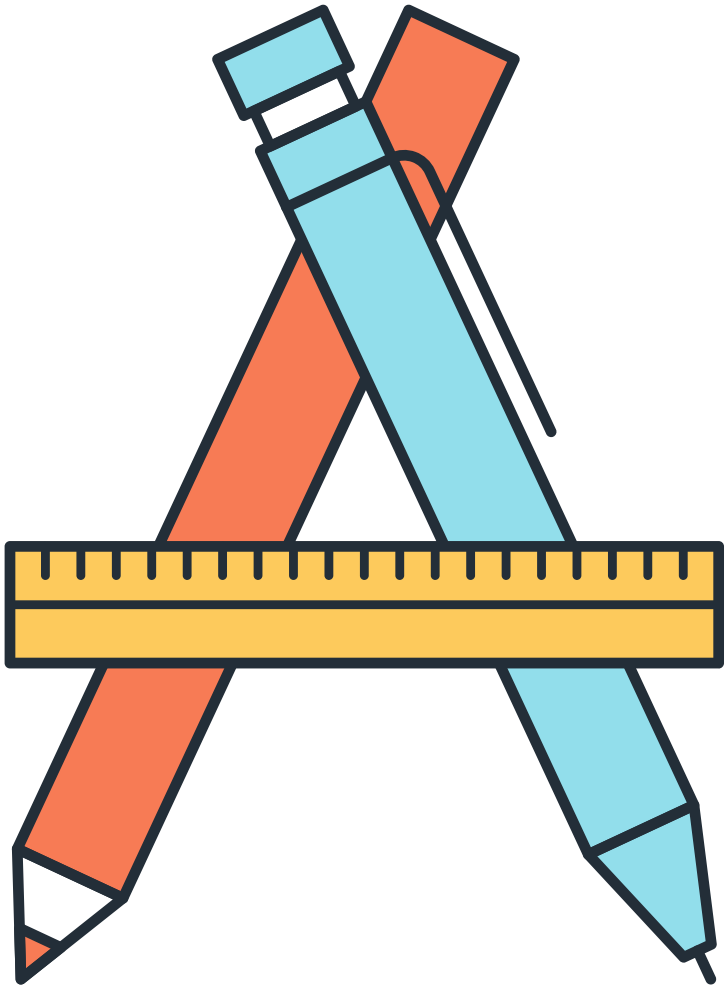
For Xcode 14, Swift 5 & iOS 16

MODERN AUTO LAYOUT

Building Adaptive Layouts For iOS

Keith Harrison



Modern Auto Layout

Building Adaptive Layouts For iOS

Keith Harrison

Web: useyourloaf.com
Version: 7 (2022-11-01)

Contents

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 only needed rebuilding with the latest SDK to support the new screen sizes of the iPhone XR and iPhone XS Max introduced 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 through iOS 16 the Auto Layout API has remained mostly stable. Apple improved the performance of Auto Layout in iOS 12, tweaked the handling of trait collection changes in iOS 13, and changed the stack view implementation in iOS 14 making it respond to background colors. In iOS 15, Apple added a keyboard layout guide, adding it to Interface Builder in Xcode 14.

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. Apple publishes free guides, available in Apple Books, as part of its Everyone Can Code initiative:

- Develop in Swift Fundamentals

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've updated this edition of the book using Xcode 14, iOS 16 and Swift 5.7. The Auto Layout API's have not changed significantly since iOS 11 so most of the code should work unchanged back that far. I do my best to point out changes and how to fall back for earlier iOS versions 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. We also look at using layout margins for extra padding and layout guides as an alternative to spacer views. Finally we take a look at the keyboard layout guide introduced in iOS 15.
- 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 often 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, how to use readable content guides with table views and how to use Auto Layout to create self-sizing table view headers and footers.
- 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 iOS App 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

Change History

Seventh Edition

The seventh edition of this book covers Xcode 14, iOS 16 and Swift 5.7. This is mostly a minor update for the cosmetic changes the Xcode team likes to make each year. See Appendix A for a recap on Interface Builder. I've also streamlined the sections, now of minor interest, covering backwards compatibility with iOS 9/10. A summary of the main changes:

- Chapter 4: Xcode 14 removes preview from the editor options in Interface Builder. Use the device, appearance, orientation and accessibility controls in the canvas toolbar instead.
- Chapter 6 The bugs with keyboard layout guide added in iOS 15 are now mostly fixed. Xcode 14 adds the guide to Interface Builder.
- Chapter 14: Added the iPhone 14 models to the size class graphics.
- Appendix C: Add the iPhone 14 and 2022 iPad devices.

Sixth Edition

The sixth edition of this book covers Xcode 13, iOS 15 and Swift 5.5. A summary of the main changes:

- Chapter 2 Xcode 13 has a different way of managing Info.plist settings. I've updated the procedure to remove storyboards from a project accordingly.
- Chapter 6 Added a new section on the keyboard layout guide added in iOS 15.
- Chapter 12 Dynamic type size can be restricted in iOS 15.
- Chapter 14: The Vary For Traits buttons has been removed in Xcode 13 changing the way you create adaptive layouts in Interface Builder.

- Appendix C: Add the iPhone 13 devices.

Fifth Edition

The fifth edition of this book covers Xcode 12, iOS 14 and Swift 5.3. A summary of the main changes:

- Chapter 4: Xcode 12 renames the default layout mode in the size inspector from “Automatic” to “Inferred” when indicating if you’re using the autoresizing mask or constraints.
- Chapter 8: Stack views in iOS 14 now display their background color. I recommend reviewing the section on Stack View Background Color for details.
- Chapter 14: The userInterfaceIdiom trait property adds the .mac value in iOS 14. There’s also a new trait for activeAppearance that is used on mac platforms to indicate a window should have an active appearance. The size class details have been updated to include the iPhone 12 models.
- Appendix C: I’ve added the screen sizes for iPad and iPhone devices that support a minimum of iOS 13. This includes the latest iPhone 12 devices.

Fourth Edition

The fourth edition of this book covers Xcode 11, iOS 13 and Swift 5.1. A summary of the main changes:

- Chapter 2: iOS 13 adds iPad multiple window support. I don’t cover this in detail but I’ve updated the section on removing the storyboard to show how to setup your Xcode project with or without the scene delegate.
- Chapter 4: I’ve updated the examples for the new Xcode user interface. Also note that Interface Builder has a new layout mode option in the size inspector to show whether a view translates the autoresizing mask into constraints.
- Chapter 9: Interface Builder now shows custom alignment rectangles in the canvas.
- Chapter 11: Scroll view frame and content layout guides are now supported in Interface Builder.
- Chapter 12: Environment overrides in the Xcode debugger make it easier to preview different dynamic text sizes for a running App

(iOS 13 only).

- Chapter 13: Interface Builder now correctly sizes self-sizing table view cells in the storyboard (but not NIB) canvas.
- Chapter 14: There's a potentially layout breaking change for trait collections in iOS 13. UIKit now guesses the likely traits for a view when you create it. If UIKit has guessed correctly you no longer get a call to `traitCollectionDidChange` when adding a view to the view hierarchy. If you've been relying on this for initial view setup in iOS 12 you'll need to make changes for iOS 13.

Third Edition

A summary of the main changes:

- Chapter 6: Added note on a bug in iOS 9 and iOS 10 storyboards that could prevent a custom layout margin from working.
- Chapter 13: Added an example of a self-sizing table view header.

Second Edition

A summary of the main changes:

- Chapter 4: Updated screenshots for the changed appearance of the add constraint menu.
- Appendix C: Added the screen sizes of the new iPad models.

