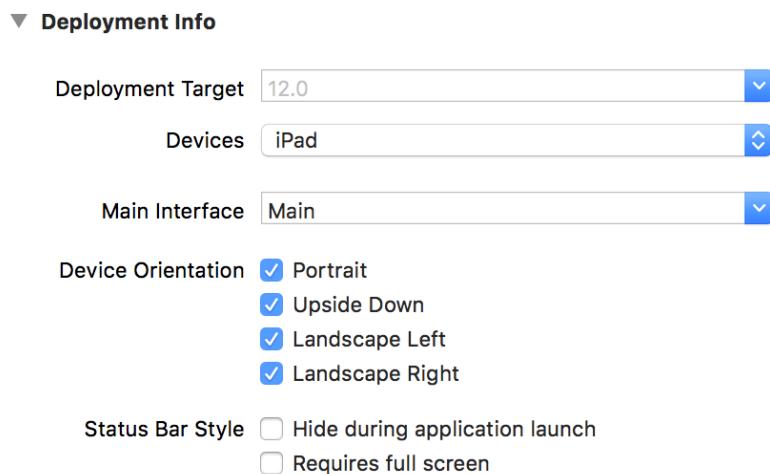


When designing your layouts try to avoid device or orientation specific layouts. Instead of designing for iPad/iPhone or portrait/landscape design for regular/compact heights and widths.

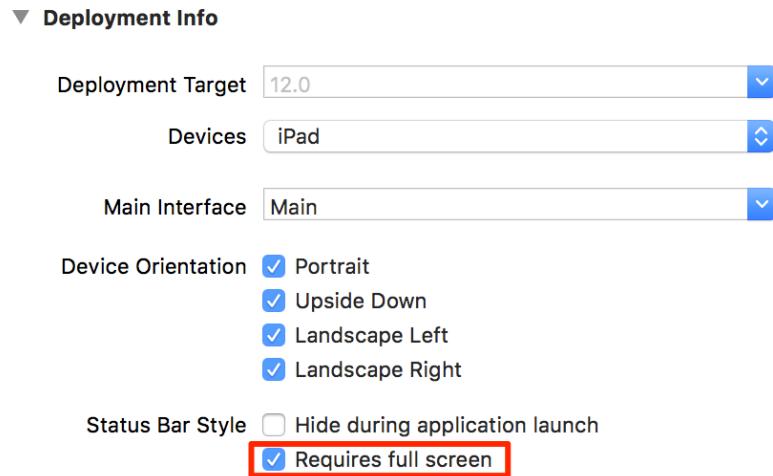
Supporting iPad Multitasking

Projects that you create with Xcode 10 support the iPad slide over and split view multitasking modes by default. Assuming you're building against the latest SDK, there are two prerequisites for iPad multitasking support:

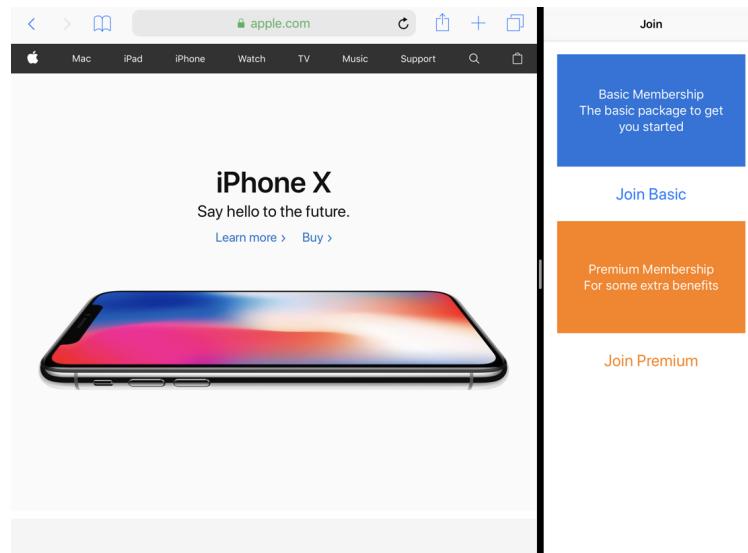
- Use a launch screen storyboard rather than launch images. (This doesn't mean you have to use a storyboard for your main user interface).
- Support all four device orientations on the iPad:



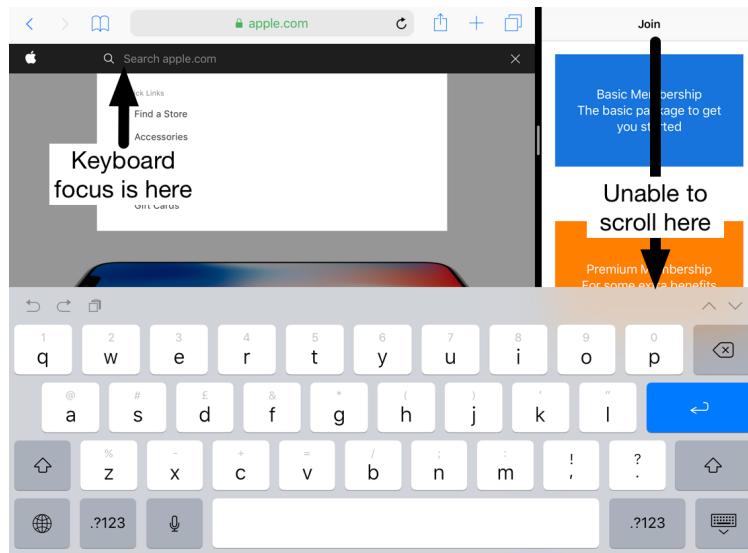
If your App doesn't support all four orientations or you want to disable it from working in slide over or split view modes set the option "Requires Full Screen":



The multitasking split view breaks some assumptions you might have for an iPad App. You can no longer assume your App has the full screen. It may only have a compact width typical of an iPhone App. Here's my App on the right-hand side of a landscape iPad sharing the screen with Safari:



You also need to remember the keyboard. My App is showing a sign-up screen using a mix of labels and buttons. It doesn't expect to interact with the keyboard. The problem is the keyboard can still appear if the other App needs it:

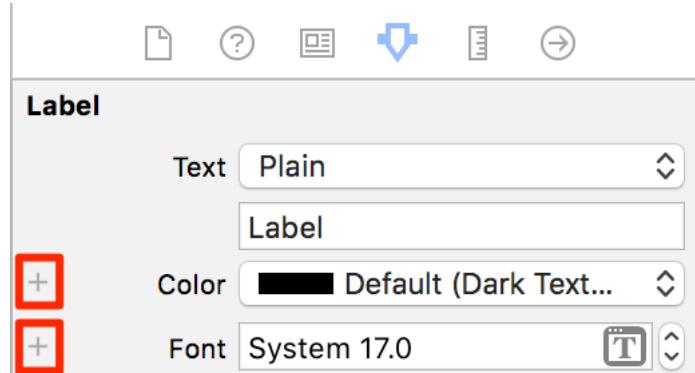


Unless you're using a table view or collection view which are aware of the keyboard you need to embed the layout in a scroll view that adjusts its content insets when the keyboard is visible. For a recap of the approach see the chapter on scroll views where we looked at [Managing The Keyboard](#).

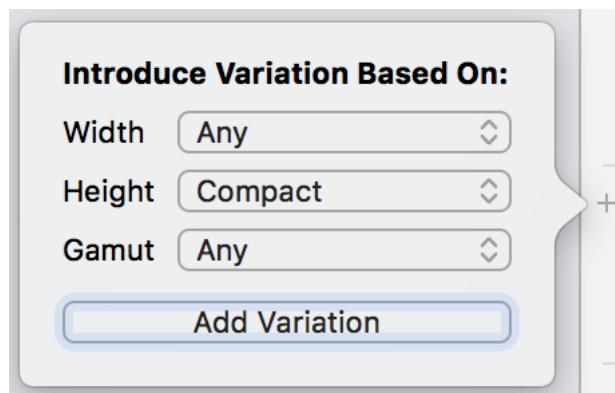
Using Size Classes With Interface Builder

Interface Builder has several ways to build layouts that adapt to the traits of the user interface:

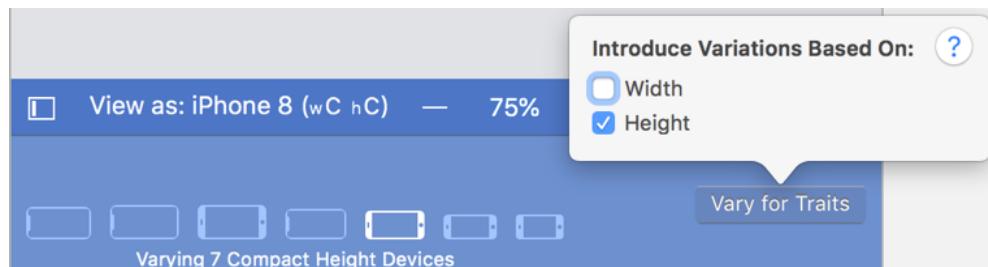
- Use the attributes and size inspectors to create a trait variation for a property of a view or constraint. The properties you can adapt this way have a **[+]** symbol on the left in the inspector:



You can vary the property for a combination of width, height or color gamut:



- Create a layout variation for the whole user interface using the preview controls. This allows you to change the installed views and constraints for specific size classes.

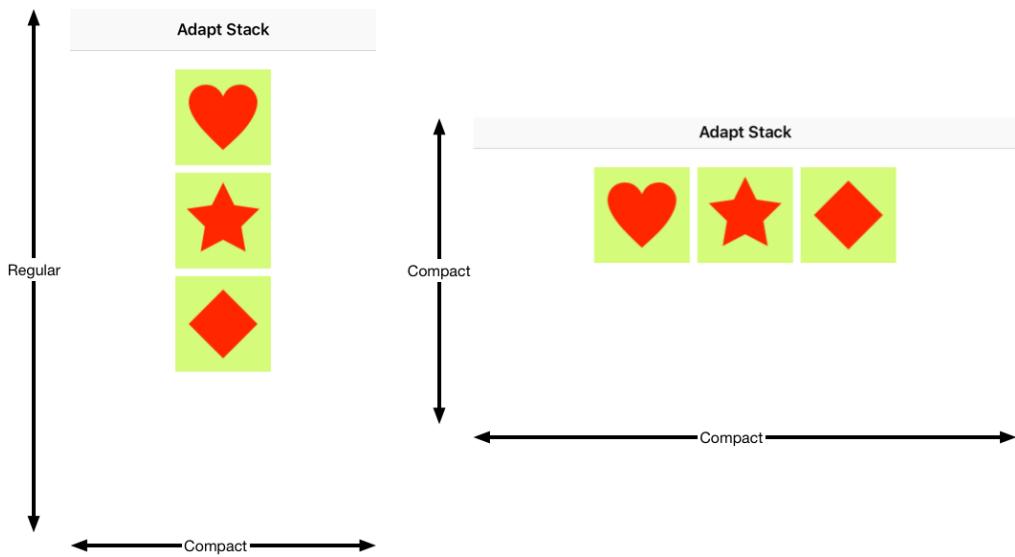


Properties that you can vary based on traits with Interface Builder include:

- Whether to install a view or constraint.
- Whether to show or hide a view.
- The font, color, tint or background.
- Layout margins.
- The image file used by an image view.
- Stack view configuration (axis, alignment, distribution, spacing, baseline).

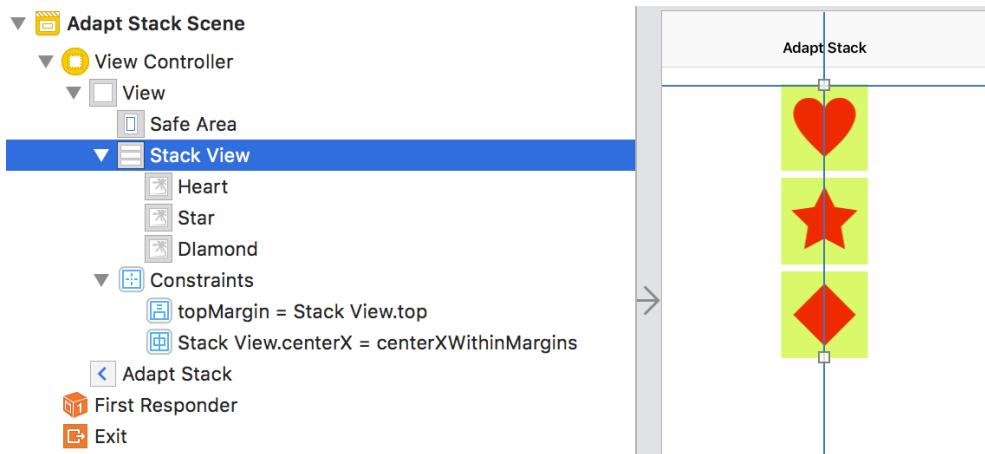
Adapting Properties For Size Classes

Let's see an example of trait-based property variations in Interface Builder by making a stack view adapt to changes in the size class. Here's a layout from when we were looking at [Dynamically Updating Stack Views](#) running on an iPhone 8:



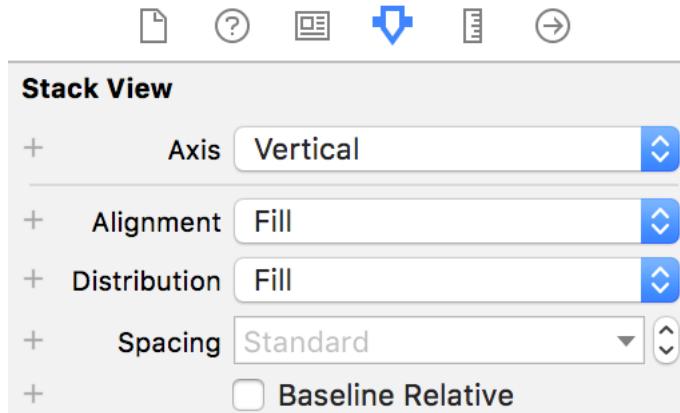
Rotating the iPhone changes from a regular height to a compact height. Let's use that to switch the stack view axis from vertical to horizontal (see sample code: [AdaptStack-v1](#)):

1. Here's my setup in Interface Builder with three images arranged in a vertical stack view:

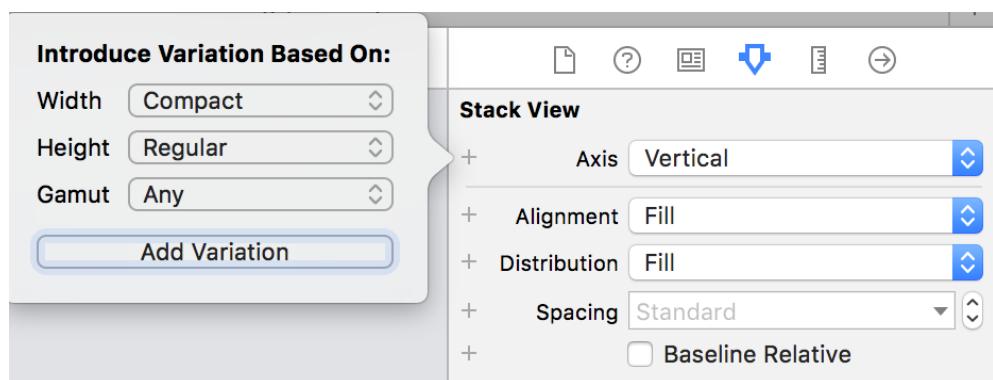


The vertical layout works with a regular height, but I would prefer a horizontal layout for an iPhone in landscape when we have a compact height. Interface Builder allows you to adapt the axis, alignment, distribution, spacing and baseline relative properties of a stack view. Let's make the axis change for a compact height size class.

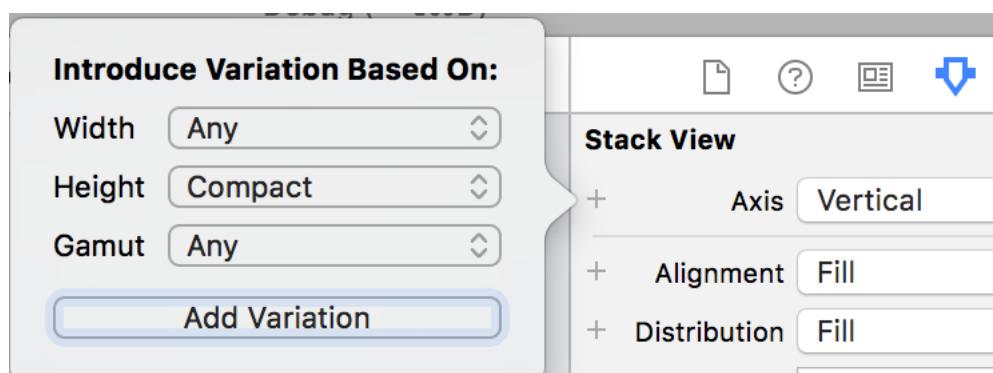
2. Select the stack view in the document outline in Interface Builder and take a look at the properties in the attributes inspector:



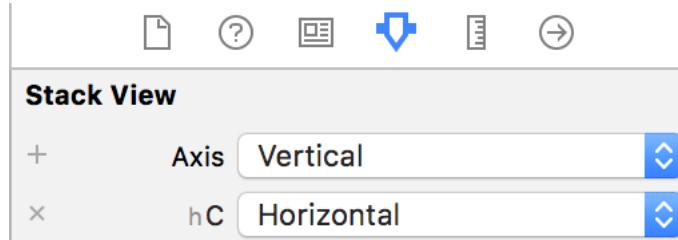
Notice how each of the properties has a **[+]** symbol on the left. Clicking one of these gives allows you to vary the property for a combination of horizontal (width) and vertical (height) size classes:



3. My stack view has a vertical axis by default. I want to change this whenever we have a compact height regardless of the width. So I need to select "Any" for the width and "Compact" for the height and then click the **[Add Variation]** button:

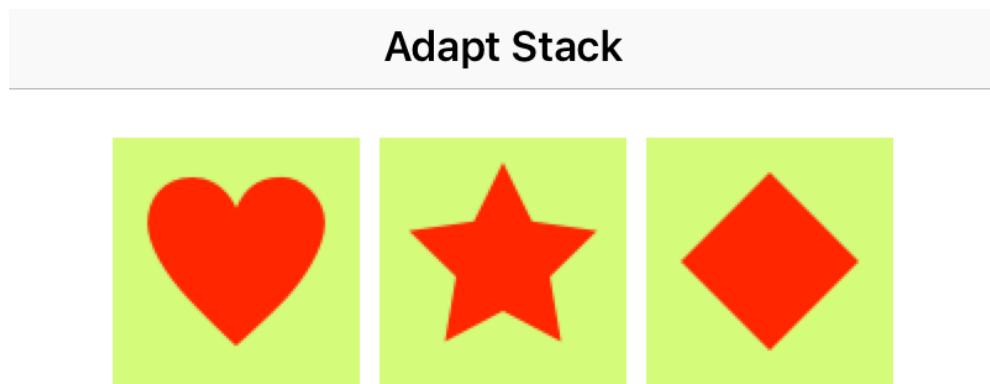


4. Our stack view now has two values for the axis. The first is the default and the second preceded by "hC" for height (h) compact (C). Change the value of the second to Horizontal:



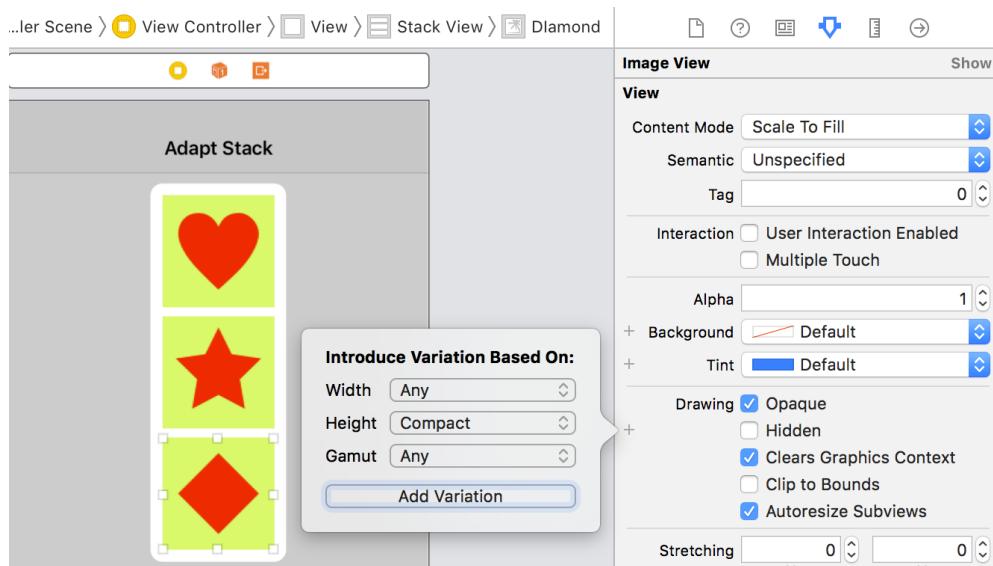
Note that you can remove a customization by clicking the small **[x]** on the left of the variation.