Chapter 7

Layout Priorities And Content Size

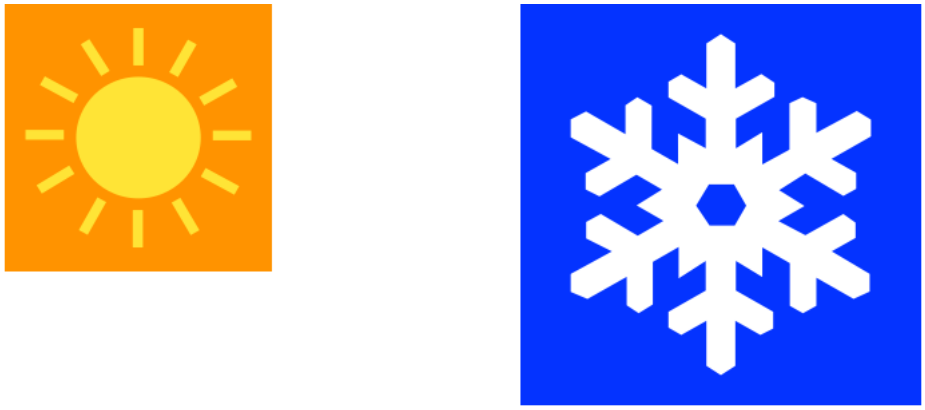
You cannot progress far with Auto Layout without understanding how to use layout priorities. In this chapter you learn:

- 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:

- `.fittingSize` (50)
- `.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. We'll see an example of using the `.fittingSize` priority when we look at Self-Sizing Table View Headers.

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.

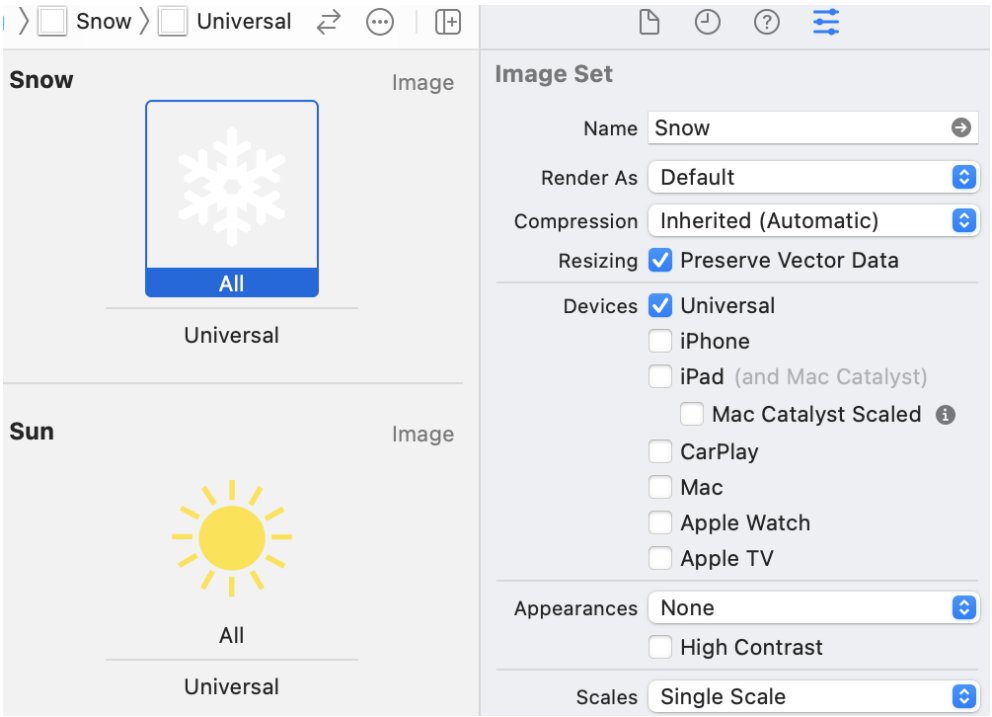
Words like **at least** or **at most** suggest a constraint using **inequalities**. A phrase like **as close as possible** suggests an **optional** constraint.

Optional Constraints In Interface Builder:

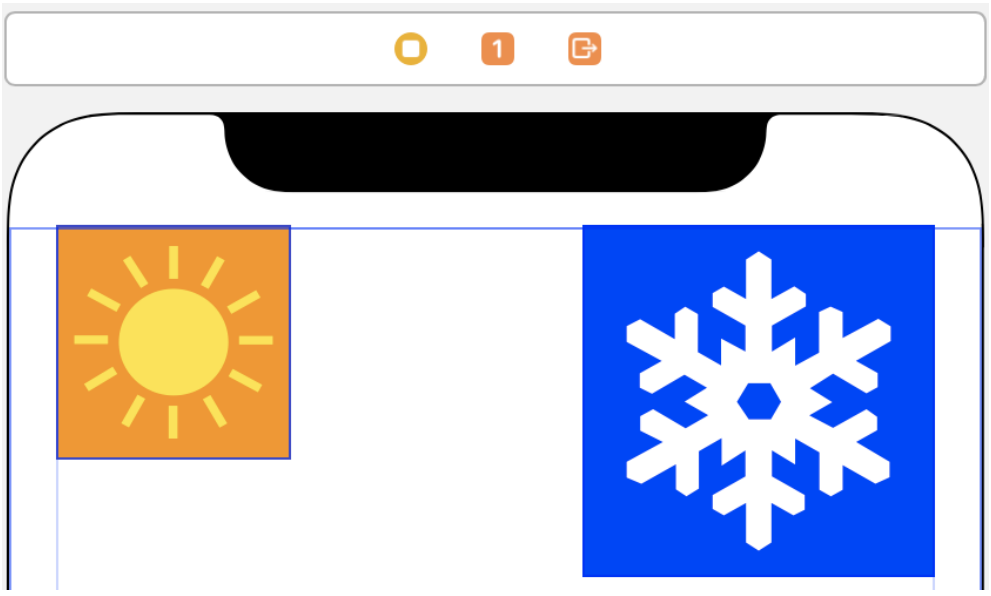
To create this layout, I'm using a sun image that's 100x100 points and a snowflake image that's 150x150 points (see sample code: Priorities-v1). The exact image size is not important. Use mine from the sample code or substitute them with your images:

1. Create a new Xcode project using the iOS App 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:

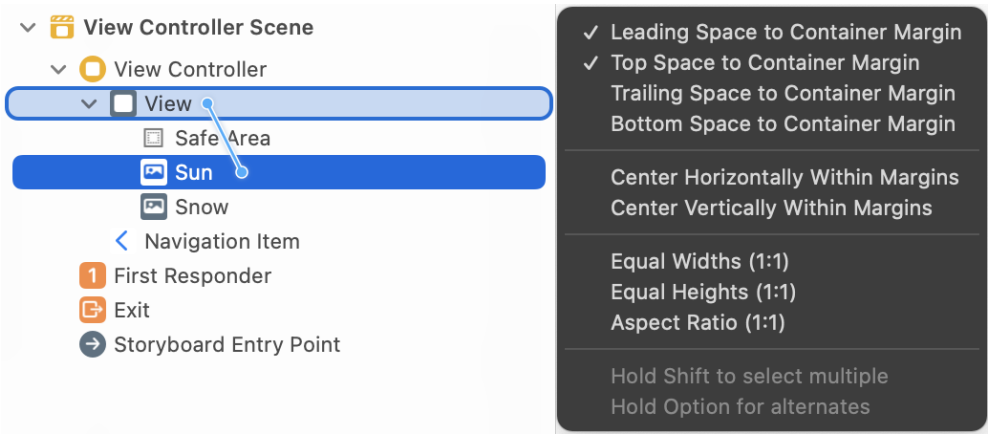


3. Drag two image views from the object library onto the view in the Interface Builder canvas. Position them against the margins towards the top corners of the view and use the Attributes inspector to show the sun image on the left and the snow image on the right:

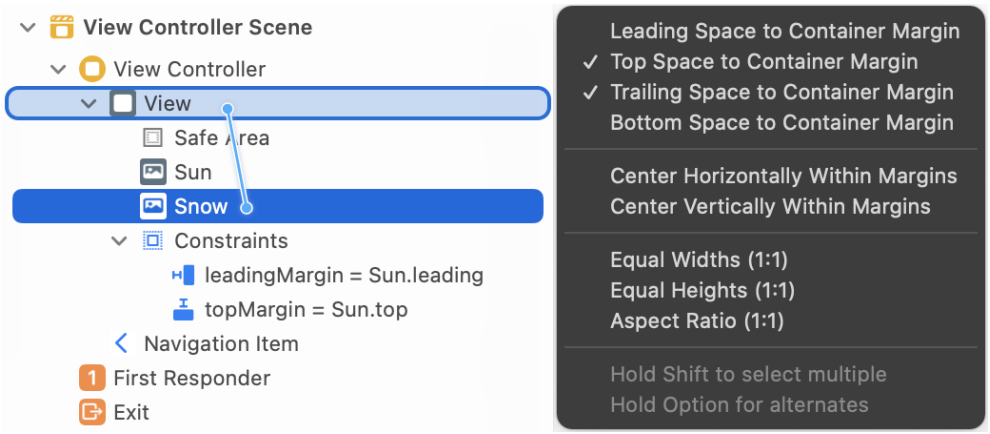


Change the background color of the sun image view to orange and the snowflake image view to blue.

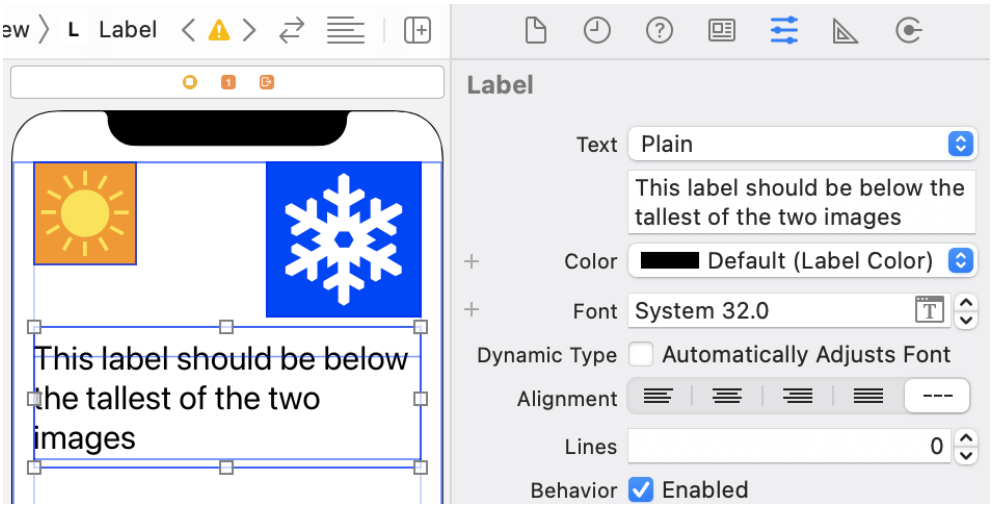
4. Fix the position of the sun image view by control-dragging from the image view to the root view in the document outline. Hold down the `Option` key and the `Shift` key to add constraints to the leading and top margins of the root view:



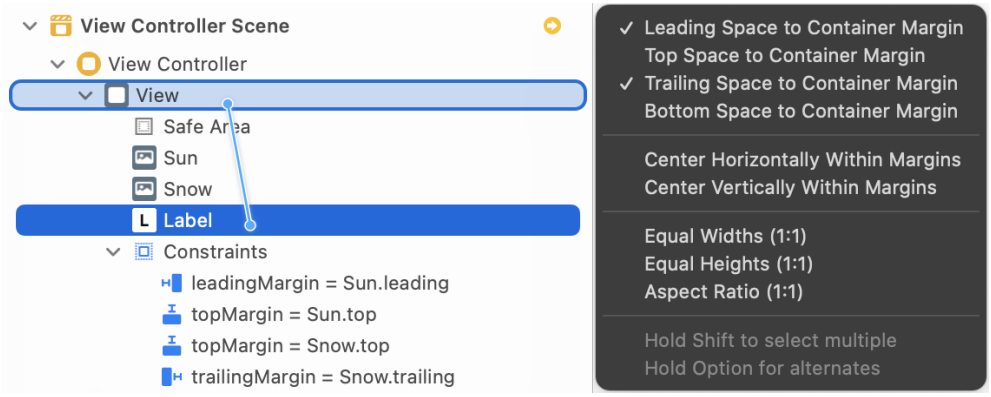
5. In the same way, pin the snowflake image view to the trailing and top margins of the root view:



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:



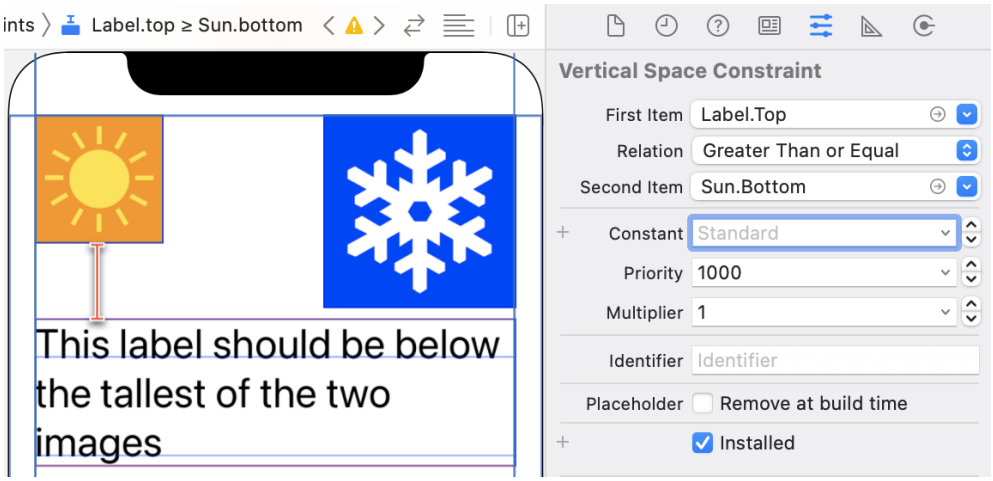
7. To fix the horizontal position control-drag from the label to the root view and create constraints to the leading and trailing margins.



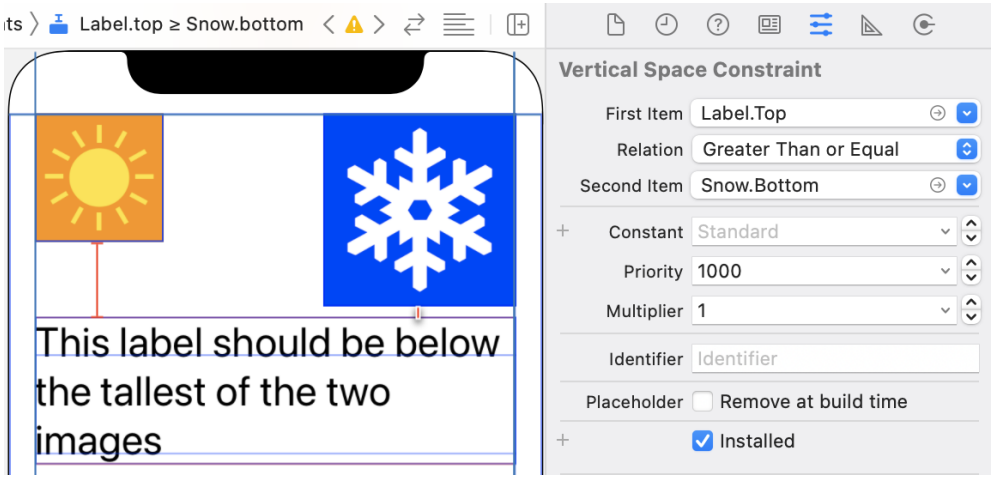
Note that this also fixes the width of the label.