5. Our stack view should now use a vertical layout when running on an iPhone in portrait (regular height) but switch to a horizontal layout when we rotate the iPhone to landscape (compact height):



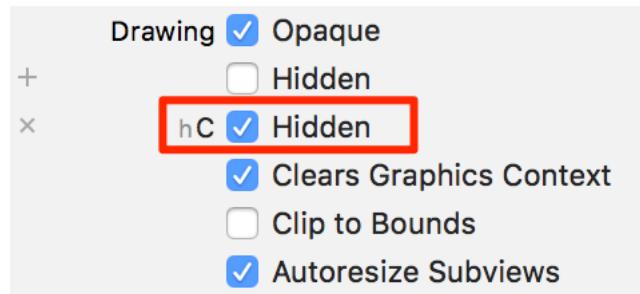
What if instead of rotating our vertical layout we hide one of the images when we have a compact height? We can do that with Interface Builder, but it's not so obvious how (see sample code: [AdaptStack-v2](#)):

1. The view setup is as before with three image views in a vertical stack view. This time instead of changing attributes for the stack view we need to change the `isHidden` property for the image view we want to hide. Select the bottom image in the stack view and use the attributes inspector to find the `Hidden` property for the view:



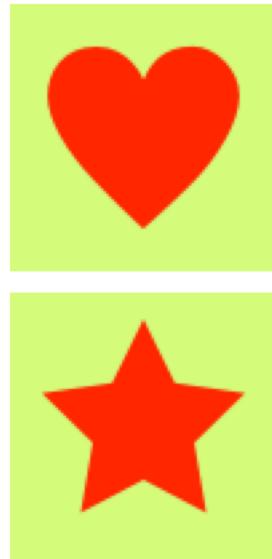
It has a **[+]** symbol on the left so we can add a variation as before for “Any” width and a “Compact” height.

2. Select the **[Hidden]** property for the compact height (hC) variation:



3. Rotating the iPhone to landscape (compact height) now hides the bottom image in the stack view:

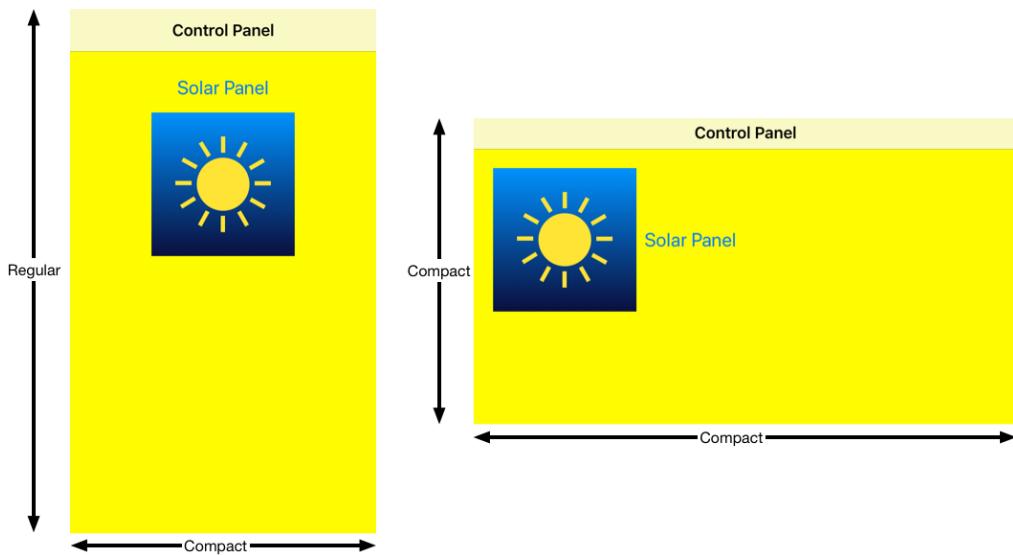
Adapt Stack



Stack views work well for adaptive layouts where you need to switch between horizontal and vertical layouts or show and hide views based on the size class. Compared to managing the constraints yourself it's much less work to use a stack view.

Adapting Layouts For Size Classes

Some layouts don't work well with stack views. In those cases, you may need to adapt the layout yourself by installing or removing constraints based on the size classes. Consider this layout with a button and image view shown on an iPhone SE:



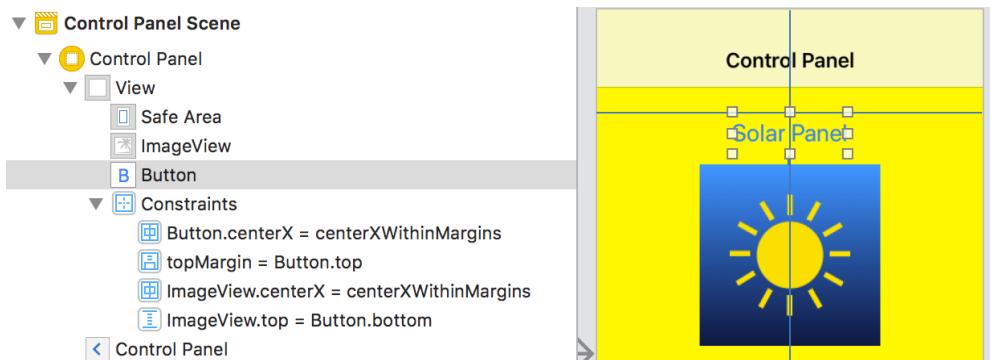
Some points to note:

- In portrait (regular height, compact width) I'm using a vertical layout centered in the superview with the button at the top.
- In landscape (compact height, compact width) I'm using a horizontal layout with the image pinned to the top-left margins and the button on the right.

This looks like it should work with a stack view but we would need to reverse the order of the arranged subviews when changing the axis. Instead, let's see how to build it with constraints.

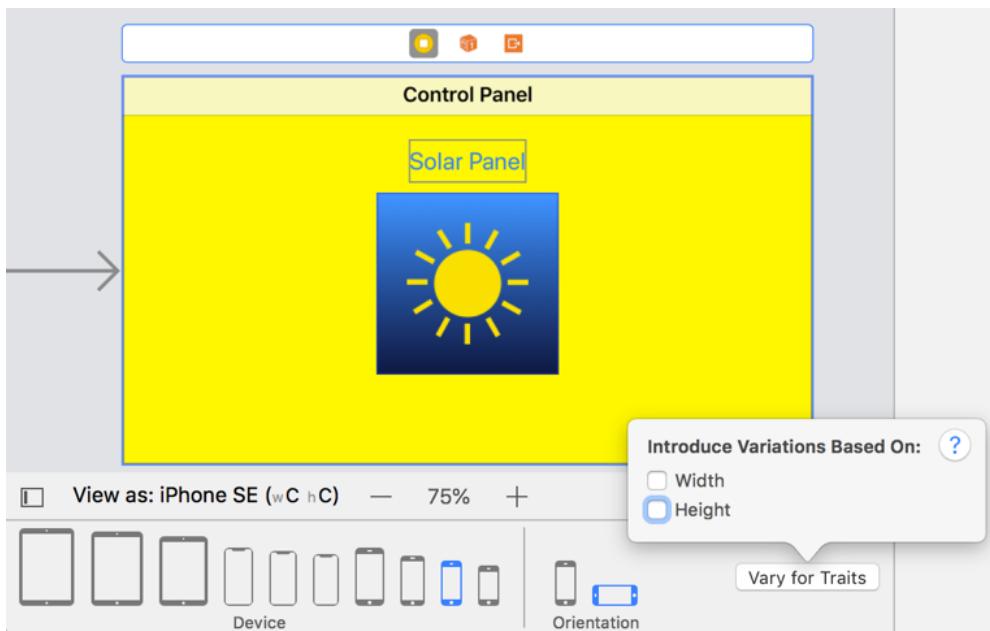
The variable is the height (vertical) size class. We need to install a set of constraints for the regular height case and another set for the compact case. We could do this by inspecting each constraint and creating a variation of its installed property. A more convenient way is to use the Interface Builder preview controls to create a layout variation (see sample code: [AdaptLayout-v1](#)):

1. Here's my layout in Interface Builder previewed on an iPhone SE in portrait (compact width, regular height):



It uses four constraints to center and vertically space the two views below the top margin.

2. To create the compact height variation use the preview controls to first switch to the landscape orientation:



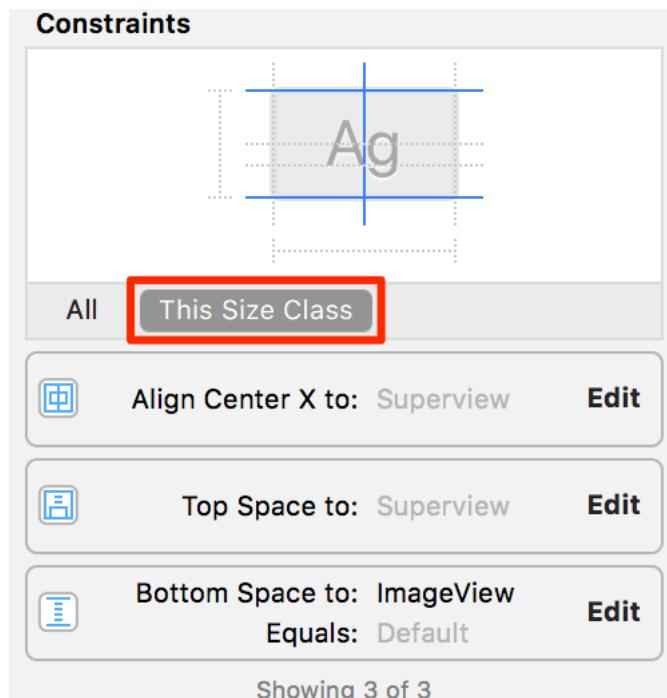
The “Vary for Traits” button gives us a choice to create a layout variation based on the width, height or both.

3. We want to create a variation for when the height changes from regular to compact. Selecting “Height” changes the preview control to blue and shows the devices where this layout variation applies:



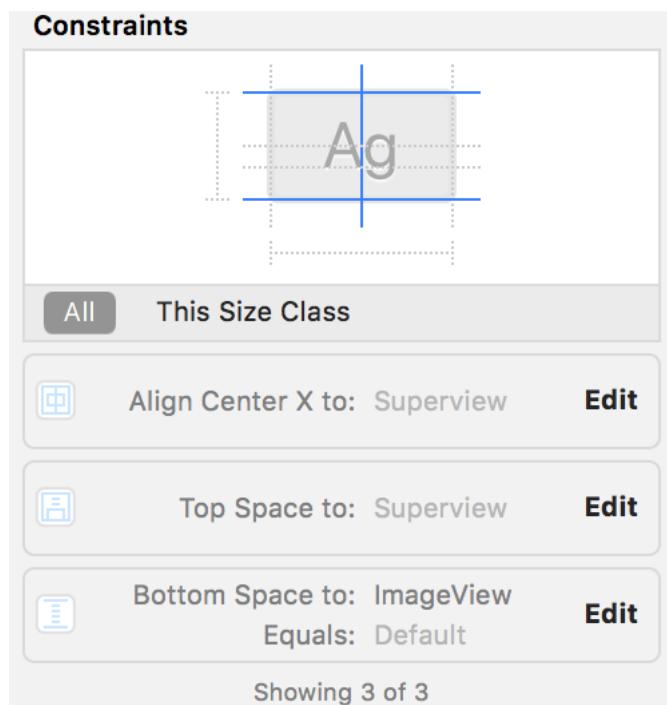
There are seven devices with a compact height (all iPhones in landscape).

4. I find it easiest to work view by view first removing the constraints we don't need in this variation. Click on the button and use the Size Inspector to views its constraints:

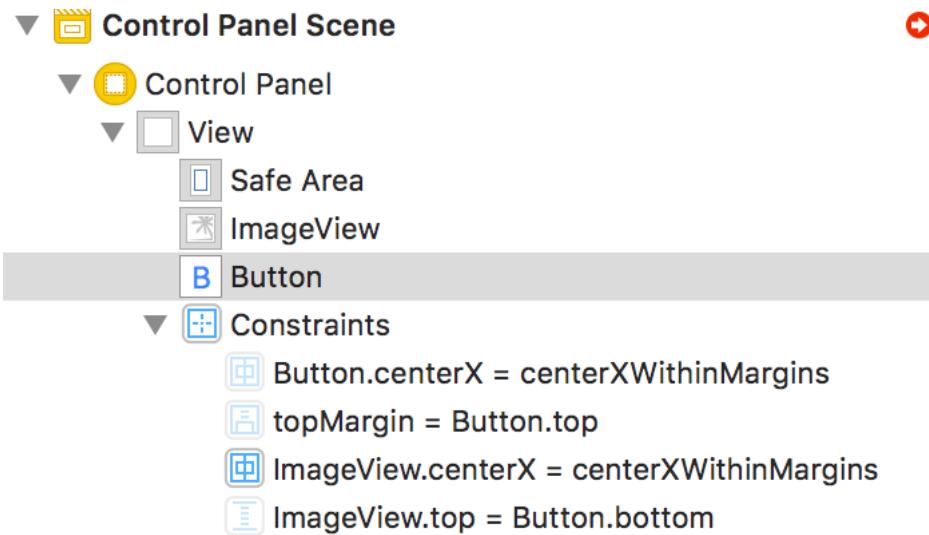


Use the filter to show only those constraints that apply to this size class.

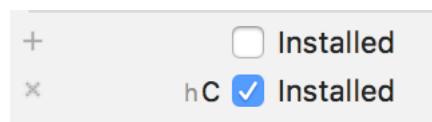
5. Click on each of the constraints and use the backspace key to remove them. If you switch the filter back to "All" you should see that Interface Builder has grayed out the constraints as they are not installed for this variation:



The constraints are also grayed out in the document outline:



If you click on one of the constraints and examine it in the size inspector, you should see that Interface Builder has created a variation on the "Installed" property of the constraint. The constraint is not installed when the layout has a compact height:



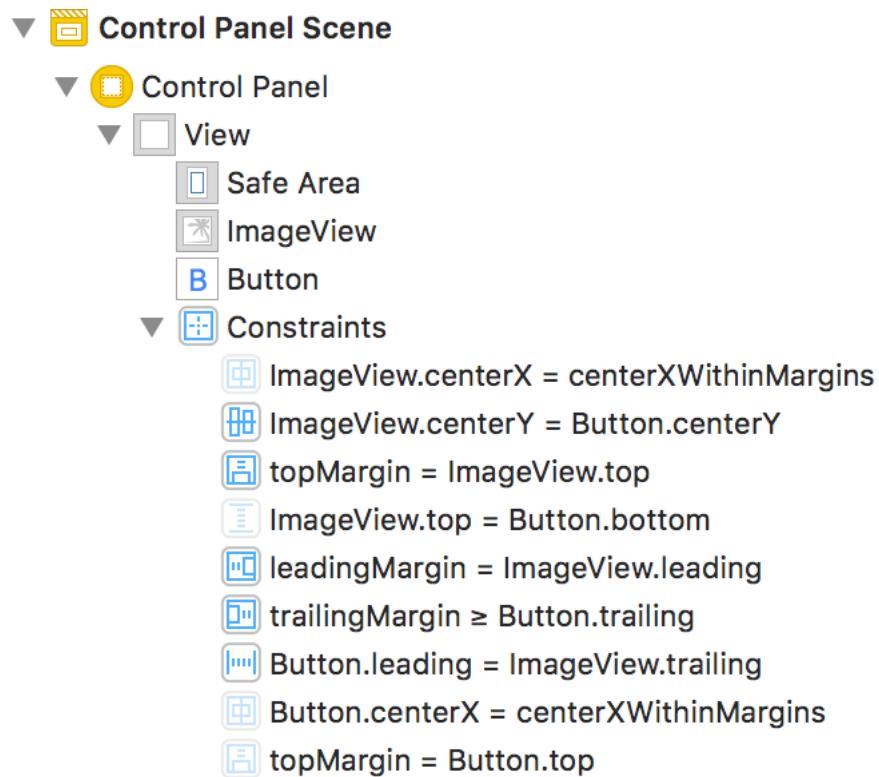


Don't delete the constraints in the document outline. That deletes the constraints for all variations. Always delete the constraints in the canvas or using the Size Inspector.

6. Select the image view and repeat the process to delete the remaining center constraint for the image view. Reposition the views in the canvas to roughly where we want them in this variation:

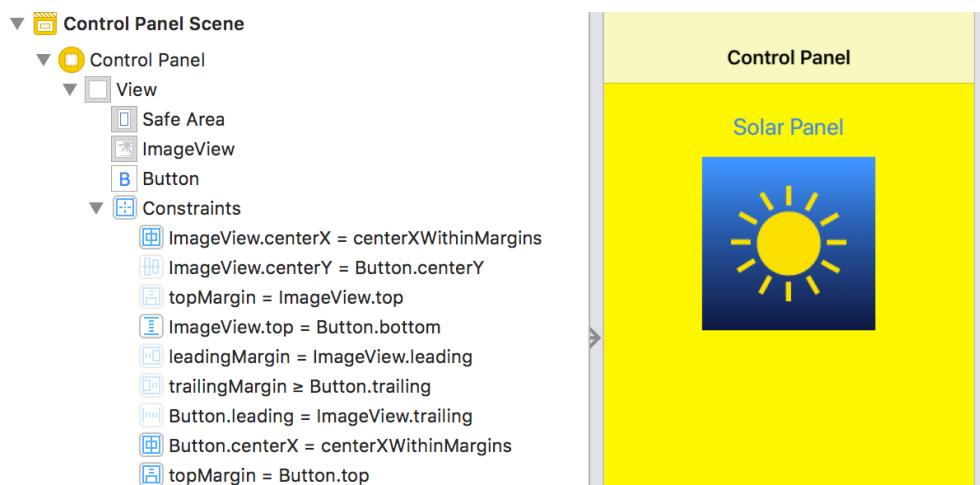


7. Add the constraints for this layout. First, add constraints to fix the position of the image view to the top and leading margins and vertically center it with the button.
8. Add a constraint to use a standard amount of horizontal spacing between the image view and the button. Add a trailing constraint from the button to the trailing margin. I don't want the button to fill the width, so I made this a greater than or equal constraint. For safety, I also increased the horizontal compression-resistance priority of the image view to 751. If our button ever gets too wide it will be squeezed before the image view.
9. You should have five installed constraints in the document outline (the order is not important):



Select the “Done Varying” button in the preview control to complete the layout variation.

10. Switch between the portrait and landscape orientation in the preview control. The layout in the canvas and the set of installed constraints in the document outline should switch between the two variations. Here’s how it looks in portrait with the compact height constraints grayed out in the outline:



Creating a layout variation using the Interface Builder preview control works well when you have a small number of views with a limited number of constraints. It can become hard to follow when you have a large number of constraints to vary.

Using Traits In Code

You don't have to use Interface Builder to build adaptive layouts. The `UITraitEnvironment` and `UIContentContainer` protocols give us two ways to react to trait changes in code.

UITraitEnvironment Protocol

The `UITraitEnvironment` protocol is adopted by `UIScreen`, `UIWindow`, `UIViewController`, `UIView` and `UIPresentationController`. It defines a property you can use to get the trait collection of the user interface and a method you can override to be informed when the trait collection changes:

```
// UITraitEnvironment
var traitCollection: UITraitCollection { get }
func traitCollectionDidChange(_ previousTraitCollection:
    UITraitCollection?)
```

The `previousTraitCollection` parameter is an optional as it can be `nil` when this method is called for the first time for a view that has just been added to the view hierarchy. If you override the method, you must first call `super`.

Let's recreate the adapting stack view example in code using the `UITraitEnvironment` protocol in our view controller (see sample code: [AdaptStack-v3](#)):

1. I'll start by creating the three image views in the view controller (assuming the image files are in the asset catalog):

```
private let heart = UIImageView(image: UIImage(named:
    "Heart"))
private let star = UIImageView(image: UIImage(named:
    "Star"))
private let diamond = UIImageView(image: UIImage(named:
    "Diamond"))
```

2. Then we can build the stack view for the image views:

```
private lazy var stackView: UIStackView = {
    let stackView = UIStackView(arrangedSubviews: [heart,
        star, diamond])
    stackView.translatesAutoresizingMaskIntoConstraints = false
    stackView.spacing = UIStackView.spacingUseSystem
    return stackView
}()
```

3. Then I add the constraints to center the stack view below the top margin (which I increase to 20 points):

```
override func viewDidLoad() {
    super.viewDidLoad()
    setupView()
}

private func setupView() {
    view.addSubview(stackView)
    view.directionalLayoutMargins =
    NSDirectionalEdgeInsets(top: 20.0, leading: 20.0,
    bottom: 20.0, trailing: 20.0)
    let margins = view.layoutMarginsGuide
    NSLayoutConstraint.activate([
        stackView.topAnchor.constraint(equalTo:
    margins.topAnchor),
        stackView.centerXAnchor.constraint(equalTo:
    margins.centerXAnchor)
    ])
}
```

4. Now we need to set the axis of the stack view based on the vertical size class. Here's a helper method to do that:

```
private func configureView(for traitCollection:
UITraitCollection) {
switch traitCollection.verticalSizeClass {
    case .compact:
        stackView.axis = .horizontal
    default:
        stackView.axis = .vertical
    }
}
```

The stack view axis is horizontal for a compact height (vertical) size class otherwise it's vertical.

5. Finally we configure the stack view anytime the vertical size class changes by implementing `traitCollectionDidChange`:

```
override func traitCollectionDidChange(_  
    previousTraitCollection: UITraitCollection?) {  
    super.traitCollectionDidChange(previousTraitCollection)  
    if previousTraitCollection?.verticalSizeClass !=  
        traitCollection.verticalSizeClass {  
        configureView(for: traitCollection)  
    }  
}
```

Some notes to remember when using `traitCollectionDidChange`:

- You **must** always call `super.traitCollectionDidChange(_ :)` first.
- This method is called with the previous value of the trait collection (can be `nil`). The current trait collection of the view controller is available in the `traitCollection` property.
- We are not interested in every trait change that might trigger this method. To avoid unnecessary work, we check if the vertical size trait changed before configuring the view.

UIContentContainer Protocol

The `UIContentContainer` protocol provides a second way to react not only to trait changes but also to size changes. It's adopted by view controllers but not by views and provides two methods for responding to trait changes:

```
// UIContentContainer  
viewWillTransition(to size: CGSize, with coordinator,  
    UIViewControllerTransitionCoordinator)  
  
willTransition(to newCollection: UITraitCollection, with  
    coordinator: UIViewControllerTransitionCoordinator)
```

The first of these methods is called before changing the size of a view controller's view. Its first parameter is the new size of the view. The view has not yet changed size so you can still get its old size if needed. We'll see an example that uses this method later.

The second method is called before changing the trait collection. Its first parameter is the new trait collection. The old trait collection is available in the `traitCollection` property of the view controller. If you override either method, you should call `super` in your implementation.

How do these methods compare to the `UITraitEnvironment` method `traitCollectionDidChange(_ :)`?

- Neither of the `UIContentContainer` methods are called when a view controller first adds its view to the view hierarchy. You cannot use them to do initial view and constraint setup.
- The `traitCollectionDidChange` method is called after a view has been added to the view hierarchy and had its trait collection set so we can use it to do trait based view setup (we saw this when adapting the stack view in the last example).
- The `UIContentContainer` methods are only available in the view controller whereas the `traitCollectionDidChange(_ :)` method can be used both in the view controller and in any custom subclass of `UIView`.
- You can use the `coordinator` object of the `UIContentContainer` methods when you want to run your animations alongside the view controller transition animation. We'll see an example of this shortly.

When a view controller is about to transition to a new size or trait collection the sequence of events is as follows:

1. During the setup of the transition the methods are called in the following order:

```
// UIContentContainer - trait will change
willTransition(to newCollection: UITraitCollection, with
    coordinator: UIViewControllerTransitionCoordinator)

// UIContentContainer - size will change
viewWillTransition(to size: CGSize, with coordinator,
    UIViewControllerTransitionCoordinator)

// UITraitEnvironment - trait changed
traitCollectionDidChange(_ previousTraitCollection:
    UITraitCollection?)
```

2. During the animation of the transition, you use the coordinator object to run your animations alongside the system animations.
3. Once the animation completes, use the completion handler of your animations to perform any cleanup.

Let's see how this works in practice by adding an animation to our adapting stack view (see sample code: [AdaptStack-v4](#)):

1. To add an animation to our adapting stack view we need to implement

one extra method of the `UIContentContainer` protocol in our view controller:

```
// ViewController.swift
override func willTransition(to newCollection:
    UITraitCollection, with coordinator:
    UIViewControllerTransitionCoordinator) {
    super.willTransition(to: newCollection, with:
        coordinator)
    // Handle change
}
```

Note that we remember first to call `super`.

2. We only want to animate changes to the vertical size class so we first check that it has changed:

```
if traitCollection.verticalSizeClass != newCollection.verticalSizeClass {
    animateStack(with: coordinator)
}
```

3. Our animation method takes the transition coordinator as a parameter:

```
private func animateStack(with coordinator:
    UIViewControllerTransitionCoordinator) {
    // animate stack view
}
```

4. The animation applies a scale transform to the stack view that makes it grow during the view controller transition.

```
coordinator.animate(alongsideTransition: { context in
    self.stackView.transform = CGAffineTransform(scaleX:
        AnimationMetrics.transformScale, y:
        AnimationMetrics.transformScale)
}, completion: { context in
    // completion handler
})
```

5. In the completion handler when the transition is complete I use a property animator to reset the transform so that the stack view shrinks back down to its normal size:

```
// completion handler
UIViewPropertyAnimator.runningPropertyAnimator(withDuration:
    AnimationMetrics.duration, delay: 0, options: [],
    animations: {
    self.stackView.transform = .identity
})
```

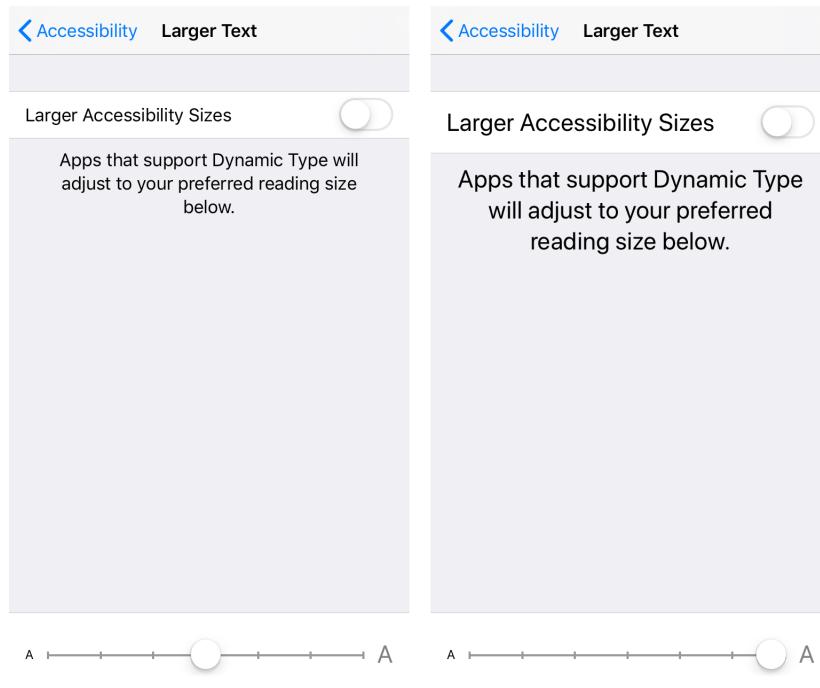
6. I defined the animation metrics I use above with a private enum in the view controller:

```
private enum AnimationMetrics {
    static let duration: TimeInterval = 0.3
    static let transformScale: CGFloat = 1.25
}
```

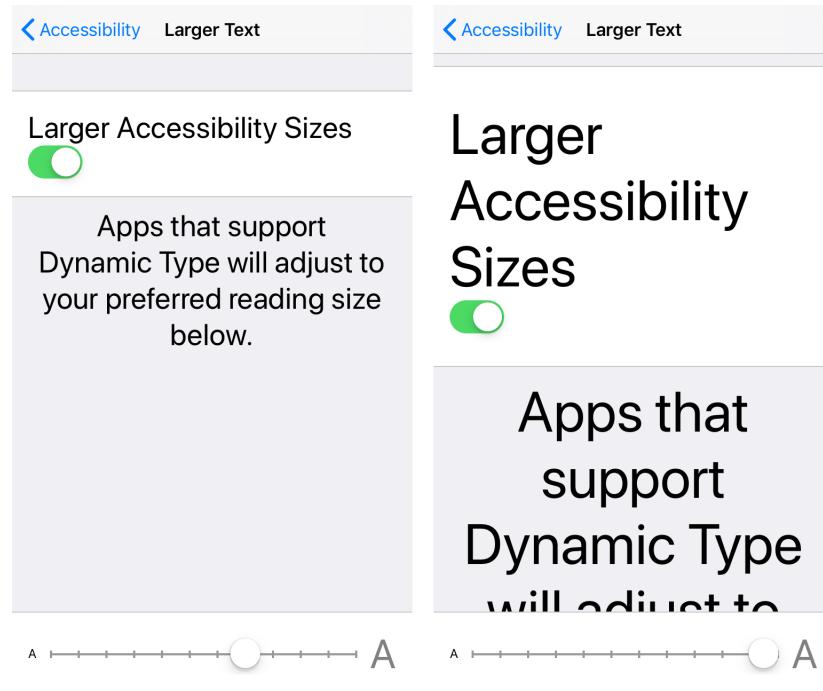
7. You have to run the sample code to see the effect, but the stack view should appear to grow during the transition as it rotates and shrink back after the transition ends.

Adapting For Dynamic Type Size

We can do more than adapt to the size class or size of the view. Let's look at an example where we adapt a layout as the dynamic type content size increases. Here's the Apple Settings screen to enable the larger dynamic type accessibility sizes, shown at the default and largest standard text sizes:

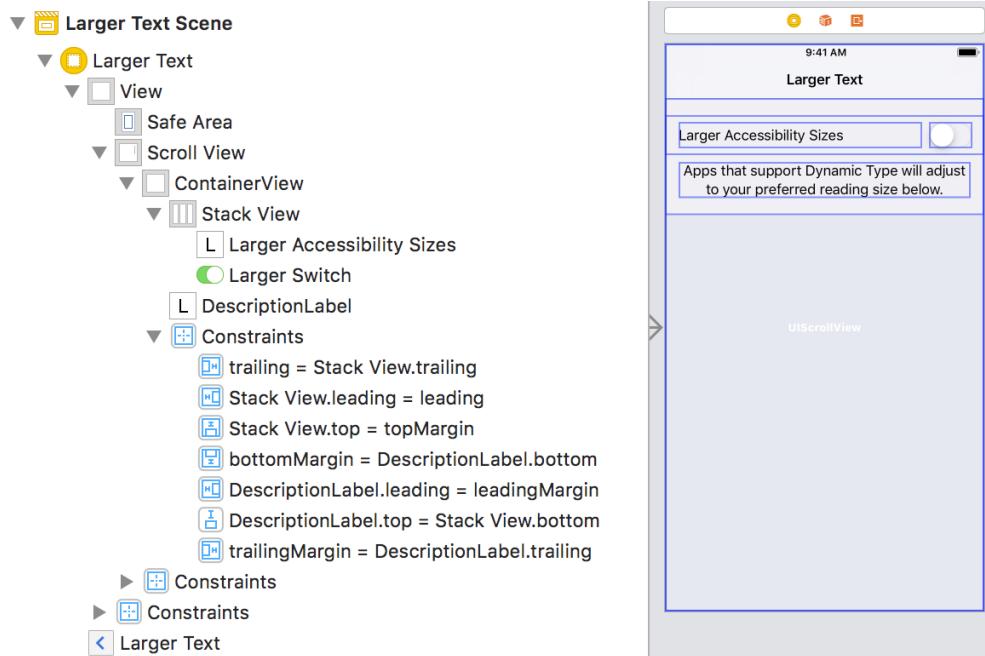


The slider at the bottom changes the preferred content size. The switch at the top allows access to the larger accessibility text sizes. Notice how the label and the switch are sharing the horizontal width. Look what happens when we enable and then use one of the larger accessibility sizes:



The label now gets to use the full width, and the switch drops down below the label. Let's see if we can recreate that behavior (see sample code: [AdaptType-v1](#)).

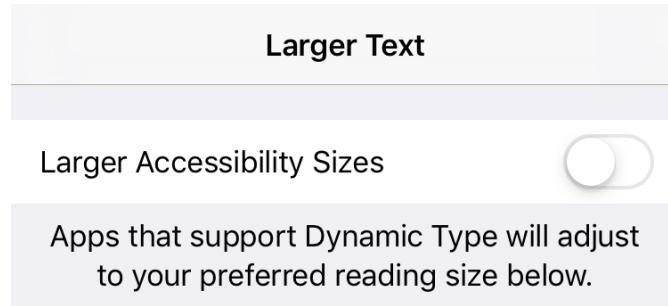
We're going to ignore the slider and only build the labels and switch. Apple uses a table view, but I went with a horizontal stack view in a container view embedded in a scroll view. Here's the view layout I came up with:



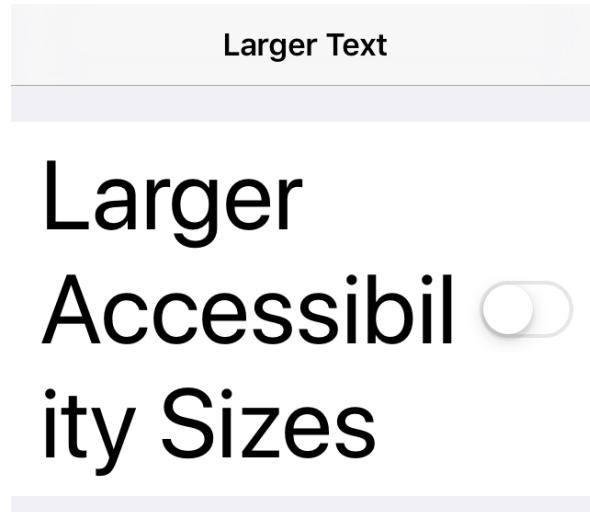
Some quick notes on this layout:

- I pinned the container view to the edges of the scroll view and gave it 20pt top/bottom margins and 8pt leading/trailing margins.
- I constrained the stack view to the top/bottom margins of the container view and to the leading and trailing edges of the container view. This allows me to add a white background view to the stack view that fills the full width of the root view.
- I gave the stack view an additional 8pt margin to inset the label and switch.
- The scroll view, container view and stack view all “Preserve Super-view Margins”.
- I constrained the description label to the leading and trailing margins of the container view and vertical spaced it below the stack view.
- The two labels are using the Body text style.

Here's how it looks on an iPhone 8 at the default text size:



At the larger accessibility sizes keeping the label and switch arranged horizontally becomes cramped especially on small devices like the iPhone SE:



To give the text as much space as possible we can move the switch down below the label when the user selects one of the larger accessibility sizes. How do we do that?