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”:

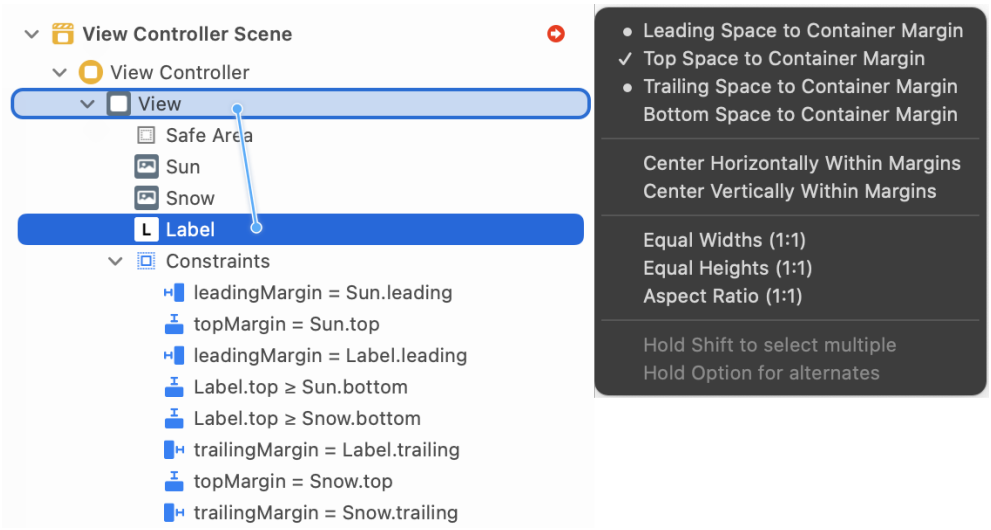


10. Create another “Greater Than Or Equal” inequality constraint from the label to the snowflake image:

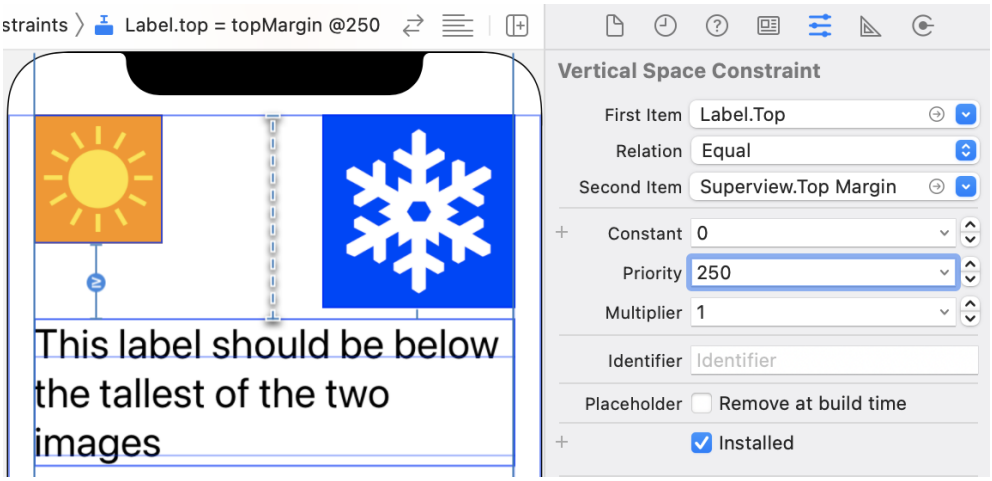


Both of these constraints are required constraints but because they are inequalities they don’t fix the vertical position of the label.

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 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 (>=) constraints that position the label at least a standard amount of spacing below the two images:

```
  captionLabel.topAnchor.constraintGreaterThan
  OrEqualToSystemSpacingBelow(sunView.bottomAnchor,
  multiplier: 1.0),
  captionLabel.topAnchor.constraintGreaterThan
  OrEqualToSystemSpacingBelow(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

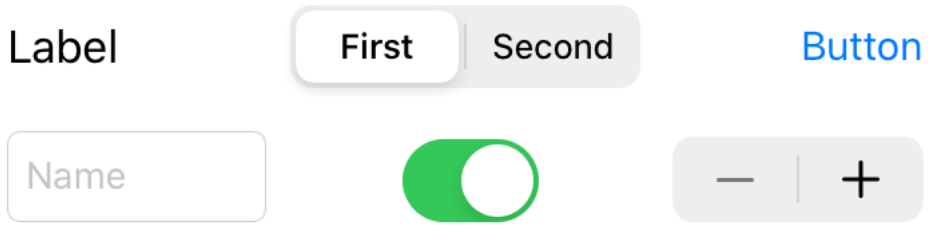
Some views have a natural size based on their content. A view wants to be this natural size unless you add constraints that stretch or squeeze it. **This natural size is also known as the intrinsic content size.**

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.

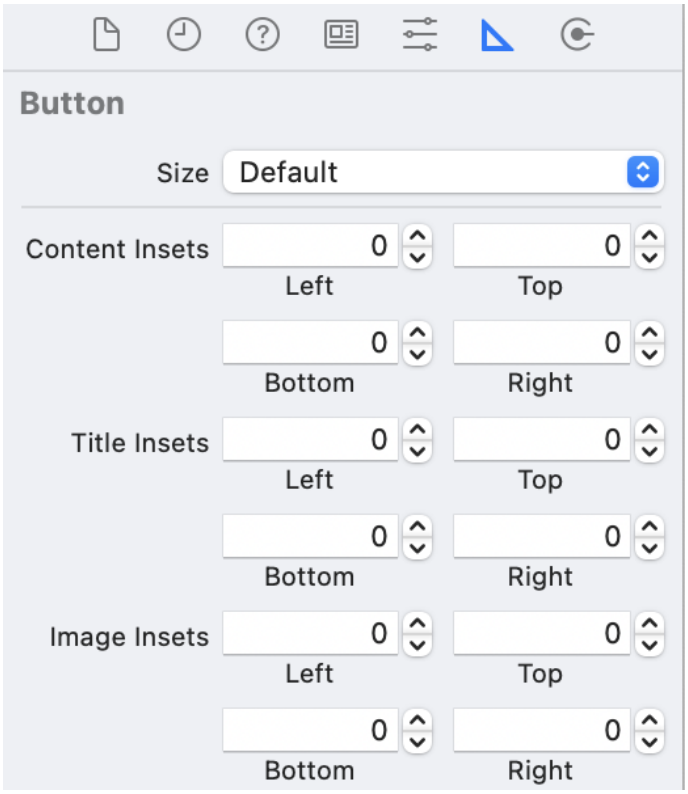
Standard UIKit Controls

Common UIKit controls like the button, label, switch, stepper, segmented control, and text field have both an intrinsic width and height:



UIButton

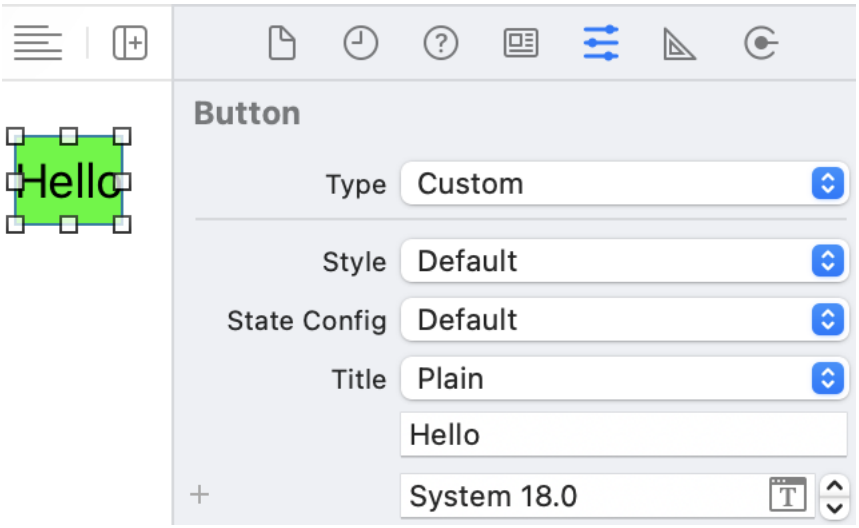
The text and the font used can change the intrinsic content size of a button. If you use insets to add padding or move the image or title be aware that only `contentEdgeInsets` has any effect on the intrinsic content size.



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.

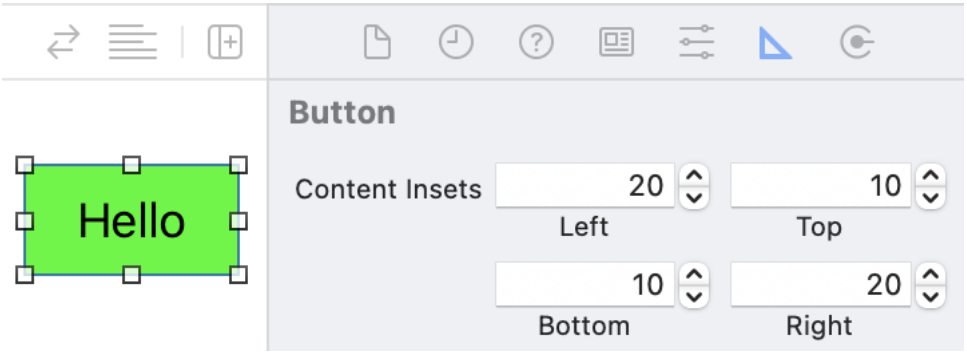
For example, suppose we have a custom button that's using the default 18 point system font:



If we create the button in code and check its intrinsic content size. It has a size of 41 x 34:

```
let button = UIButton(type: .custom)
button.setTitle("Hello", for: .normal)
button.backgroundColor = .green
button.intrinsicContentSize  // (w 41 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:



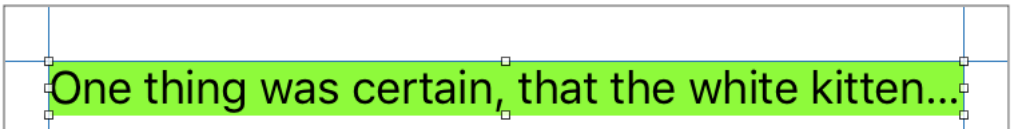

```
button.contentEdgeInsets = UIEdgeInsets(top: 10, left: 20,
  bottom: 10, right: 20)
button.intrinsicContentSize  // (w 81 h 42)
```

Changing layer properties like the corner radius doesn't change the intrinsic content size:

```
button.layer.cornerRadius = 10.0
button.intrinsicContentSize // (w 81 42)
```

UILabel

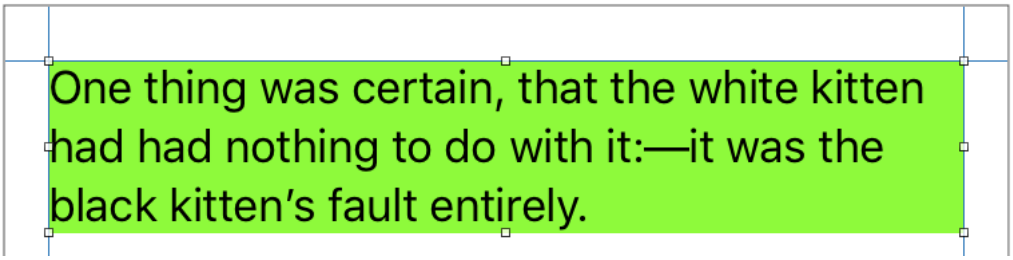
By default, a `UILabel` has an intrinsic content size that shows the text content on a single line. For long lines of text that can lead to truncation as here with a label constrained between the leading and trailing margins:



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 33)
```

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

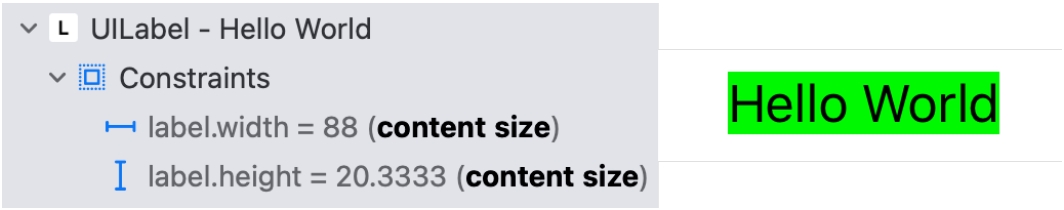
When you create a custom subclass of `UIView`, you can choose to override `intrinsicContentSize` and return a size based on the content of your custom view. If your view doesn't have a natural size for one dimension return `UIViewNoIntrinsicMetric` for that dimension:

```
// Custom view with an intrinsic height of 100 points
class CustomView: UIView {
  override var intrinsicContentSize: CGSize {
    return CGSize(width: UIViewNoIntrinsicMetric, height: 100)
  }
  // ...
}
```

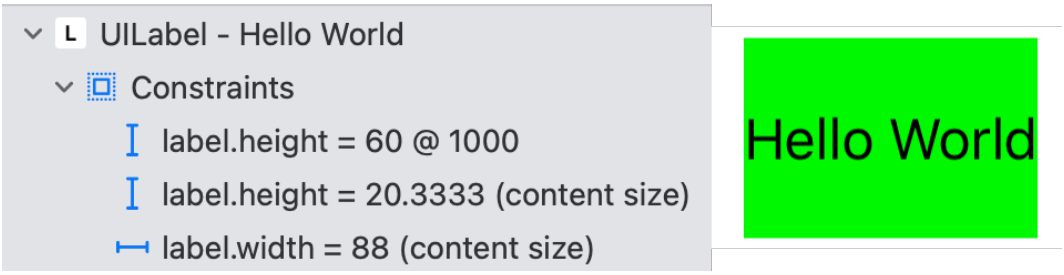
If the intrinsic content size of your custom view changes tell the layout engine by calling `invalidateIntrinsicContentSize()`.

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`, 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:

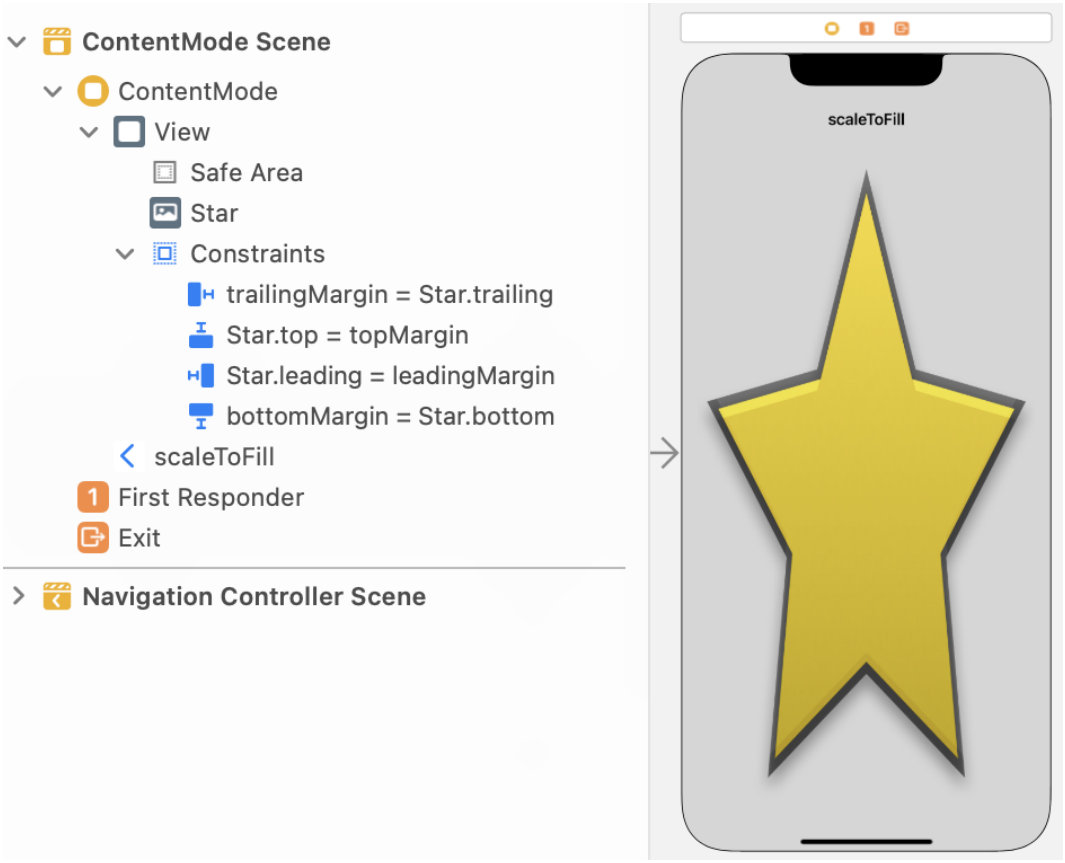


Our height constraint overrides the intrinsic height and stretches the label beyond its natural height.

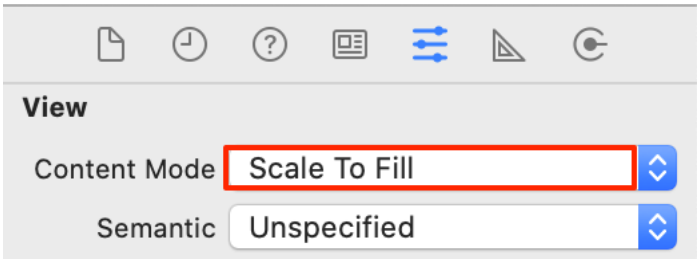
Content Mode

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:

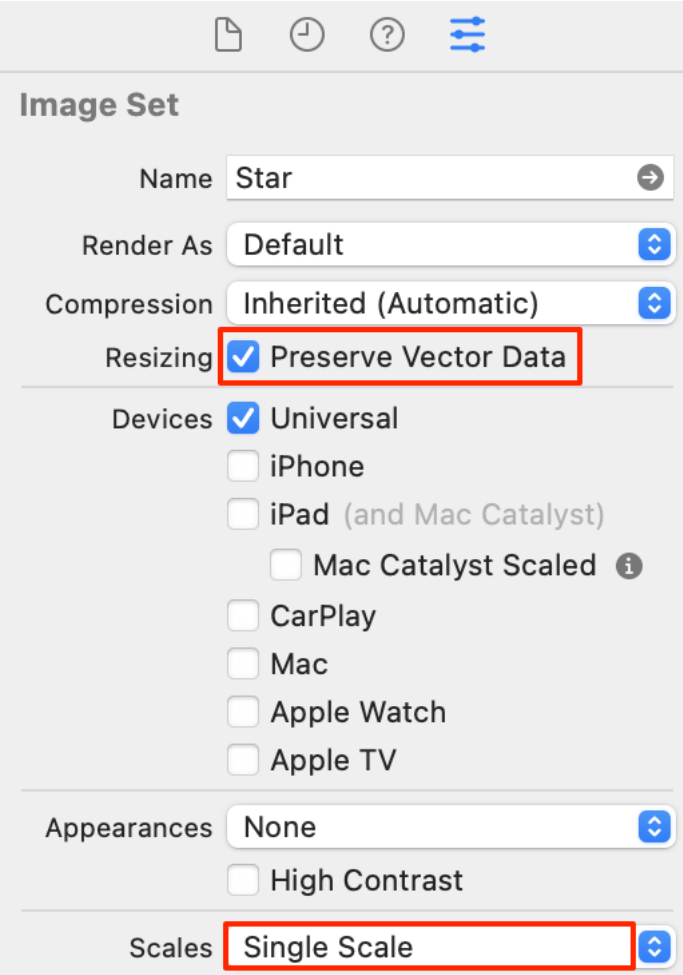
- Scaling the content (with or without maintaining the aspect ratio)
- Positioning the content
- Redrawing the content

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 select “Single Scale” and “Preserve Vector Data” in the asset catalog.



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`

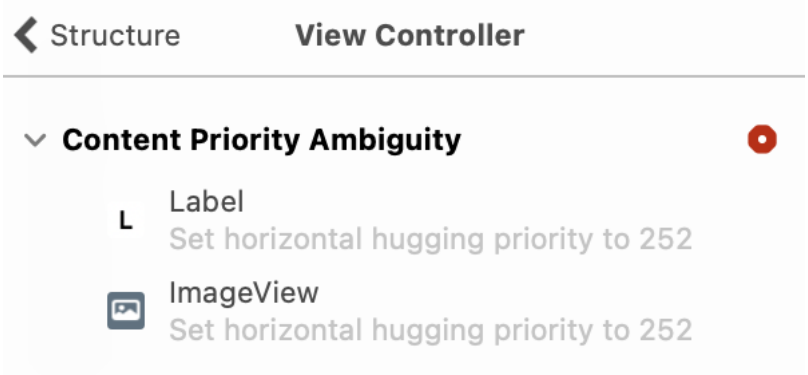
Note that the image view is still filling the space between the margins. The image view positions the image within its bounds based on the mode:

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.

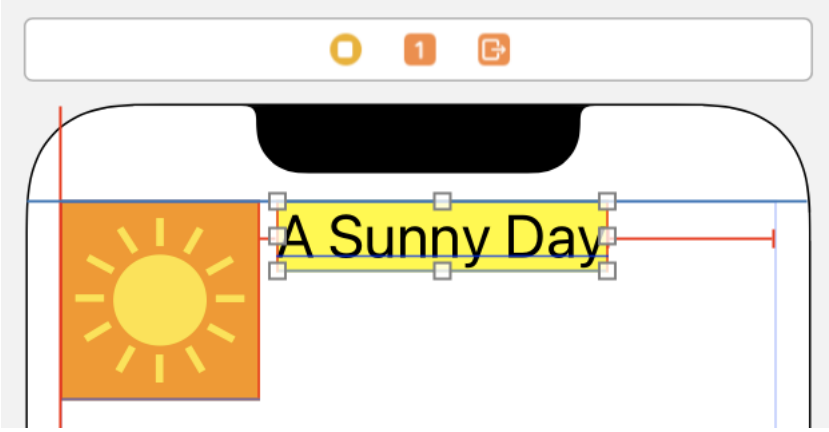
Changing the Content-Hugging or Compression-Resistance priority doesn’t affect views with no intrinsic content size.

A Practical Example Using Interface Builder

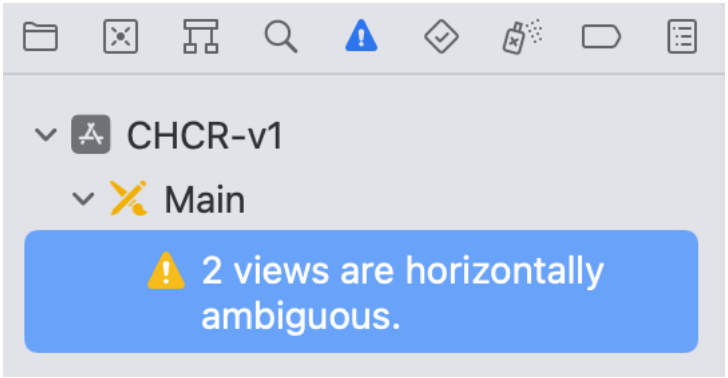
Let’s look at some situations where you need to change either the Content-Hugging or Compression-Resistance priorities (see sample code: CHCR-v1).

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.