We could observe the content size category did change notification, but I think there's a better way using trait collections. Since iOS 10 the `UITraitCollection` class has a property for the preferred content size category. We can check for changes to this property in our view controller:

```
override func traitCollectionDidChange(_  
    previousTraitCollection: UITraitCollection?) {  
    super.traitCollectionDidChange(previousTraitCollection)  
    if previousTraitCollection?.preferredContentSizeCategory !=  
        traitCollection.preferredContentSizeCategory {  
        configureView(for: traitCollection)  
    }  
}
```

If the preferred content size category has changed, we reconfigure the view based on the current trait collection:

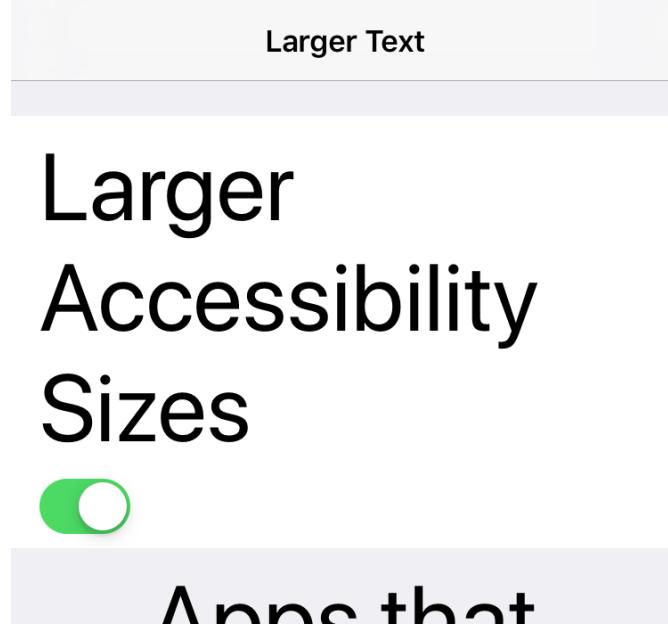
```
private func configureView(for traitCollection:  
    UITraitCollection) {  
    let contentSize =  
        traitCollection.preferredContentSizeCategory  
    if contentSize.isAccessibilityCategory {  
        stackView.axis = .vertical
```

```
    stackView.alignment = .leading
} else {
    stackView.axis = .horizontal
    stackView.alignment = .center
}
}
```

To check if the user has switched to a large accessibility text size, we get the current content size from the trait collection. The content size category structure has a convenient instance property to let us test for accessibility sizes:

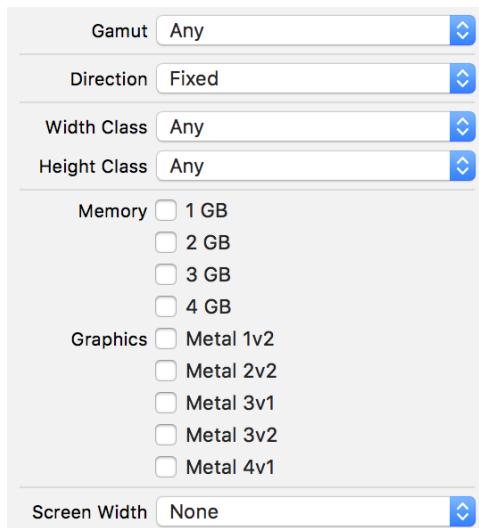
```
// UIContentSizeCategory
var isAccessibilityCategory: Bool { get }
```

If the user has chosen an accessibility size, we reconfigure the stack view to a vertical axis and change the alignment to leading. Otherwise, we use the horizontal axis with center alignment. Here's how it looks on an iPhone 8 at the largest accessibility size:



Using Traits With The Asset Catalog

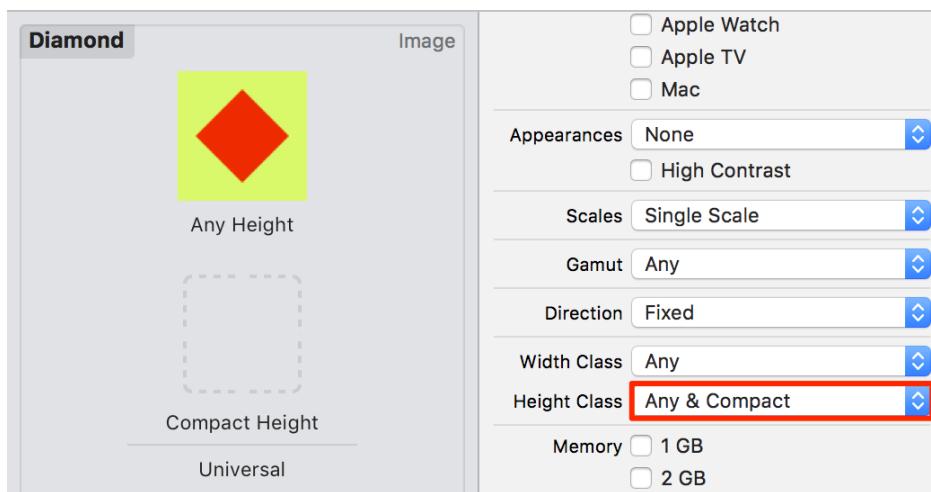
The Asset Catalog allows you to create asset variations for a variety of different device traits. Select the image asset in the catalog and then use the Attributes Inspector to configure the variations you want to add:



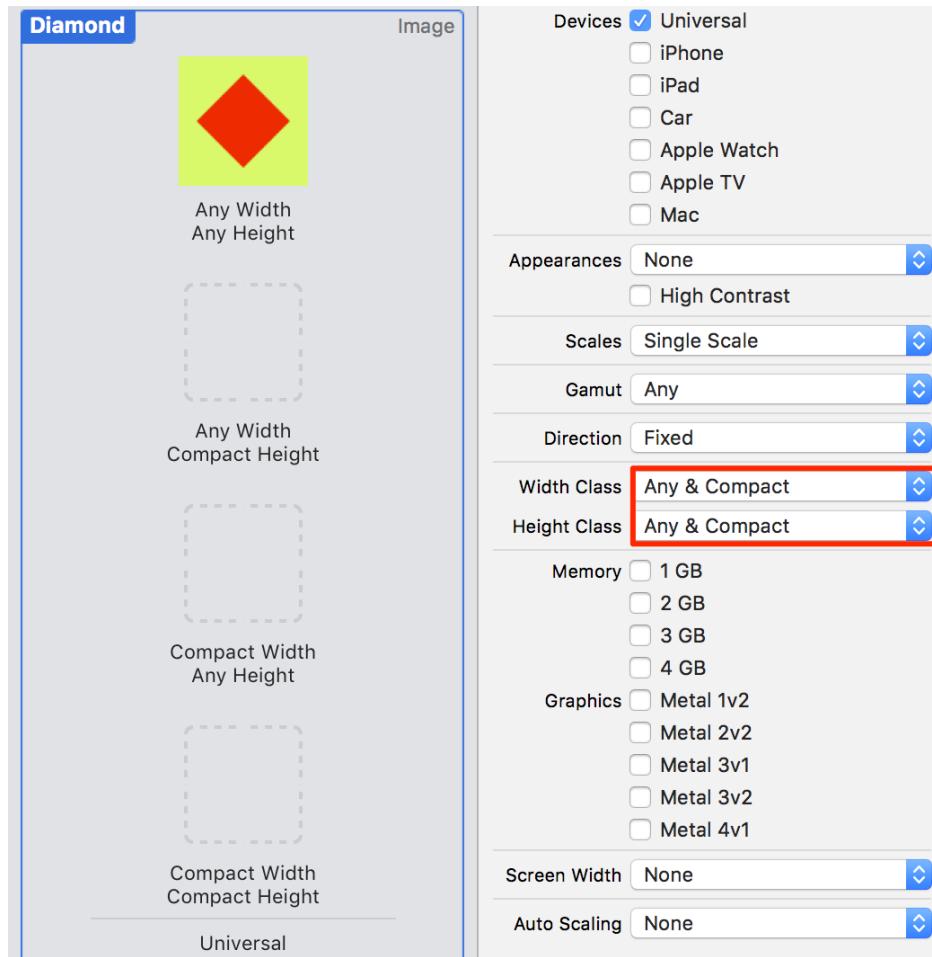
You can create asset variations for the following traits:

- **Color Gamut:** Use different images for sRGB and P3 color spaces.
- **Text Direction:** Image variations for left-to-right and right-to-left text directions.
- **Width and Height Class:** Image variations for different combinations of horizontal and vertical size class.
- **Memory:** Variations based on the memory capacity of the device.
- **Graphics:** Variations based on the graphics capability of the device.
- **Screen Width:** Apple Watch screen width variations (38mm and 42mm).

As you create new variations, additional placeholders appear in the asset catalog. Drag the asset you want to apply for a variation onto the placeholder. For example, selecting “Any & Compact” for the height class adds a “Compact Height” variation to the catalog that you can use to add an image that better fits a compact height user interface:



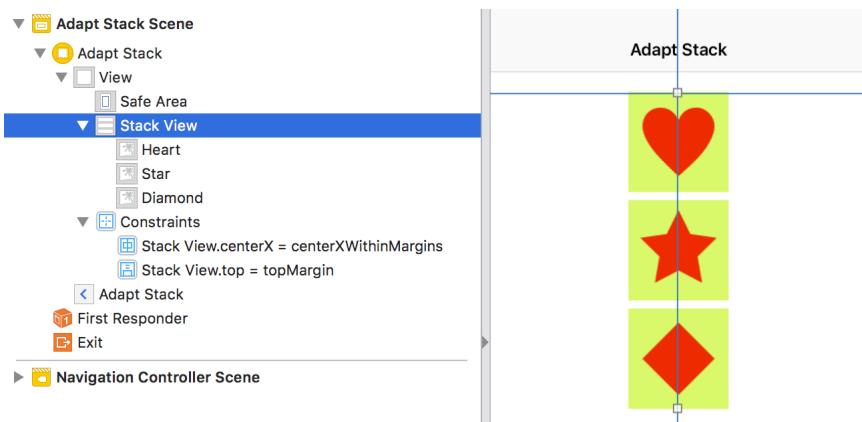
You don't need any code changes to use the image variations. The image view uses the "Compact Height" image when the vertical size class is compact. Otherwise, it uses the "Any Height" image. Things get a little more complex if we also add a width variation. Selecting "Any & Compact" for the width class gives us four variations for the image.



You don't need to add an image for each of the variations. The "Any Width, Any Height" image acts as a default if there's no better match.

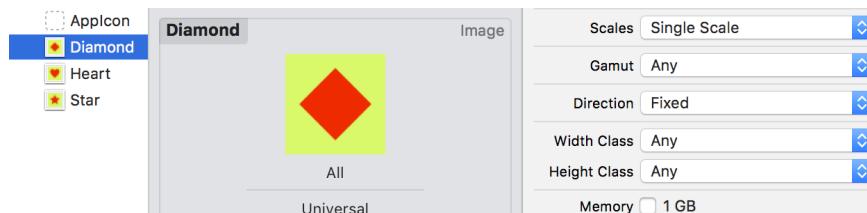
Let's revisit our stack view example and see how we can use asset catalog variations to make better use of the extra space of an iPad screen (see sample code: [AdaptAsset-v1](#)):

1. To recap our layout, we have three image views embedded in a vertical stack view that we rotate to the horizontal axis for compact heights:



I'm using PDF vector images which are all 100 x 100 points in size (@1x scale). They fit an iPhone screen but waste space when viewed full screen on an iPad.

2. I created three larger versions of the images that are 200 x 200 points in size that we can use whenever we have both regular height and regular width. Let's start with the diamond image in the asset catalog:

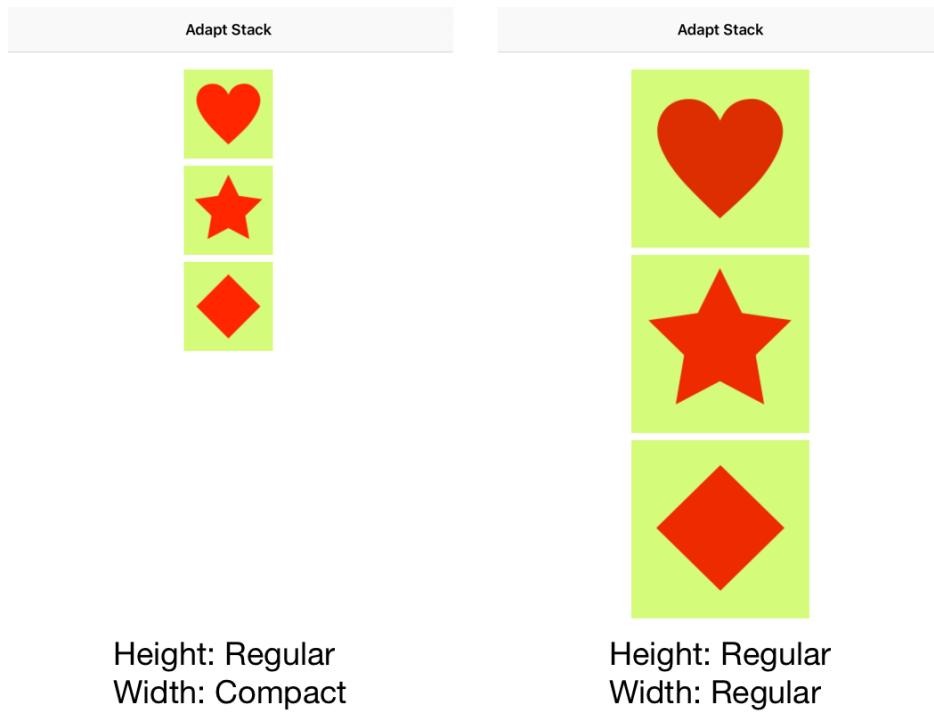


3. Use the Attributes Inspector to select the "Any & Regular" variations for both the width and height:



Add the 200x200 point image as the “Regular Width, Regular Height” variation. We don’t need images for the other two variations. If one of the dimensions is compact we fallback to the default Any-Any variation which uses the 100x100 point image.

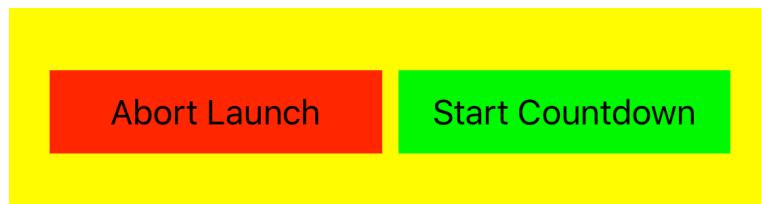
4. Repeat this process for the other two images in the asset catalog.
5. The layout uses the smaller images whenever we have a compact dimension and the larger images when we have both regular height and regular width:



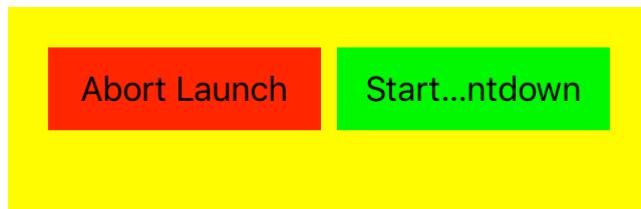
Variable Width Strings

You can use size classes to adapt the font size of the text in your user interface but how about changing the text based on the available space? This can sometimes help when you're using a fixed size font (I don't recommend doing this with dynamic type). When you have limited space, you can use a shorter version of the text rather than shrinking the font size.

For example, suppose I have two buttons arranged horizontally with titles that just about fit on an iPhone XS in portrait:



On a smaller device like an iPhone SE the button titles start to become truncated (a problem that's likely to get worse if we have to localize the text):



The localization system allows us to build a dictionary of variable width strings for this purpose. Even better both `UILabel` and `UIButton` are aware of variable width strings and automatically select the best string variation from the dictionary for the available screen width (see sample code: [AdaptText-v1](#)):

1. My layout is two buttons in a horizontal stack view that I pinned to the top, leading and trailing margins. The stack view is using a fill equally distribution with standard spacing:

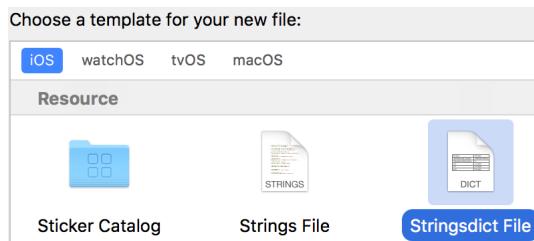


The button titles are using the standard system font at 17 points.

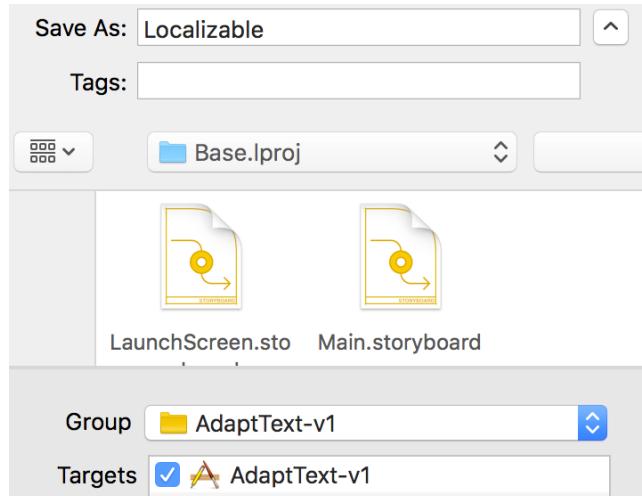
2. We have to adapt our strings in code, so we need outlets in the view controller that connect to the two buttons in Interface Builder:

```
@IBOutlet var stopButton: UIButton!
@IBOutlet var startButton: UIButton!
```

3. To adjust the text, we need a localizable strings dictionary. Add one to your Xcode project (File > New > File...). Find it in the template browser in the iOS Resource section named “Stringsdict File”:



4. Name the file `Localizable.stringsdict` and save it in the base localization directory (`Base.lproj`) along with the project storyboards:



5. The strings dictionary needs an item for each of the button titles that we want to localize. We use the key for the item when looking up the localization to set our button titles. Each string item is a dictionary containing another dictionary with width variations rules:

Key	Type	Value
▼ Strings Dictionary	Dictionary	(2 items)
▼ Abort	Dictionary	(1 item)
▼ NSStringVariableWidthRuleType	Dictionary	(3 items)
20	String	Abort
25	String	Abort Launch
50	String	Abort Launch Sequence
▼ Start	Dictionary	(1 item)
▼ NSStringVariableWidthRuleType	Dictionary	(3 items)
20	String	Start
25	String	Start Countdown
50	String	Start Launch Countdown

Each width variation entry has a width in *em* units and a string value (see the sidebar - [How Big Is An Em?](#)). When looking up the localized string, we get back the variation that best fits the available width.

You may need to experiment with the em values to use. I followed an Apple example and used 20, 25 and 50. The 20em variation only shows up on the iPhone SE. The 25em variation covers the other iPhones in portrait and 50em the larger iPhones in landscape and

all iPads.

6. To use our variable width strings, we need to manually set the button titles in our view controller when loading the view:

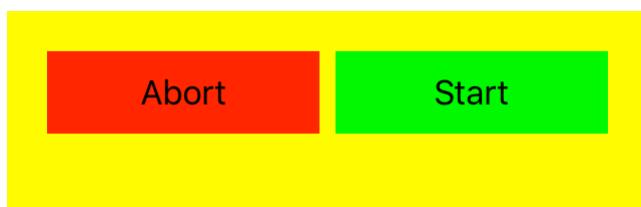
```
override func viewDidLoad() {
    super.viewDidLoad()
    setupView()
}

private func setupView() {
    let startTitle = NSLocalizedString("Start", comment:
        "Start button")
    startButton.setTitle(startTitle, for: .normal)

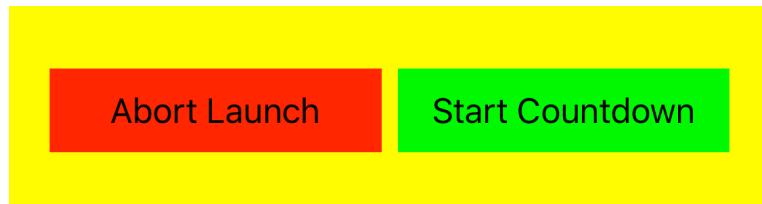
    let stopTitle = NSLocalizedString("Abort", comment:
        "Abort button")
    stopButton.setTitle(stopTitle, for: .normal)
}
```

We use `NSLocalizedString` to look up the localization based on the key name we used in the strings dictionary.

7. There's nothing more to do but build and run. The standard UIKit controls like `UILabel` and `UIButton` are smart enough to know about adaptive strings and apply the appropriate localization based on the screen size at runtime.
8. Here's how it looks when running on the iPhone SE which uses the 20em string variation:



9. Running on an iPhone XS in portrait we get the 25em variation:



10. Rotating the iPhone XS to landscape gives us the 50em variation:



I find that the button and label text don't always adapt perfectly to width changes at runtime. They sometimes use the shorter string when a longer one would fit.

How Big Is An Em?

An *em* is a typographical measure that defines a distance equal to the type size or a space equal to the square of the type size. So for a 17 point type, 1 em is a distance of 17 points or an area of 17 x 17 points. The name comes from the traditional use of the width of the capital M to approximate the type size.

Apple doesn't document how UIKit calculates the em value for variable width strings. In Session 227 from WWDC 2015 on internationalization, Apple says they calculate it from the number of "M" characters that fit in the visible screen width at the standard system font size (14 points).

My experimentation gives these em sizes for a selection of devices:

Device	Portrait (em)	Landscape (em)
iPhone SE	22	39
iPhone 8	25	46
iPhone 8 Plus	28	50
iPhone X/XS	25	56
iPhone XR/XS Max	28	61
iPad Pro 10.5"	57	76

iPad Pro 12.9"	70	94
----------------	----	----

When Size Classes Are Not Enough

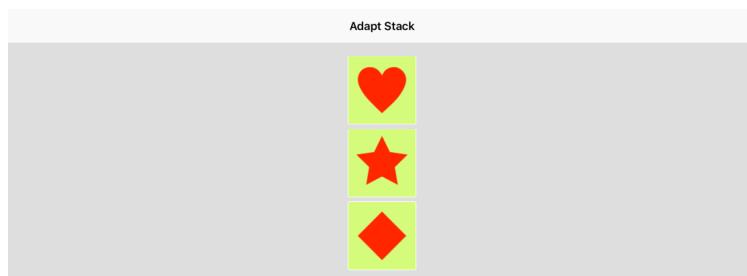
A frequent complaint about size classes is that they are too coarse of a control. This can be a challenge when creating layouts that work across all devices:

- A compact width size class covers a range from the smallest iPhone up to the iPad Pro 10.5" in 50:50 split view.
- A full-screen iPad App always has a regular width and height even when rotated.
- An iPhone 8 Plus, iPhone XR and iPhone XS Max in landscape all have a regular width but an iPhone X and iPhone XS are only compact.

What if you want to have a finer grained control than size classes? We have already seen that [UIContentContainer Protocol](#) has a method that tells a view controller when the size of its view changes. Let's see how we can use it to adapt a layout based on the size (rather than the size class).

Adapting Stack Views For Size

Looking again at our now familiar adaptive stack view I would prefer if it switched to a horizontal layout on an iPad in landscape:



Unfortunately, our stack view only switches to horizontal when the height is compact, and on an iPad, a full-sized App always has a regular height. Let's see if we can change that (see sample code: [AdaptSize-v1](#)):

1. Instead of using `traitCollectionDidChange(_ :)` to react to a size class change we can use `viewWillTransition(to:with:)` and work with size changes:

```
override func viewWillTransition(to size: CGSize, with coordinator: UIViewControllerTransitionCoordinator) {
    super.viewWillTransition(to: size, with: coordinator)
    if size != view.bounds.size {
        configureView(for: size)
    }
}
```

Remember that first, we need to call `super`. Rotating the device 180 degrees doesn't change the size, but we still get called, so we check if the size has changed before configuring the stack view. Note that the size change has not yet happened, so we get the size from the bounds of the view.

2. The `configureView(for:)` method takes the new size and tests if we have a wide layout (`width > height`) and if so switches the stack view to use a horizontal axis:

```
private func configureView(for size: CGSize) {
    if size.width > size.height {
        stackView.axis = .horizontal
    } else {
        stackView.axis = .vertical
    }
}
```

3. If you build and run at this point on a device in portrait orientation, it may appear to be working. The stack view starts vertical and rotates to horizontal when you rotate the device. Unfortunately, if you launch the App with the device in landscape, you still start with a vertical stack view. What's going wrong?

When we first looked at [UIContentContainer Protocol](#) I mentioned that its two methods are not called when a view is first added to the view hierarchy by the view controller. So we are only configuring the stack view when we rotate the device. Where could we do the initial configuration of the stack view?

4. Your first thought might be `viewDidLoad`:

```
override func viewDidLoad() {
    super.viewDidLoad()
    // *** DON'T DO THIS ***
    configureView(for: view.bounds.size)
}
```

This will most likely work if you try it so what's wrong with using `viewDidLoad`?

5. If we go back to the start of this book when we were looking at manual frame-based layout, we saw a similar problem with [viewDidLoad and Friends](#). The view controller's view is not yet added to the view hierarchy (`view.superview` is `nil`), so you cannot rely on its size (or trait collection) being correctly set in `viewDidLoad`.

It's easy to forget if you're in the habit of setting up your initial constraints in `viewDidLoad`. The difference is that constraints are relationships that describe the layout we want and don't (usually) need the actual size of a view when you create them. So where should we configure our stack view?

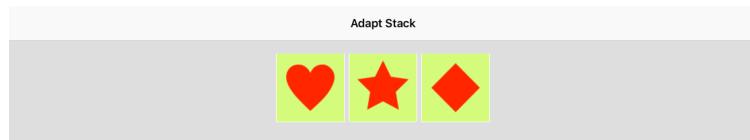
6. We saw the answer when looking at [Where To Put Manual Layout Code?](#). When the system calls `viewWillLayoutSubviews()` the view is in the view hierarchy, and we can rely on its size and traits. We need to limit the work we do in it as the system can call it a lot including several times during rotation.

To make sure we are quick I'm using a flag so that we only set up the stack view once:

```
private var initialSetupDone = false

override func viewWillLayoutSubviews() {
    super.viewWillLayoutSubviews()
    if !initialSetupDone {
        configureView(for: view.bounds.size)
        initialSetupDone = true
    }
}
```

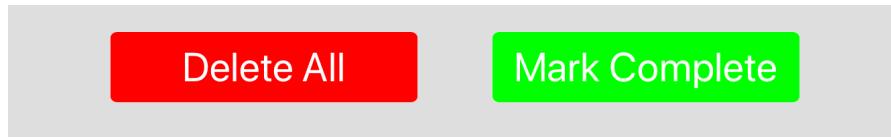
7. If you build and run now the stack view layout should be as we expect even if we launch in landscape:



Adapting Constraints For Size

Let's try this with a more complicated example where we cannot rely on a stack view to do the work for us. I used this style of layout to show

[Equal Spacing With Layout Guides](#). Here's how it looks in landscape on an iPhone 8:



The two buttons are of equal width. Layout guides create equal spacing between each button and the margin and between the two buttons. My button labels are too wide to fit without truncation in the compact width of a portrait iPhone. So for narrow widths, I want to switch to a vertical layout:

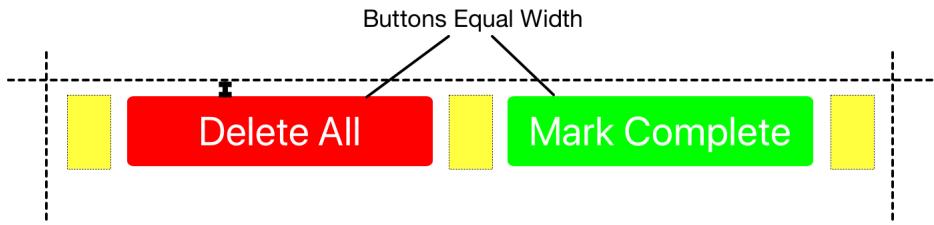


I don't want to use this narrow layout for all situations where we have a compact width. The iPad split-screen modes where the App has at least 50% of the screen are plenty big enough to use the horizontal layout.

Let's make this layout adapt to use the wide, horizontal layout any time the available screen width is more than 500 points. Otherwise, it should use the narrow vertical layout (see sample code: [AdaptConstraints-v1](#)):

1. I'll skip the details on the creation of the buttons, layout guides, and general setup. Instead, let's focus on the three different sets of constraints that we need:
 - Common constraints used by both layouts.
 - Wide layout constraints ($\text{width} > 500$)
 - Narrow layout constraints ($\text{width} \leq 500$)
2. Let's start with the common constraints. If you struggle with this try creating both layouts independently and then extract the common constraints.

There are only two constraints in common to both layouts. A vertical spacing constraint to pin the delete button to the top margin and an equal width constraint between the two buttons:



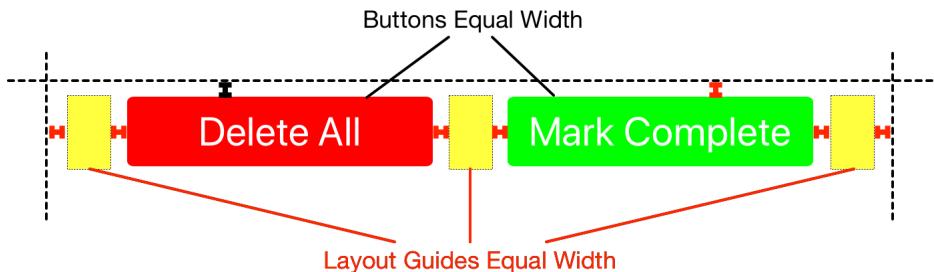
I create these in a lazy property of the view controller. See the sample code for the details:

```
private lazy var commonConstraints: [NSLayoutConstraint] = { ... }()
```

3. We always want the common constraints so I activate them from `viewDidLoad`:

```
NSLayoutConstraint.activate(commonConstraints)
```

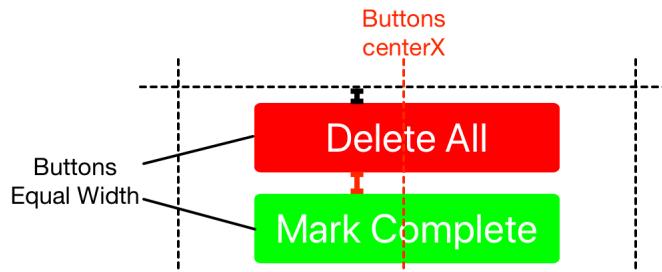
4. The wide layout constraints (shown in red) together with the common constraints create the horizontal layout:



```
private lazy var wideConstraints: [NSLayoutConstraint] = { ... }()
```

Note that we don't activate these constraints yet.

5. The narrow layout constraints (shown in red) together with the common constraints create the vertical layout:



```
private lazy var narrowConstraints: [NSLayoutConstraint] =  
{ ... }()
```

6. With the constraints created we need a method to switch between the two sets based on the width of the view:

```
private func activateConstraints(for width: CGFloat) {  
    if width > ViewMetrics.narrowLayoutLimit {  
        NSLayoutConstraint.deactivate(narrowConstraints)  
        NSLayoutConstraint.activate(wideConstraints)  
    } else {  
        NSLayoutConstraint.deactivate(wideConstraints)  
        NSLayoutConstraint.activate(narrowConstraints)  
    }  
}
```

To avoid conflicts, we deactivate one set of constraints before activating the other set.

I defined `narrowLayoutLimit` as part of an enum for the various view metrics I'm using:

```
private enum ViewMetrics {  
    static let narrowLayoutLimit: CGFloat = 500.0  
    ...  
}
```

7. We use the `viewWillTransition(to:with)` method to switch constraints when the size of our view changes:

```
override func viewWillTransition(to size: CGSize, with coordinator: UIViewControllerTransitionCoordinator) {  
    super.viewWillTransition(to: size, with: coordinator)  
    if size != view.bounds.size {  
        activateConstraints(for: size.width)  
    }  
}
```

8. Finally, we need to set the initial state of our layout. Since this depends on the size of the view we do it as a one-time activity from `viewWillLayoutSubviews`:

```
private var initialSetupDone = false  
  
override func viewWillLayoutSubviews() {
```

```
super.viewWillLayoutSubviews()
if !initialSetupDone {
    activateConstraints(for: view.bounds.width)
    initialSetupDone = true
}
}
```

Key Points To Remember

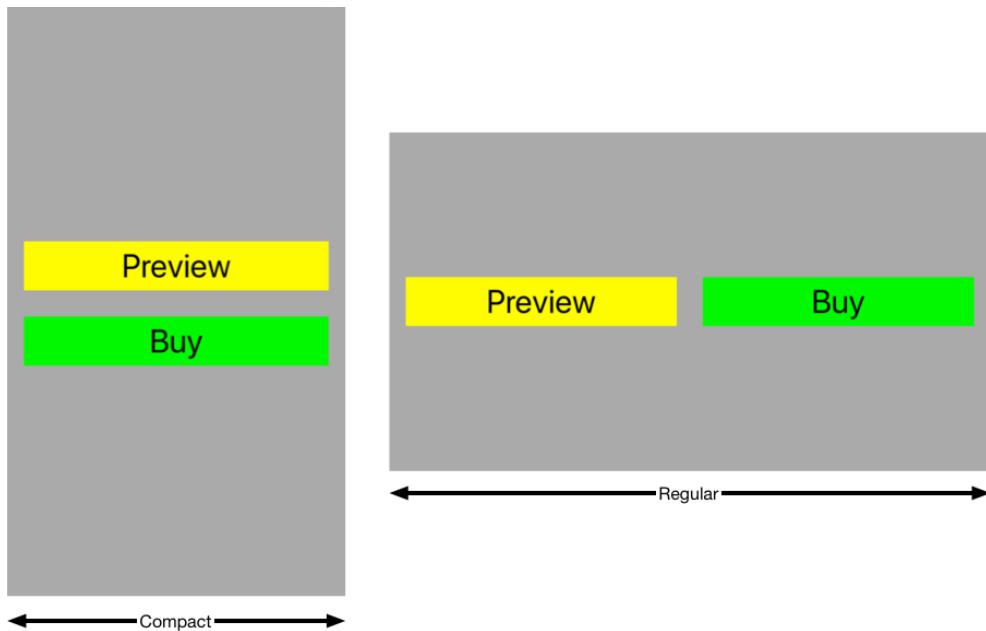
- You get the horizontal and vertical size classes of a view or view controller from its `traitCollection` property. Prefer designing your layout for the size class over the device idiom (iPhone or iPad) or orientation (portrait or landscape).
- Don't forget about managing the keyboard in iPad split-view modes. The keyboard can cover your App even if you don't accept text input.
- Adapt your layouts to size class changes using Interface Builder or by overriding `traitCollectionDidChange` in your view or view controller.
- A stack view saves you work when you want to switch between horizontal and vertical layouts or when you want to show or hide views selectively.
- To animate changes to your layouts when the size class or view size changes use the `UIContentContainer` methods in your view controller.
- Don't forget that you can add size class specific images to the asset catalog.
- If you're struggling to squeeze text with a fixed font into the available space try using localization strings dictionaries to create variations for different widths.
- Horizontal and vertical size classes give you a crude measure of how much screen space is available. When you need finer control create layout variations based on the available space. Use the content container methods to react to size changes in your view controller.

Test Your Knowledge

Practice building adaptive layouts with these challenges:

Challenge 14.1 Adapting A Stack View

This layout has two buttons that use a vertical layout with a compact width but switch to a horizontal layout when the width is regular. Here's how it looks on an iPhone 8 Plus which has a regular width in landscape:



I vertically centered the buttons in the root view. They use a 40 point system font and 32 points of spacing. The buttons fill the available width equally between the margins.

1. I suggest trying to build this layout both using Interface Builder and in code.
2. Test the layout on a variety of iPhone and iPad devices. For example, on an iPhone XS, you should see the compact vertical layout in both portrait and landscape. On a full-screen iPad, you should always see the regular horizontal layout.

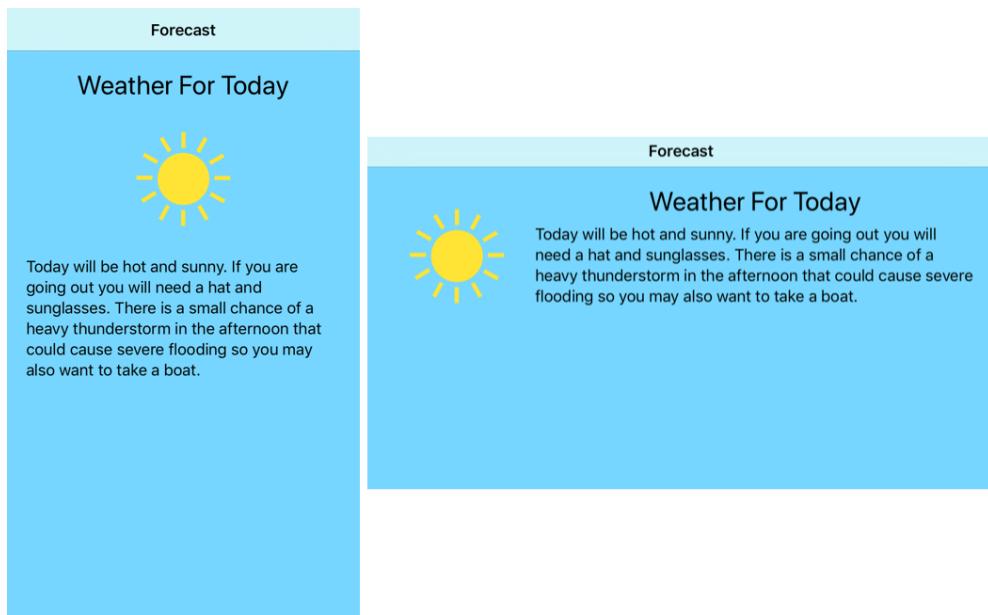
Hints And Tips

1. This layout is ideal for a stack view.
2. Pin the stack view to the leading and trailing margins of the root view and center it vertically.
3. If using Interface Builder, you need to create a variation on the stack view axis property. The axis should be horizontal when the width is regular and vertical otherwise.