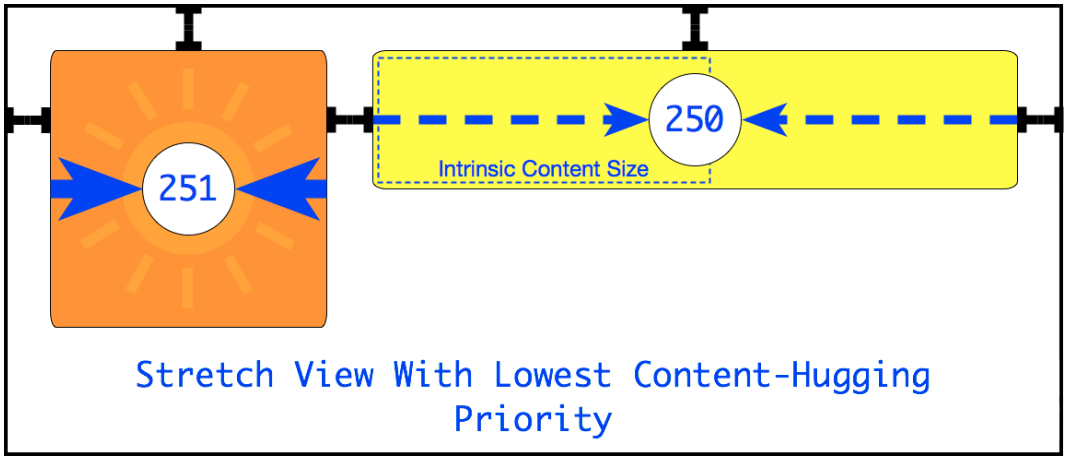
I've shown the views at their natural intrinsic content sizes. The widths of the two views are not enough to fill the space between the margins. Interface Builder warns you of an ambiguous layout:



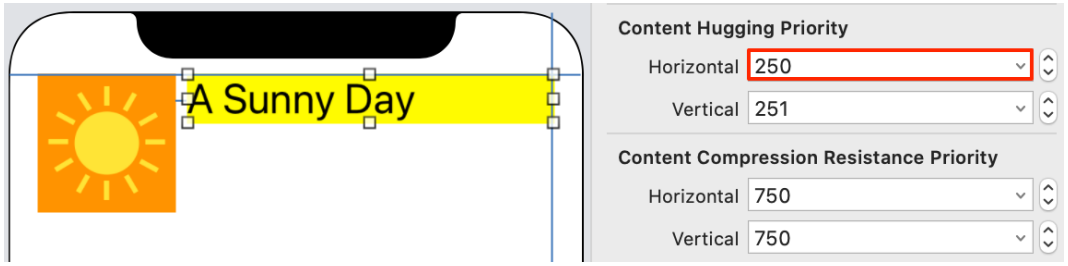
The layout engine wants to stretch one of the views to fill the space but which one? Well, that depends on the Content-Hugging priorities. If you create this layout with Interface Builder both the image view and the label have a default horizontal Content-Hugging priority of 251.

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.

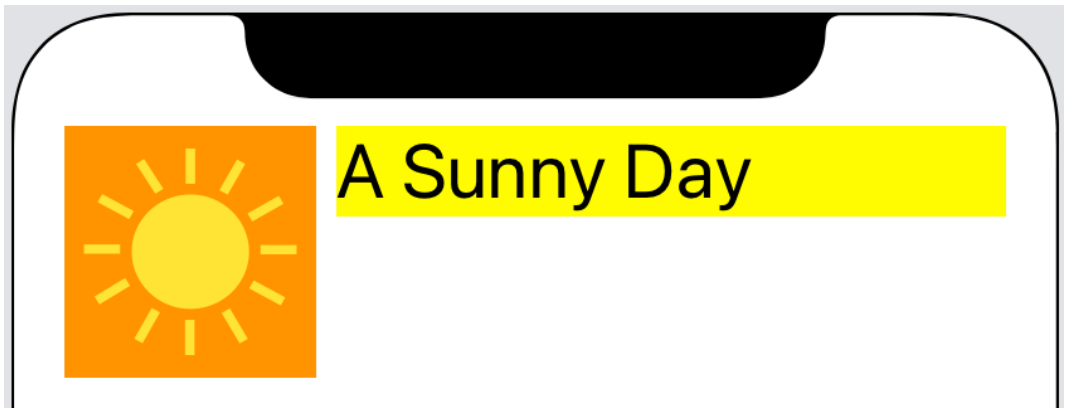
I want to stretch the text label, so I need to make sure it has the lowest Content-Hugging priority. For example, I can keep the image view priority at 251 and lower the label priority to 250:



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:

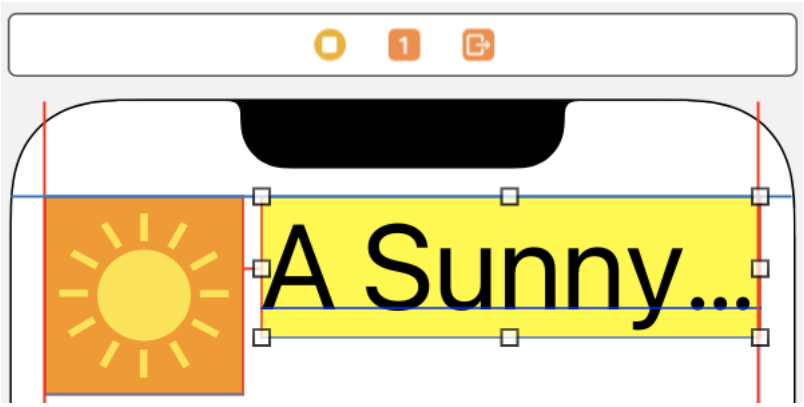


Once we remove the ambiguity the layout engine stretches the label to fill the available space keeping the image view at its natural size:

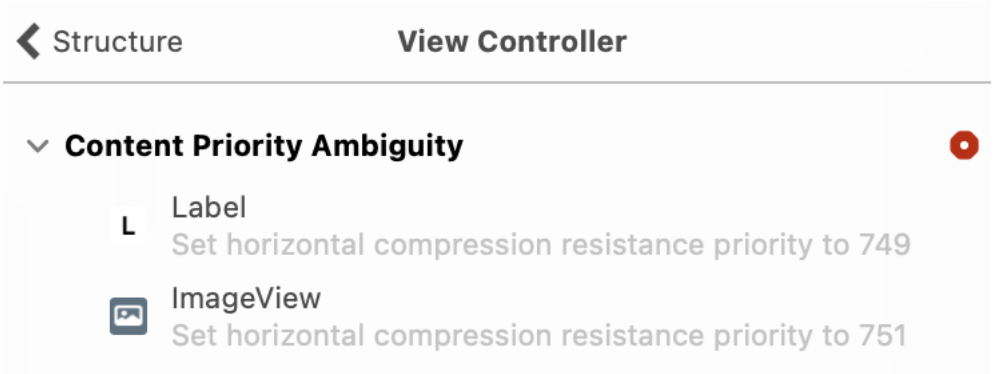


Squeezing A View

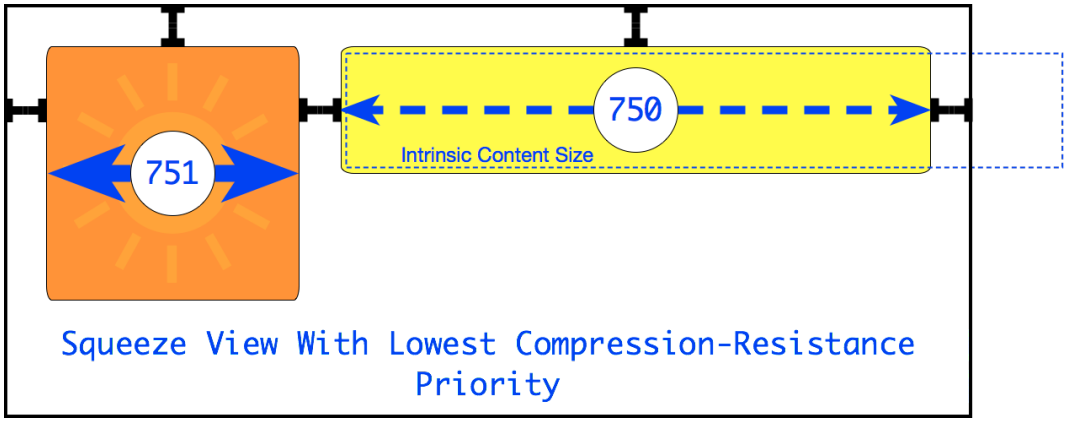
What would happen if our layout was too big to fit in the available space? This can happen easily enough when localizing or working with Dynamic Type. Here's what happens to our layout when I double the font size of the label from 30 points to 60 points:



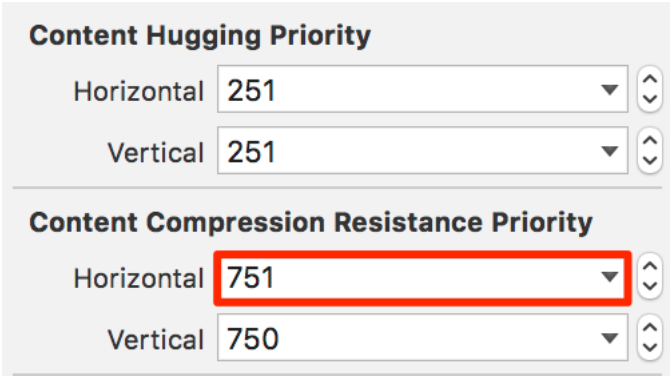
The text label has grown beyond the bounds of the superview. Our layout no longer fits within the margins and is once more ambiguous:



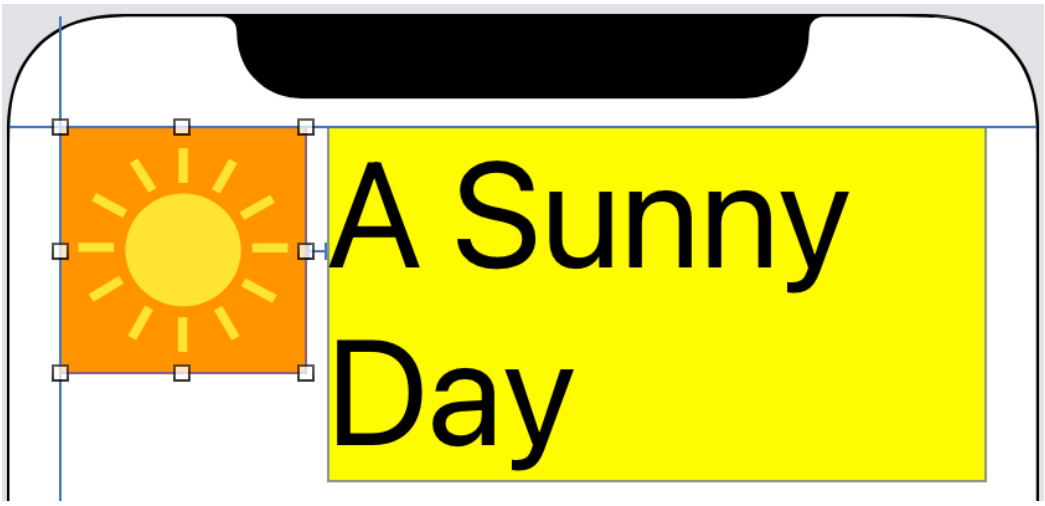