4. What distribution should you use for the stack view? The default `.fill` is fine for the vertical layout but not the horizontal.
5. You need to use a `.fillEqually` for the horizontal layout to give the buttons equal width. Since we are not constraining the height of the stack view, you could also use `.fillEqually` for the vertical layout. Alternatively, create a variation for the distribution to switch to `.fillEqually` for a regular width.
6. If building this layout in code, override `traitCollectionDidChange` to configure the stack view.

Challenge 14.2 An Adaptive Layout

Create this layout for a weather forecast App that shows a forecast for the day. Here's how the view looks on an iPhone 8 in both portrait and landscape:

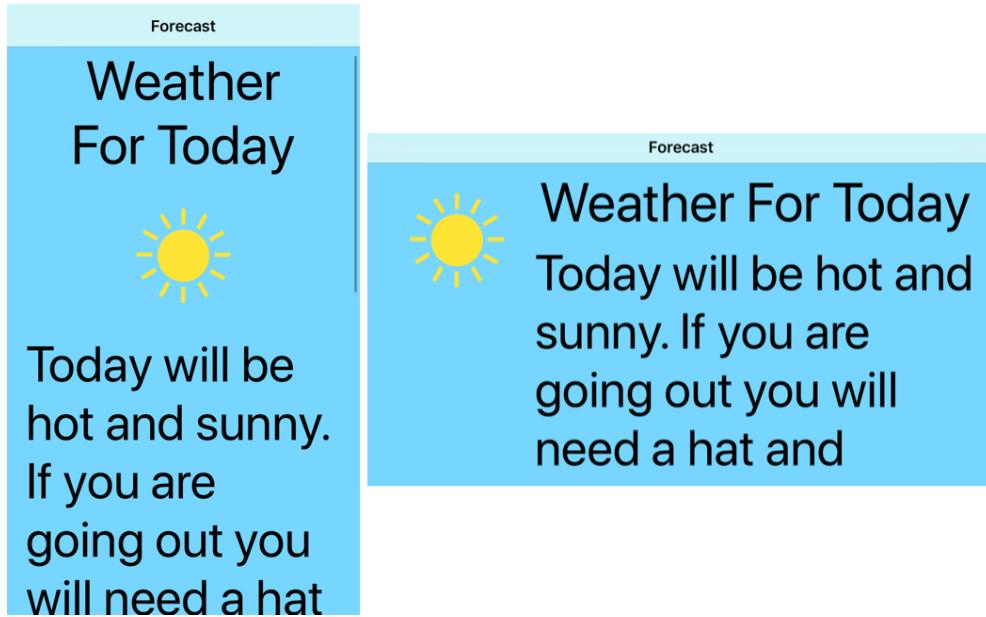


The layout switches the text to the side of the image when there's a compact height (landscape on iPhone devices). The image moves from being horizontally centered and below the title to the top-left corner.

I'm using dynamic type. The top title label is using the Title 1 style, and the bottom summary label is using the Body style. The image is 150 x 150 points in size. When both the height and width are regular, the image is 300 x 300 points.

The labels should flow across multiple lines if required to allow for the user's preferred text size. The view needs to scroll vertically at the larger

text sizes. Here's how it looks using the largest accessibility size:



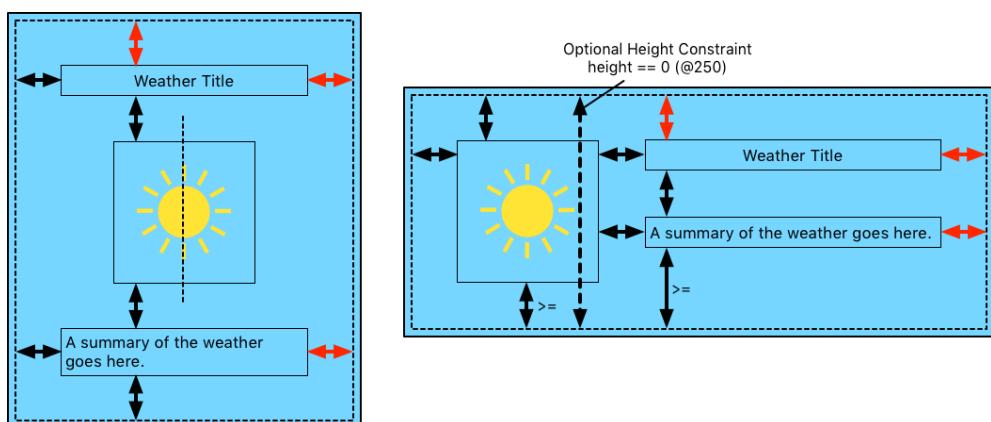
1. Build this layout to work across all devices and preferred text sizes.
2. Use the side layout when the height is compact.
3. The view must scroll to keep the text visible without truncation when it's too large to fit on the screen or when covered by the keyboard in an iPad split-screen.
4. I'm using a 20 point margin on all sides with a standard amount of horizontal and vertical spacing between views.
5. The text should keep a readable line length. Here's how it looks on an iPad Pro 10.5" using the default preferred text size and double sized image:



Hints And Tips

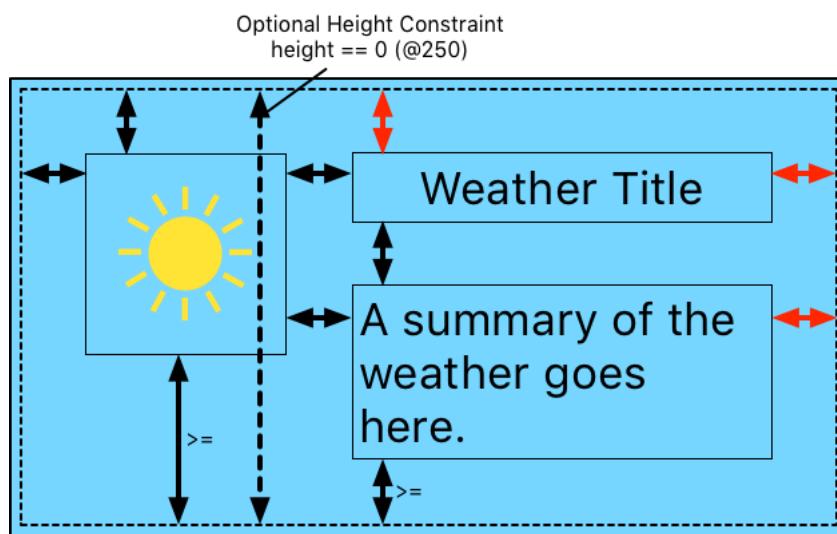
Some general hints and tips that apply to both Interface Builder and programmatic layouts:

1. This layout is not easy, so don't worry if it takes you time to get it right. You need to use techniques from several different parts of this book to build this layout.
2. I recommend you start by first building and testing the regular height layout (portrait on iPhone). When you have it fully working take a look at the compact height layout.
3. You may be tempted to start with a stack view, but I find that makes adapting the layout more difficult.
4. Don't forget to set the number of lines for the labels to zero and enable the automatic adjustment for content size categories.
5. I recommend using a container view to hold the two labels and image view. Add the container view to a scroll view and then pin the scroll view to the edges of the root view. You need 9 constraints for the scroll view. See [Scroll Views And Auto Layout](#) for a recap.
6. You don't need any code to make the image size adapt. Add an image variation to the asset catalog as explained in [Using Traits With The Asset Catalog](#).
7. Ignoring the scroll view, these are the constraints I used internally to the container view for the two layout variations (common constraints in red):



8. I created the regular layout with 9 constraints using the readable content guide of the container view.

9. The compact height layout is more work. When the text is small, the height of the image view sets the height of the container. When the text is large, the labels and the vertical spacing set the height. Here's the same layout at a larger text size:



This is similar to a layout we saw when looking at [Layout Priorities](#). The bottom margin of the container view is *at least* below or equal to the bottom of the image view and *at least* below or equal to the bottom of the summary label. So both constraints use an inequality. To keep the container view height as small as possible, I use an optional height constraint.

10. There's one other complication with the compact height layout. You need to adjust the horizontal content hugging and compression resistance priorities of the views.
11. I always want the image to be at its intrinsic content size. So it needs to have higher horizontal content hugging and compression resistance priorities than the two labels.

Some hints and tips to building this layout with Interface Builder:

1. It can be a challenge to build these layout variations in Interface Builder. I built the regular height layout first using the iPhone 8 in portrait. I then used the preview control to rotate the device to landscape and create a variation for compact height.
2. I find it less confusing to work view by view using the inspector to delete those constraints that don't apply to the compact variation (don't delete constraints in the document outline). You can then move the views to their new positions and add the constraints for

this variation.

Some hints and tips if you're building this layout in code:

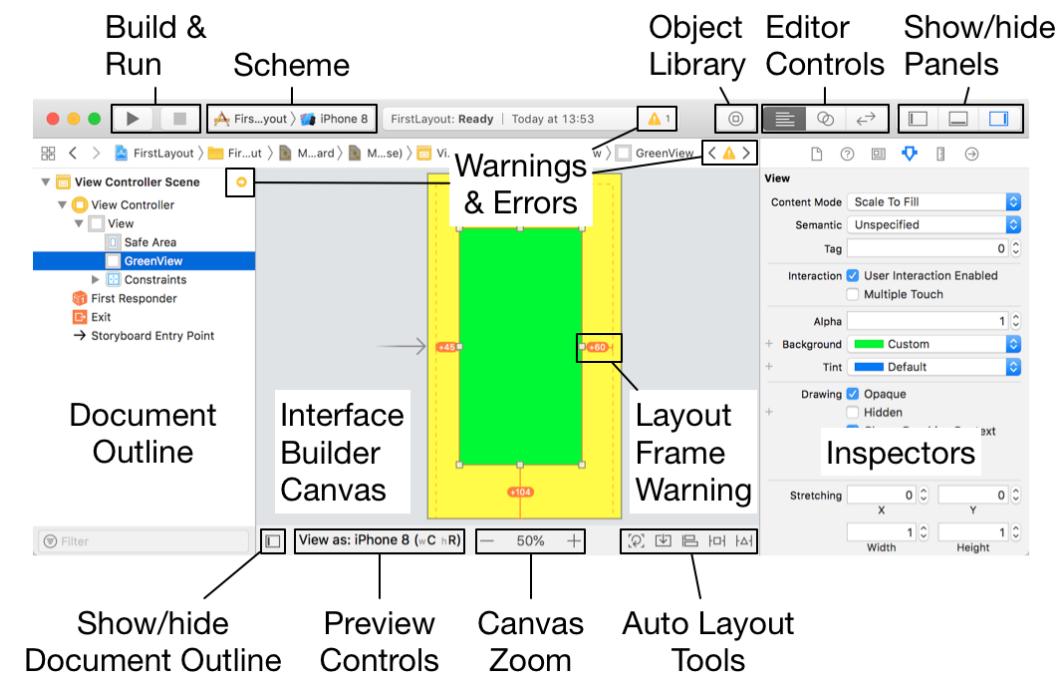
1. I suggest you create a subclass of `UIView` for the container view that holds the two labels and image view. The view controller can then take care of setting up the scroll view. Set the container view margin to 20 points.
2. You need three sets of constraints. Common constraints, regular constraints and compact constraints. The common constraints are always active. Switch between the regular or compact constraints depending on the vertical size class.
3. Override `traitCollectionDidChange` in your view controller (or custom subview if you're using one) to switch between layouts. Remember to deactivate the old constraints before activating the new constraints when the vertical size class changes.

Appendix A

Tour Of Interface Builder

If you're new to Xcode spend some time working through the guides I suggest in the [Introduction](#). The Xcode Help guide (Help > Xcode Help) is also worth reading to get you started. I'm assuming you know the basics of creating and running an iOS app on a device or the simulator. What follows is a brief recap of the Interface Builder features that you use most often with Auto Layout.

When you open a storyboard or xib file in Xcode the editor window switches to show the document using Interface Builder:

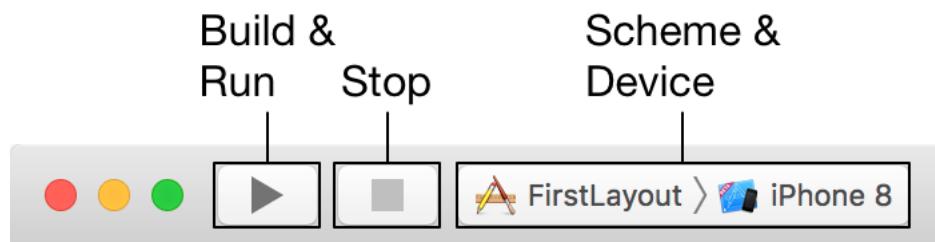


I hid the left side file navigator and bottom debug panels, but that's still

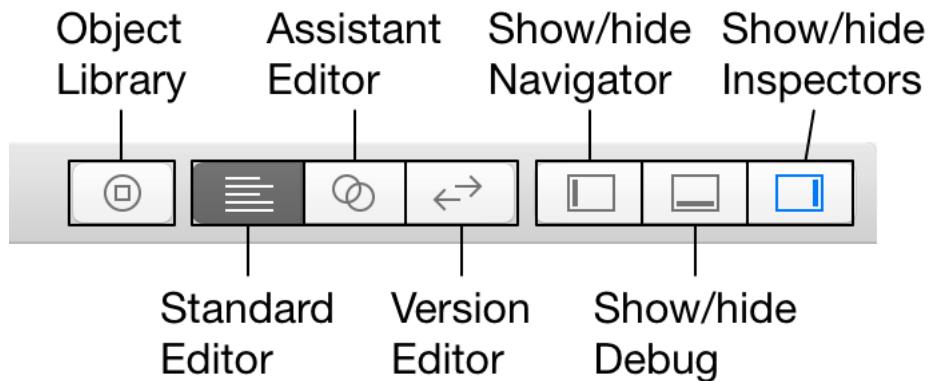
a busy user interface! Let's walk through the important bits for creating layouts:

Xcode Toolbar

As well being a general status area, the Xcode toolbar is where you find the build and run and stop buttons, choose the target and select the device or simulator to run on:

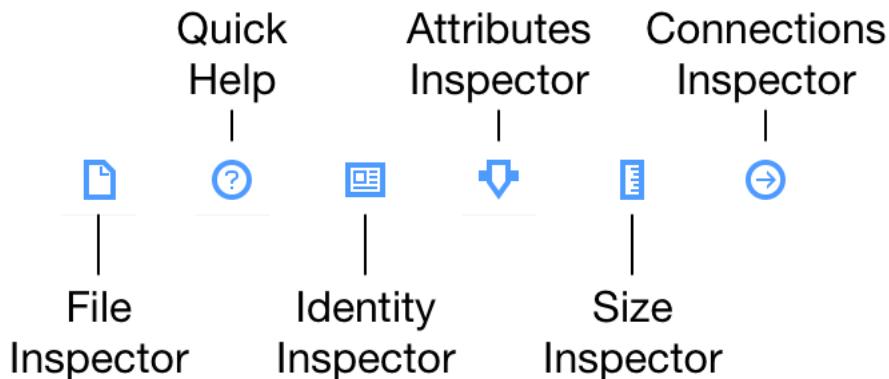


The controls to show the object library, switch between editors and show/hide the extra panels are to the right:



Inspectors

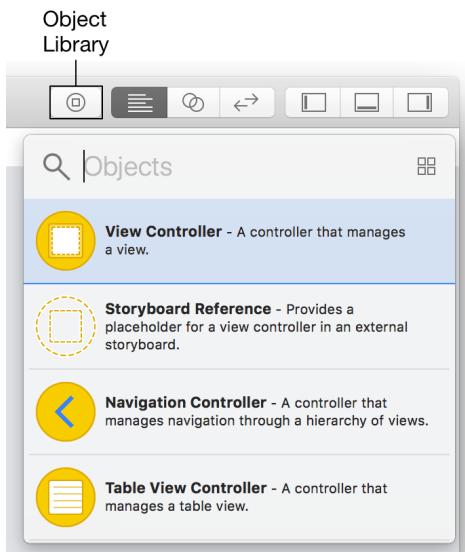
The utilities/inspectors pane is on the right. Use the button in the Xcode toolbar to show or hide it. Take the time to get familiar with each of the six inspectors as you use them a lot when working with Interface Builder:



- **File Inspector** Control if this storyboard or XIB file uses Auto Layout, trait variations and safe area layout guides (for new projects these all default to on).
- **Quick Help** Brief help summary for selected item.
- **Identity Inspector** Set the custom class and storyboard ID.
- **Attributes Inspector** The attributes you see depend on the type of the item you inspect. Set the background color, hide a view or for text-based views the font or number of lines. Choose a constraint to directly edit its attributes here (or using the size inspector).
- **Size Inspector** With a view selected you can see and edit the constraints involving the view as well as change content hugging and compression resistance priorities. You can also set and control layout margins for a view.
- **Connections inspector** Show connections from selected object to outlets and actions.

Object Library

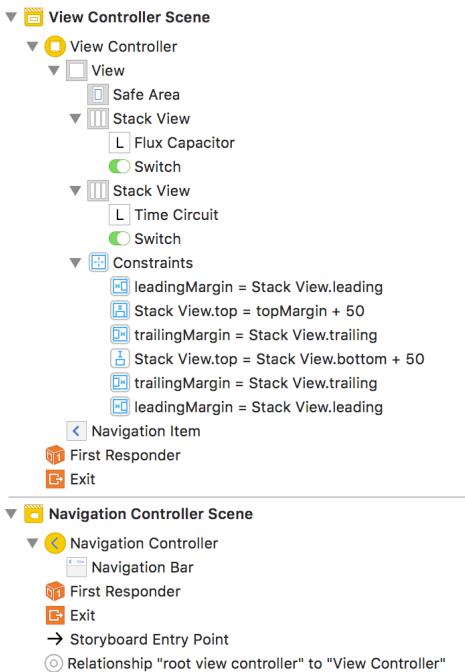
Use the button in the Xcode toolbar to open the object library and drag objects into the canvas area. To quickly find an object start typing the name in the filter box:



Use the Option key when opening the object library or when dragging an item from the library to keep it open after making a selection.

Document Outline

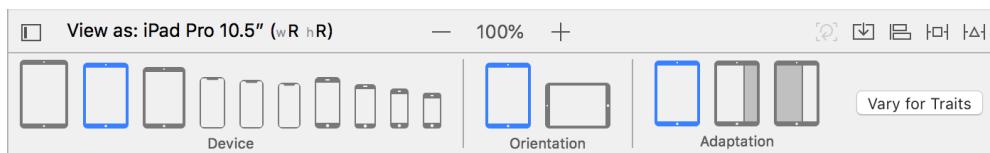
The document outline is to the left of the Interface Builder canvas. You can show or hide it using the button in the Interface Builder toolbar.



1. The outline shows each scene with the view hierarchy nested below the owning view controller.
2. Auto Layout constraints are one level below the view that owns them.
3. Control-drag between views and the view controller to connect outlets or actions. Control-drag between views to create constraints between those views.

Preview Controls

Clicking on the preview control in the Interface Builder toolbar opens the preview control panel:

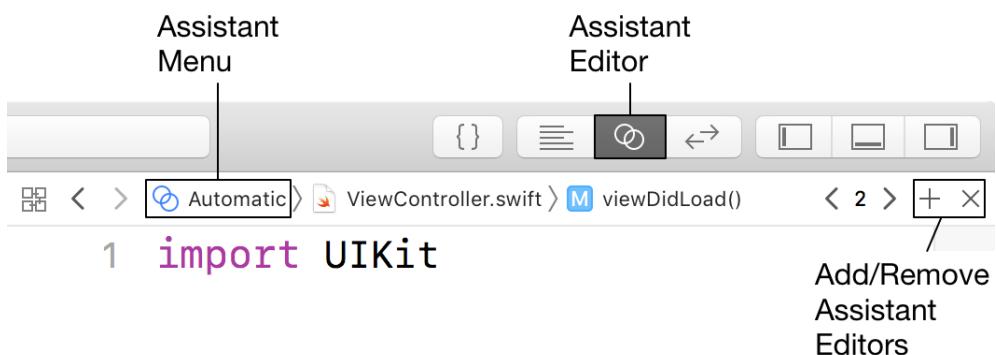


Choose one of the supported iPhone and iPad devices to change how you see the layout in the Interface Builder canvas. You can also switch between portrait and landscape orientation and for iPads choose one of the split view modes.

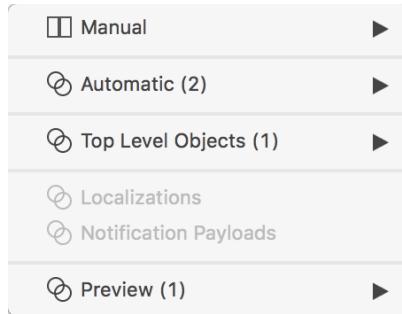
The Vary for Traits button allows you to create layout variations for the different horizontal and vertical size classes. See [Size Classes](#) for more details.

Assistant Editor

Show the Interface Builder assistant editor using the button in the Xcode toolbar. Use the **[+]** and **[x]** buttons to create and remove extra assistant editors if you need them and have the screen space:



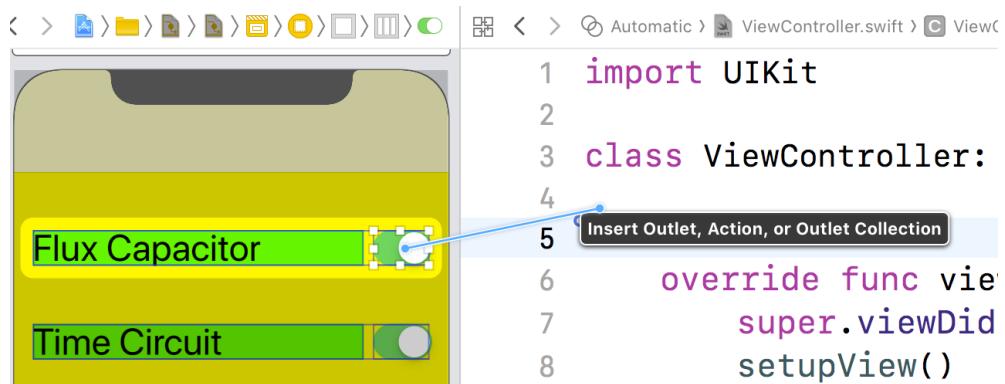
Use the assistant menu in the jump bar of the assistant editor to choose what to show:



When used with Interface Builder the assistant editor can help in several ways:

Creating Connections

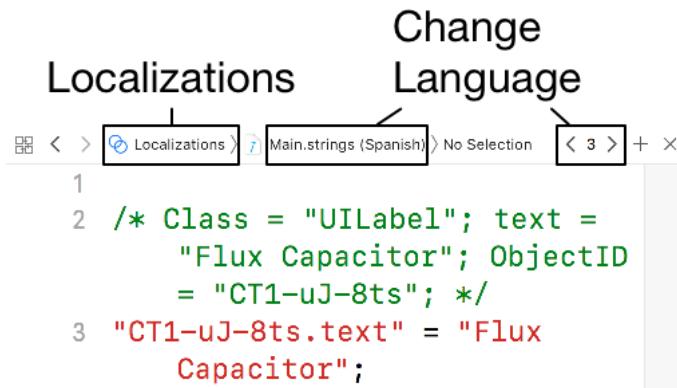
If you choose Automatic from the assistant menu, the assistant editor shows the view controller code for the storyboard scene you're working on in the canvas. You can also manually choose a file to show from the popover menu. Control-drag from an item in the canvas or document outline into the code to create IBOutlet or IBAction connections between the code and item:



Note that you can also create outlets for constraints created in Interface Builder. Find the constraint in the document outline and control-drag from it to the code in the assistant editor.

View Localizations

With a storyboard showing in the main editor choose Localizations from the assistant menu to see the strings localization files for that storyboard. The option is only enabled if you have added at least one localization (other than the base localization):

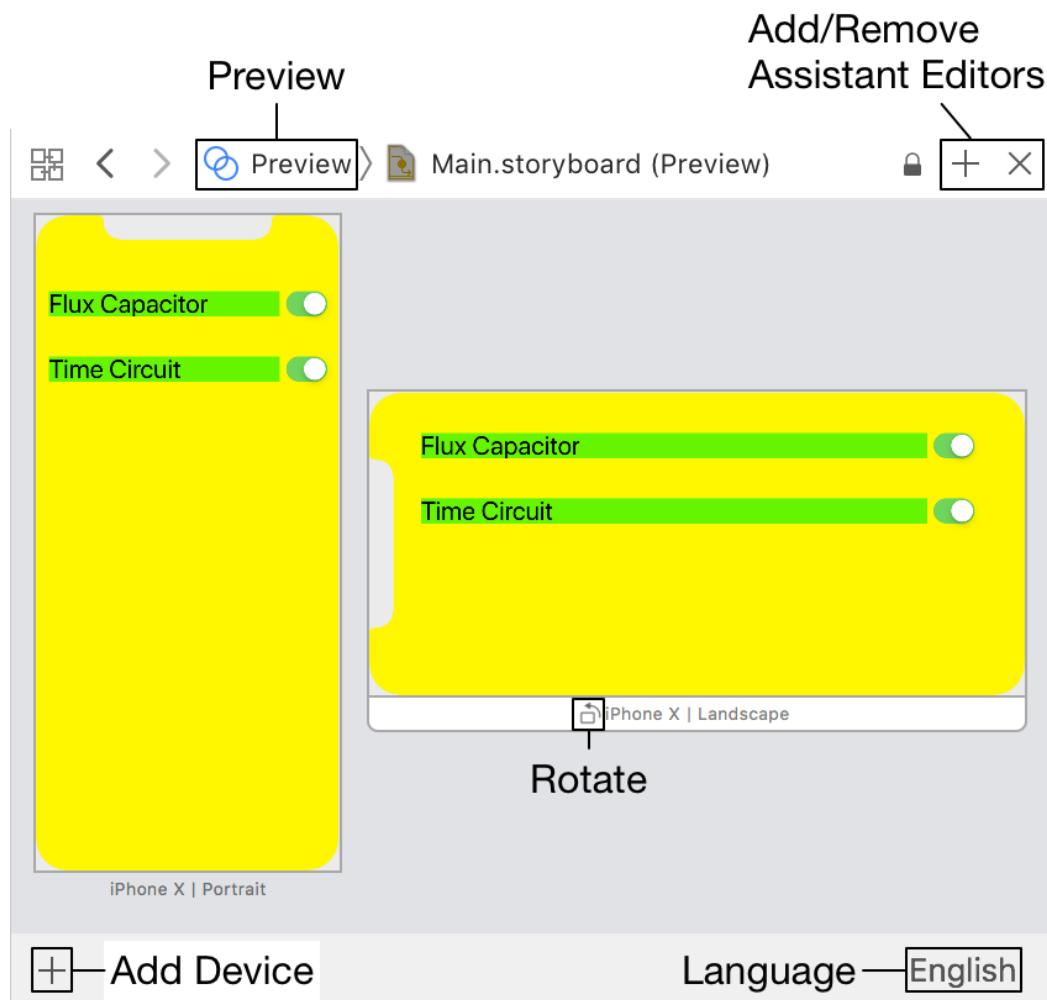


Switch between localization language files using the menu or buttons in the assistant editor toolbar.

Preview Layout

One of the challenges with designing your layout is understanding how it adapts to the different devices sizes. How does a layout created for an iPhone 8 Plus look when squeezed onto an iPhone 4S? Does it still work when you rotate the device from portrait to landscape? Do your text fields fit when using a different, much longer, language?

Use the assistant editor to preview your layout on different devices, orientations and language localizations. Choose **Preview** from the assistant menu, and the assistant editor shows the view controller you're working on in the canvas on a simulated device.



Zoom the preview in and out using pinch gestures or double-click to zoom to fit. Use the controls in the preview pane to add more devices, change the orientation or localization language.

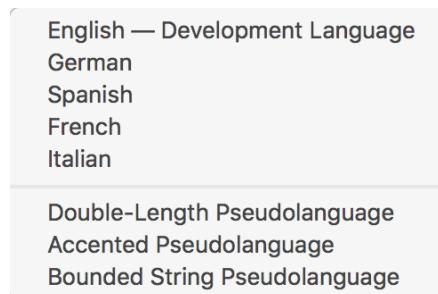
1. Hover the mouse over the caption below each device and click the rotate icon to switch between portrait and landscape:

iPhone X | Landscape

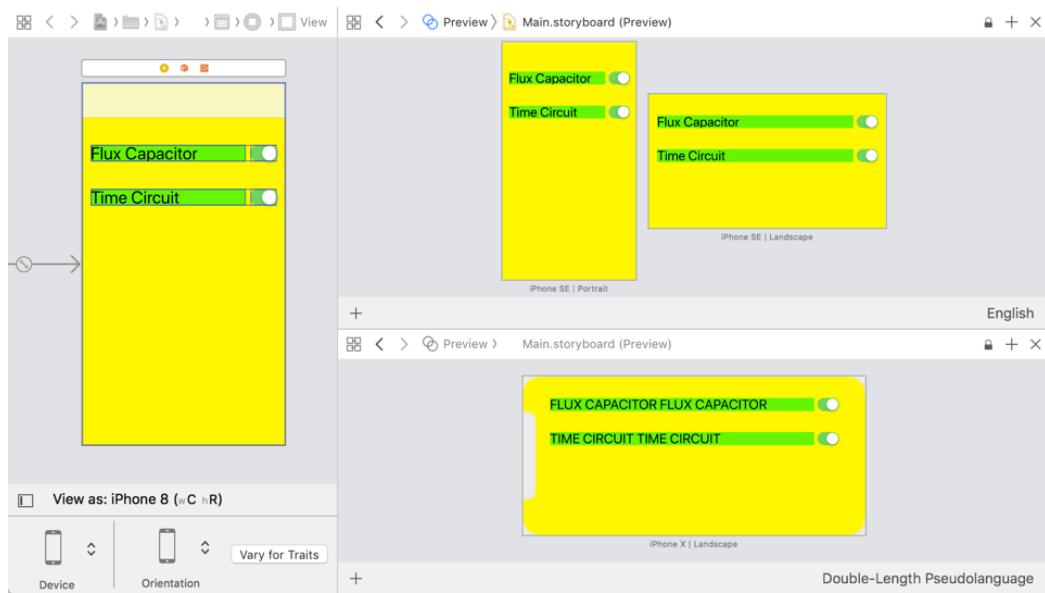
2. Use the **[+]** button in the bottom-left corner to add devices. Use the delete key to remove any devices you no longer want to see:



3. Change the localization language using the language popup menu in the bottom-right corner. You can choose between the available project localizations or use a pseudo language to see how your layout adapts to different size texts:

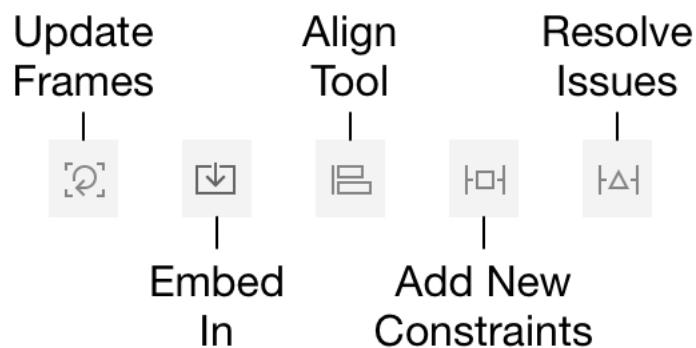


If you have the screen space you can add extra assistant editors with the **[+]** button in the top right corner of the assistant editor. For example, here's a layout in the canvas viewed on an iPhone 8. The first assistant editor on the top right shows the layout on an iPhone SE in portrait and landscape. The second assistant editor on the bottom right shows the layout on an iPhone X in landscape with the language set to "Double-Length Pseudolanguage":



Auto Layout Tools

Interface Builder hides some Auto Layout tools down in the toolbar in the bottom right corner under the canvas. I cover these in detail in [The Many Ways To Create A Constraint](#) but here's a summary:



- **Update Frames** Interface Builder warns you when the position or size of a view on the canvas doesn't match the Auto Layout calculated values. It shows the out of position constraints in orange. Use the update frames tool to update the selected views to the calculated values.
- **Embed In** Embed the selected view controller in a navigation or tab bar controller. Embed selected views in another view with or without an inset, in a scroll view or a stack view.
- **Align Tool** Useful when you want to create constraints that align the edges, centers or text baselines of several views. Also used to

center views in their superview.

- **Add New Constraints** A multi-purpose tool for creating horizontal or vertical spacing constraints between views, giving views a fixed width or height, giving views equal width or height, setting the aspect ratio of views and aligning views on an edge, baseline or center (like the align tool).
- **Resolve Auto Layout Issues** Update, add or reset constraints based on the canvas positions of views. Not as useful as it sounds as it rarely does what you want when resolving issues. I see people use it sometimes to pin a view to the superview by using “Reset to Suggested Constraints” to quickly add the leading, trailing, top and bottom constraints. I recommend ignoring it.

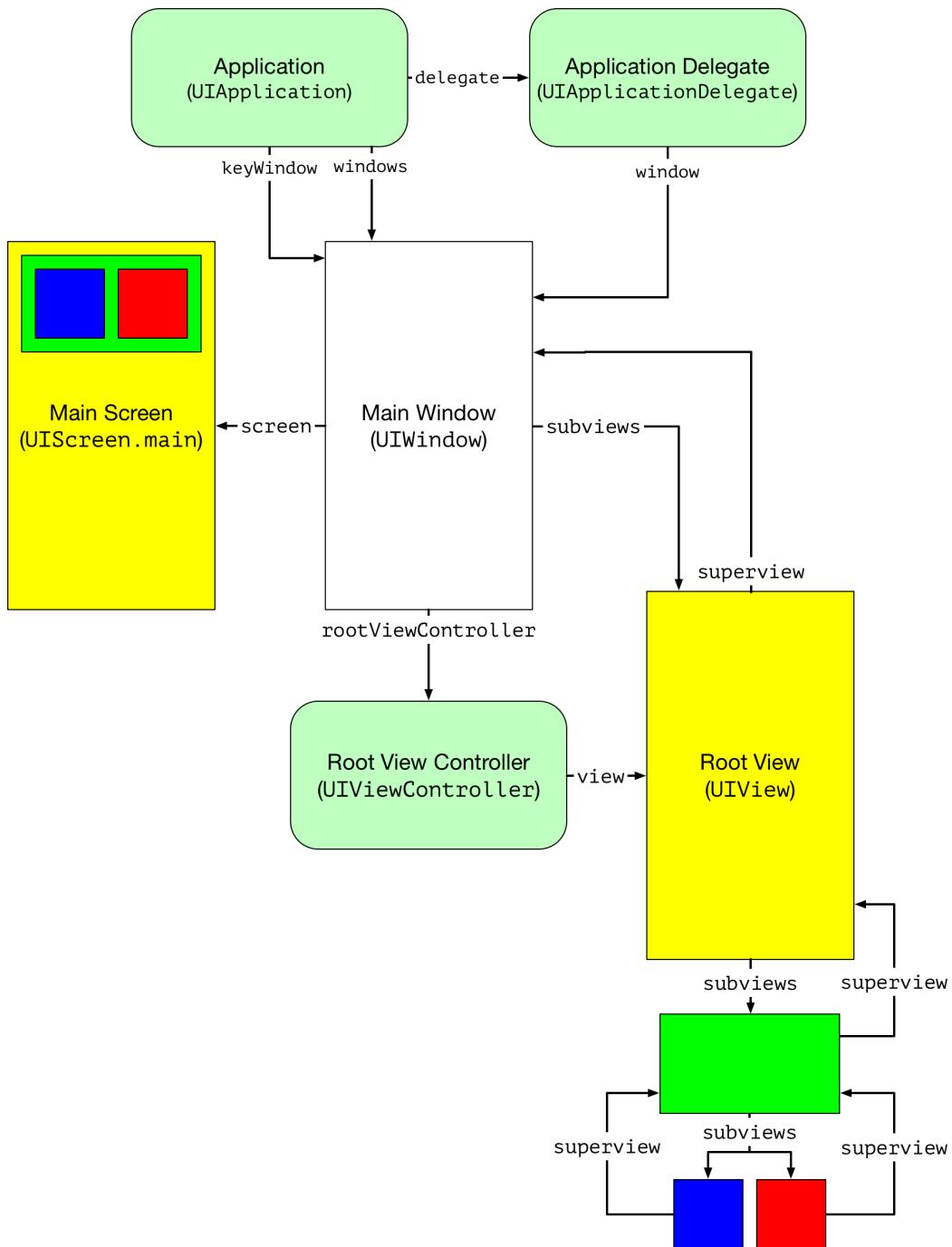
Appendix B

Layout Essentials

If you have been an iOS developer for a while but have a sneaky suspicion, you're missing some basics about how views work don't panic! Here's a quick summary of the essentials of how an iOS app loads its window and views and what you need to know about view geometry.

The View Hierarchy

Unlike many desktop applications, an iOS application usually has a single window that fills the main screen of the device. The content for the window comes from the view of the root view controller. You build your user interface by adding views to this root view:



Screen (UIScreen)

All iOS devices have a main screen, but you can also connect them to an external screen. You access the main screen and any connected screens using type properties of the `UIScreen` class:

```
let mainScreen = UIScreen.main // UIScreen
let screens = UIScreen.screens // [UIScreen]
```

iOS measures screen size in points that are resolution independent. The mapping of a point to a physical pixel depends on the resolution of the device. For example, an iPhone XS has a screen size of 375×812 points which it scales 3x to the native resolution of 1125×2436 pixels. See [Appendix C: Points vs. Pixels](#) for the screen sizes and scale factors of some different iOS devices.

Window (**UIWindow**)

The window is an instance of the **UIWindow** class and is at the root of the view hierarchy that contains our app's user interface. Most apps have a single window for displaying content on the main screen of the device. You might create a second window to show content on an external screen. By default when you create a window, it displays on the main screen unless you assign a different screen.