To satisfy the constraints the layout engine looks for the view with the lowest Compression-Resistance priority to squeeze. Unfortunately, both views have a horizontal Compression-Resistance value of `.defaultHigh` (750). Raising the priority for the image view to 751 fixes the problem:



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.

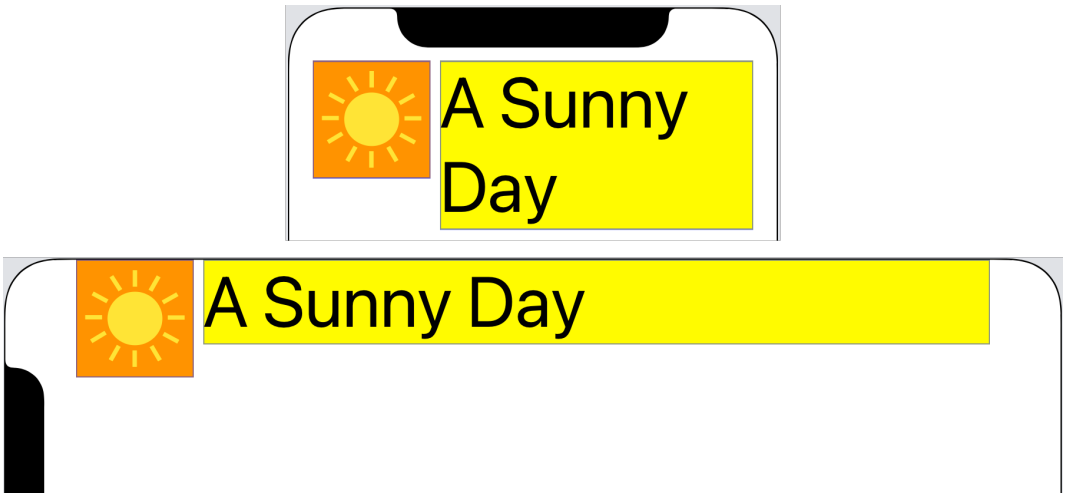


Summarizing the horizontal priorities we set for the two views:

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	250 (Code) 251 (IB)	750
UISwitch UIStepper UIDatePicker UIPageControl	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. Re-

member 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:

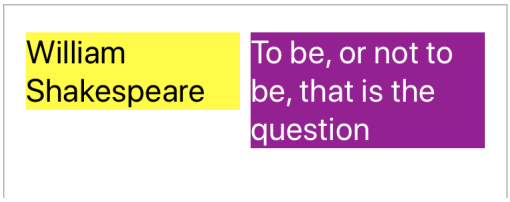
- Can my layout ever be too big to fit in the available space? If so decide which view to squeeze first and give it the lowest Compression-Resistance priority.
- Can my layout ever be too small to fit in the available space? If so decide which view to stretch first and give it the lowest Content-Hugging priority.

Test Your Knowledge

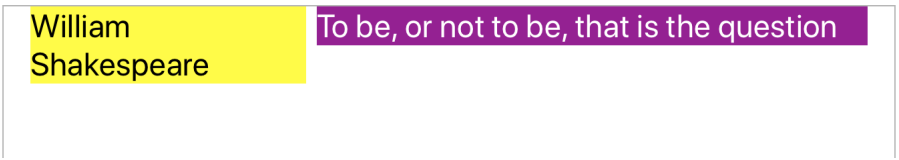
Practice using optional constraints when you need to pull a view as close as possible to a size or location. Learn to spot when you're working with views that have an intrinsic content size and need to change the content priority to stretch or squeeze a view.

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.

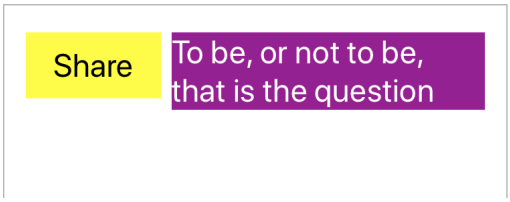
Hints And Tips

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.

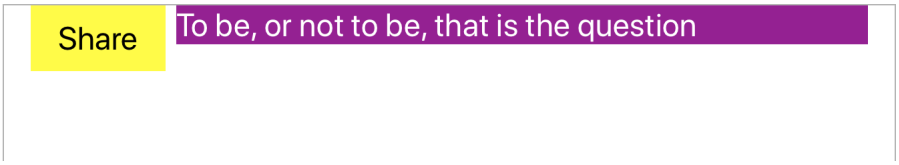
7. You don't need to change content hugging or compression resistance priorities.

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.

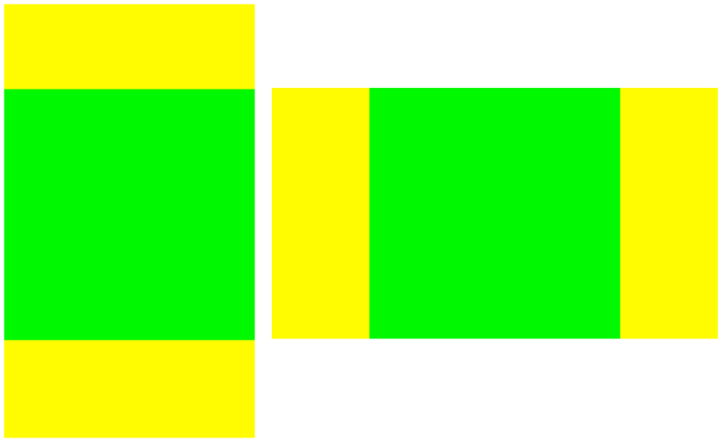
Hints And Tips

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.

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