You don't usually have to think about your app's window. When your app launches, it creates the window for you when loading the main storyboard and sizes it to fill the main screen. If you're not using a storyboard, a few lines of code in the application delegate does the job of creating the window and the initial root view controller. See [Removing The Main Storyboard](#) for the details.

Root View Controller (**UIViewController**)

The root view controller is usually a custom view controller subclass and supplies the view content for the main window. You typically either load it from your main storyboard or create it manually in the application delegate. Either way, the act of making it the root view controller of the main window adds the view controller's view to the main window.

The visible user interface doesn't have to come from a single root view controller. It's common for the root view controller to be a container for other view controllers. The view hierarchy is then a combination of views from the parent container view and the views from the child view controllers. Navigation and tab bar controllers are two examples from **UIKit**, but you can also build your own.

Views (`UIView`)

The content that your app displays comes from its views. These views might be a plain old `UIView`, something more complex like a table view, one of the many `UIKit` controls or even our custom views. These views, including the window, are all subclasses of the basic `UIView` class.

A view has at most a single superview but can have zero, one or many subviews. The `superview` property of a `UIView` can be `nil` if you have not added the view to a view hierarchy. Swift handles values that can be `nil` by making them `Optional`. So `superview` is an optional `UIView`:

```
var superview: UIView? { get }
```

Note that this property doesn't have a setter so you cannot change it directly. Adding or removing a subview sets the `superview` of the subview. A view keeps track of its immediate subviews in an array of `UIView` items:

```
var subviews: [UIView] { get }
```

The order of the views in the array is crucial as it sets the display order from back to front. So the view at index `0` is at the back, and the last view in the array is at the front.

Add a subview to a parent view by calling the `addSubview` method of the parent. This appends the subview to the end of the `subviews` array, so it appears at the front. There are other methods for adding a subview at a specific position or before or after another subview. Some examples:

```
yellowView.addSubview(greenView)
greenView.insertSubview(blueView, at: 0)
greenView.insertSubview(whiteView, aboveSubview: blueView)
greenView.insertSubview(redView, belowSubview: whiteView)
```

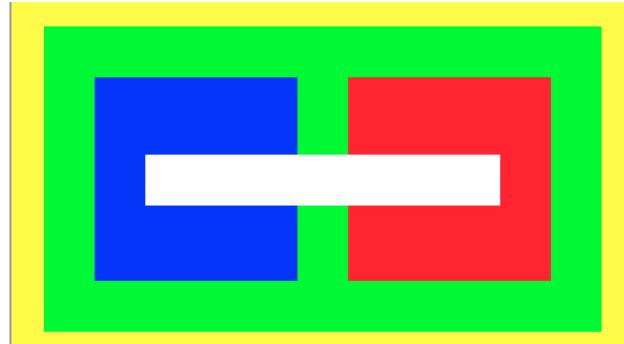
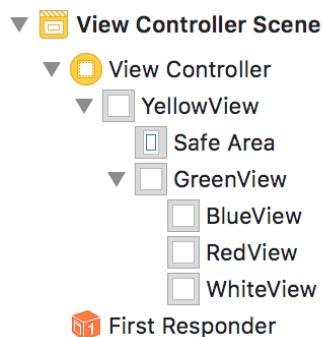
Note that you add or insert subviews to the superview, but when you want to remove a subview you use `removeFromSuperview` on the subview:

```
whiteView.removeFromSuperview()
```

You can also bring a view to the front, send a view to the back or swap the position of two views:

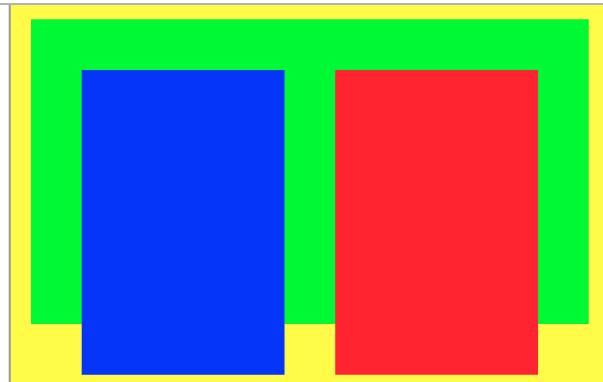
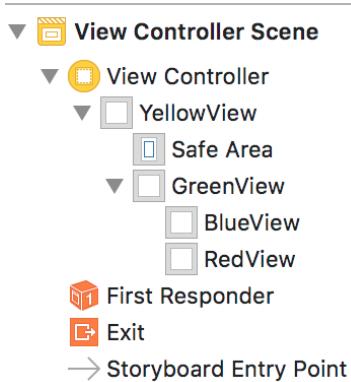
```
greenView.bringSubview(toFront: whiteView)
greenView.sendSubview(toBack: blueView)
greenView.exchangeSubview(at: 0, withSubviewAt: 1)
```

If you use Interface Builder to create your layouts, it handles the view hierarchy for you.

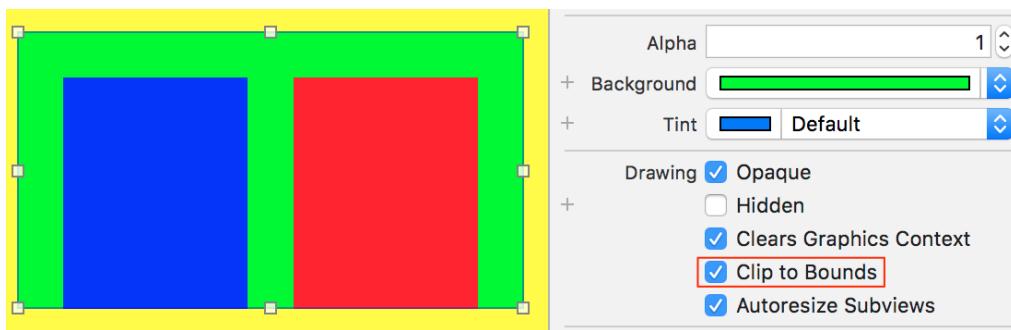


Note the order of the subviews in the document outline. The white view is last and so appears in front of the blue and red views. Drag and drop the views in the document outline to change the order.

What happens if aSubview is too big to fit within the bounds of its superview? By default, nothing prevents aSubview from extending beyond the bounds of its superview. For example, the blue and red views in this example are subviews of the green view but extend beyond the bottom of the green view:



You can change this by selecting the “Clip to Bounds” property for the green view in Interface Builder:



To set the property in code:

```
greenView.clipsToBounds = true
```

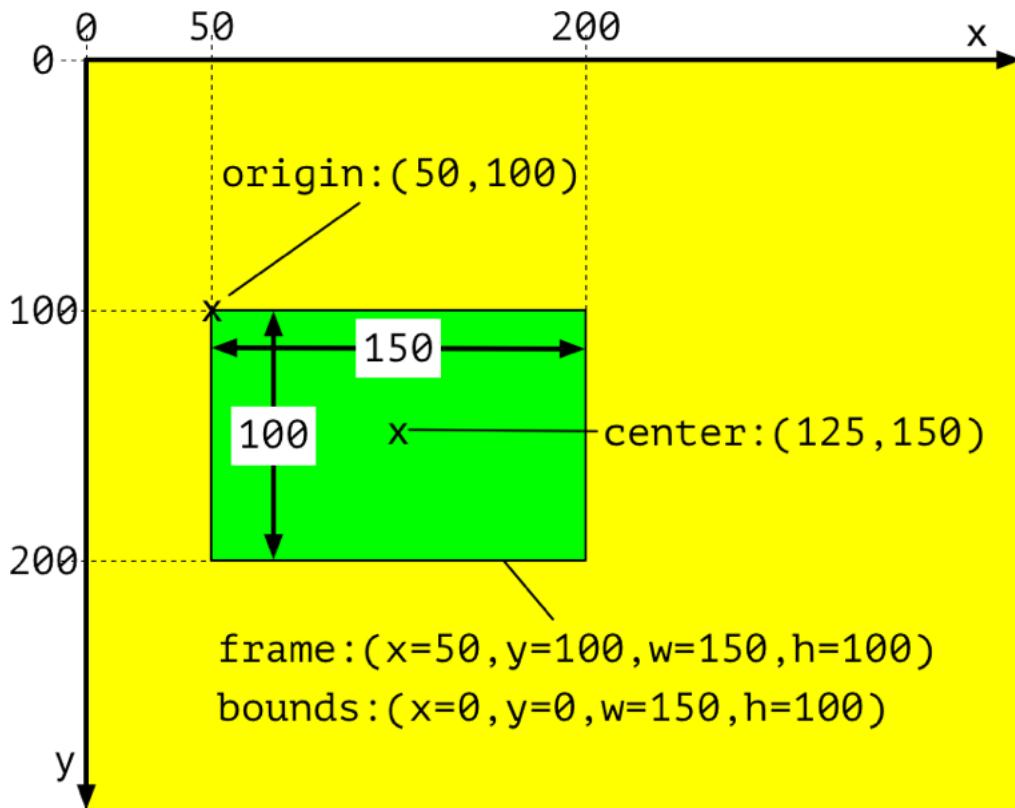
Remember that for your views to be visible they must also eventually be part of a view hierarchy that has the main window at its root. You can check for this with the `window` property of `UIView`. If this property is `nil` for a view, then it's not on-screen.

View Geometry

To build a working layout, you need to set the size and position of each of the views you add. There are four properties of `UIView` that control the size and position of a view:

- **frame:** The rectangle that gives the position and size of the view in the coordinate system of its superview.
- **bounds:** The rectangle that gives the internal size of the view in its coordinate system.
- **center:** The center point of the view in the coordinate system of its superview.
- **transform:** A transform you apply to scale or rotate the view.

The relationship between the `frame`, `bounds` and `center` is easier to understand with an example. Look at this green view which is a subview of the yellow view:



The green view is 150 points wide and 100 points high and has an origin at (50,100). The center of the green view is at (125,150). Remember that both of these coordinates are in the coordinate system of the yellow superview.



The coordinate systems for iOS and macOS are not the same. The origin (0,0) is in the top-left corner for iOS and the bottom-left corner for macOS.

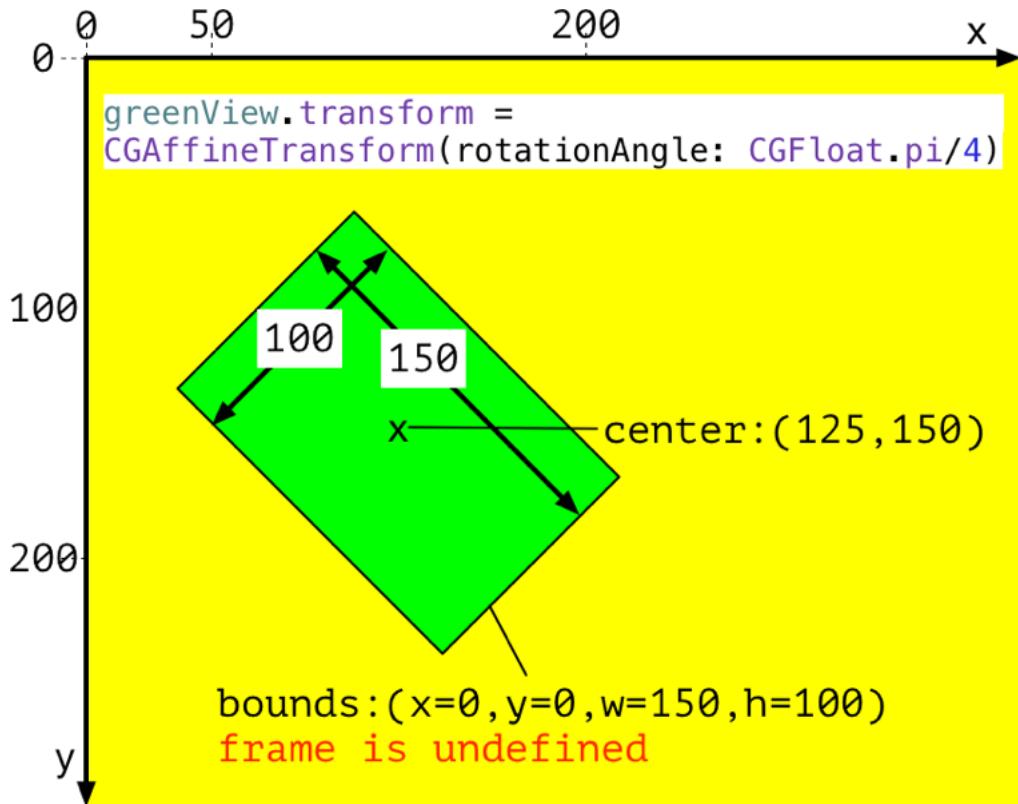
You apply a transform to a view to rotate or scale it. You create a transform using `CGAffineTransform`. Here are some examples:

```
// Rotate by 45 degrees (angle in radians)
greenView.transform = CGAffineTransform(rotationAngle:
    CGFloat.pi/4)

// Scale width x 2 and height x 0.5
greenView.transform = CGAffineTransform(scaleX: 2, y: 0.5)

// Reset transform to identity
greenView.transform = CGAffineTransform.identity
```

Here's what happens to the green view when we apply a 45-degree rotation transform:



The transform doesn't change the center or bounds of the view.



Apple warns that when a view has a transform, other than the identity transform, the frame is undefined. If you need to get the size, or resize or move a view with a transform use the bounds and center.

Some examples to show how to use the frame, bounds and center to manage the size and position of a view:

```
// Use frame to set initial size and position of view
let greenView = UIView(frame: CGRect(x: 25, y: 25, width: 125,
height: 125))

// Use bounds to change size of a view without moving
// center of view
greenView.bounds.size = CGSize(width: 50, height: 50)
```

```
// Use center to move a view  
greenView.center = CGPoint(x: 100, y: 100)
```



When using Auto Layout you should never directly change the frame, bounds or center of a view. Use constraints to describe your layout and let the layout engine take care of sizing and positioning your views.

Core Graphics Data Types

The four most common data types you need when working with views come from the Core Graphics framework.

CGFloat For Numeric Values

Use a CGFloat anytime you need a numeric value for things like view sizes, spacing, and positions. It's a Double on 64-bit platforms and a Float on 32-bit platforms. You'll save yourself some casting if you make your constants and variables of type CGFloat:

```
let spacing: CGFloat = 25.0  
let padding: CGFloat = 8.0
```

If you mix types you will need to cast to CGFloat:

```
let offset = 10 // Int by default  
let x = padding + CGFloat(offset) // x is of type  
CGFloat
```

Note also the type property CGFloat.pi for the mathematical constant (3.14159...):

```
let rotation = CGFloat.pi/4
```

CGPoint For Coordinates

A struct with an x and y value (both CGFloat) for the two-dimensional coordinates of a point. The origin (0,0) is the top-left corner on iOS.

```
struct CGPoint {
    var x: CGFloat
    var y: CGFloat
}

let startPoint = CGPoint(x: 25, y: 25)
```

CGSize For Width And Height

A struct with a width and height value.

```
struct CGSize {
    var width: CGFloat
    var height: CGFloat
}

let size = CGSize(width: 325, height: 175)
```

CGRect For Rectangles

A struct that defines a rectangle with an origin and size. The frame and bounds of a view are both of type CGRect. You can create a CGRect from a CGPoint and CGSize, but it also has initializers that take the four values (x, y, width, height) directly as Int, Double or CGFloat types:

```
// Rectangle
struct CGRect {
    var origin: CGPoint
    var size: CGSize
}

let origin = CGPoint(x: 25, y: 25)
let size = CGSize(width: 325, height: 175)
let rect1 = CGRect(origin: origin, size: size)
let rect2 = CGRect(x: 25, y: 25, width: 325, height: 175)
```

There are functions for working with insets, offsets, intersections and unions:

```
let container = CGRect(x: 100, y: 100, width: 200,  
height: 100)  
let inset = container.insetBy(dx: 20, dy: 20)  
// {x 120 y 120 w 160 h 60}  
let offset = container.offsetBy(dx: 20, dy: 20)  
// {x 120 y 120 w 200 h 100}
```

Note that each struct also has a static type property to represent zero:

```
let zeroPoint = CGPoint.zero // {x 0 y 0}  
let zeroSize = CGSize.zero // {w 0 y 0}  
let zeroRect = CGRect.zero // {x 0 y 0 w 0 h 0}
```

Appendix C

Points vs. Pixels

The iOS (UIKit) coordinate system uses points, not physical screen pixels. The way a point maps to a physical pixel depends on the resolution of the device. On early iPhone devices, one point was equal to one pixel. More recent devices have higher resolution screens where one point scales to 2x or even 3x the pixels.

Auto Layout always works in points but the scale factor becomes important for any images you use in your app. So a 50px × 50px image at standard (1x) resolution would need a @2x image that's 100px × 100px for the iPhone 8. and a @3x image of 150px × 150px for the iPhone 8 Plus.

The table shows the UIKit point size and scale factor and the native pixel size and scale factor of each device. Dimensions in portrait (w × h):

Device	UIKit Size (Points)	UIKit Scale Factor	Native Size (Pixels)	Native Scale Factor
iPad Pro 12.9"	1024 x 1366	2x	2048 x 2732	2x
iPad Pro 10.5"	834 x 1112	2x	1668 x 2224	2x
iPad Pro 9.7" iPad Air 2 iPad Mini 4	768 x 1024	2x	1536 x 2048	2x
iPhone XS Max	414 x 896	3x	1242 x 2688	3x
iPhone XR	414 x 896	2x	828 x 1792	2x

Device	UIKit Size (Points)	UIKit Scale Factor	Native Size (Pixels)	Native Scale Factor
iPhone 8 Plus iPhone 7 Plus	414 x 736	3x	1080 x 1920	2.608x
iPhone X iPhone XS	375 x 812	3x	1125 x 2436	3x
iPhone 8	375 x 667	2x	750 x 1334	2x
iPhone 6s Plus iPhone 6 Plus	375 x 667	3x	1080 x 1920	2.608x
iPhone 7 iPhone 6s iPhone 6	375 x 667	2x	750 x 1334	2x
iPhone SE	320 x 568	2x	640 x 1136	2x

1. On devices like the iPhone 8 Plus the native scale and the UIKit scale factor are different. In those cases, iOS first renders at the UIKit scale and then scales down to fit the native pixel size.
2. If you want to know the UIKit size or scale factor of a screen use the `bounds` and `scale` properties of the `UIScreen` instance. The `nativeBounds` and `nativeScale` properties give you the actual pixel size and scale.

One More Thing

Already A Subscriber?

Are you an iOS developer interested in learning what's new but struggling to keep up? Maybe you're also a little tired of watching all those WWDC videos?

I write regular articles covering what's new in iOS and Swift. Find out what changed in the latest release of Xcode. What new features you need to support? What new bugs will waste your time!

Sign up to my **free iOS newsletter** and I'll send the full text of each new article direct to your inbox so you never miss a post!

Find out more and subscribe

useyourloaf.com/newsletter