Making Sense of

Auto Layout

Build user interface that looks great across all iOS devices

Axel Kee

Table of Contents

0 - Introduction

I used to have headache around constraints and fixing red lines feels like duct taping a leaking water pipe, new leak appears as soon as you finished applying duct tape on the previous leak. I realized trial-and-error my way out to fix constraint doesn't work hence I turned to read Apple documentation on the fundamental of constraint / Auto Layout to try to gain some understanding of it. If you are struggling with Auto Layout, you are not alone, I have been there, and so have many others.

In this book, you will learn the fundamentals of Auto Layout, how constraints work and why Apple designed it to work that way.

This book will use Interface Builder to create constraint throughtout the chapters, it's fine if you prefer to use code to create constraint, the concept of constraint / auto layout remains same on both Interface Builder and code.

There's some exercise projects that you can use to follow along in some chapters, the projects are located in the `exercises` folder. There's also some videos that will be used in some chapters, they are located in the `videos` folder.

Remember to set `view.translatesAutoResizingMaskIntoConstraints = false` if you are creating view that uses constraint programmatically. I can't count how many times I've stumbled into weird layout problems just because I forgot to set that property to false.

As you read this book, you will learn new concepts that might be difficult to grasp at first. If you don't understand something, it is not your fault , just send your question to axel@fluffy.es and I will try my best to clear things up.

I hope this book can make Auto Layout 'clicks' for you.

Axel Kee

axel@fluffy.es

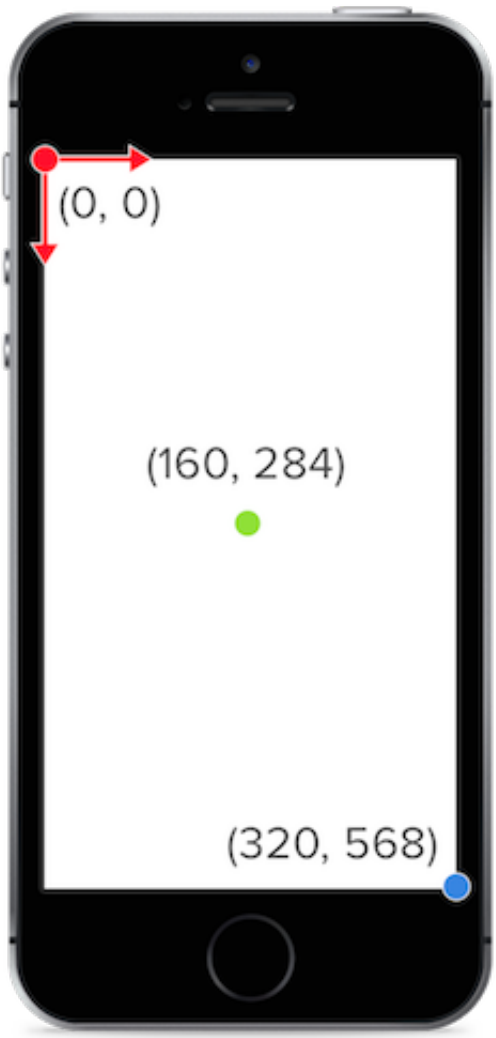
1 - Why Auto Layout is introduced by Apple

Before diving into Auto Layout, let's look at how user interface (UI) is being implemented on iOS prior to Auto Layout.

In the early days of iOS development, developing UI for iOS app is pretty straightforward as there is only one screen size. The screen point size of the original first iPhone up to iPhone 4s is 320 x 480. (iPhone 4s introduced Retina display with 4x more pixels but its screen point size is still remains 320 x 480)

Frame based layout with coordinate system

During this era, frame-based layout is used and position of UI elements are usually hardcoded as the screen size is fixed. Frame based layout uses a **coordinate system** like this :

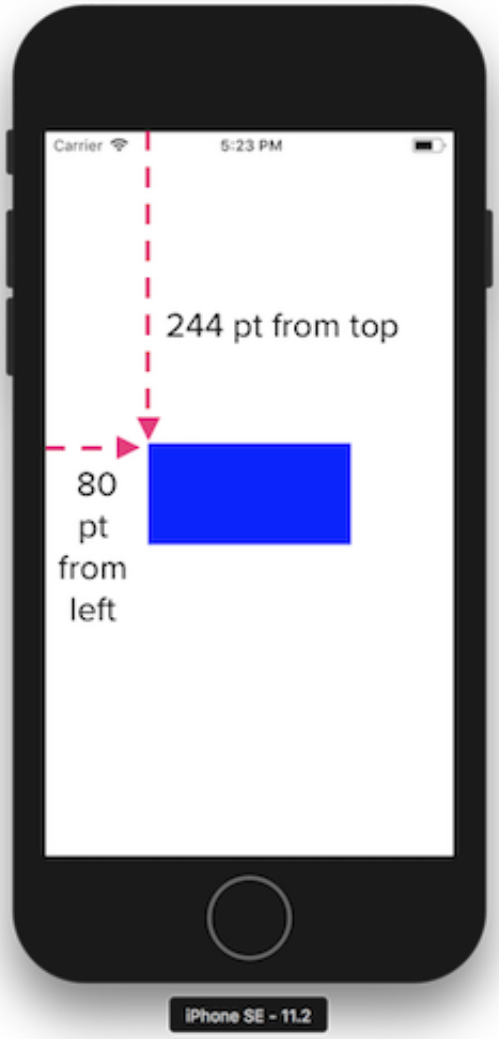


Coordinate system in iOS starts from top left and ends at bottom right. The top left coordinate is (0, 0) , which mean the horizontal (x) position is 0 and vertical (y) position is 0. For this example we used iPhone SE which have a point resolution of 320pt x 568 pt, so the bottom right coordinate is (320, 568), x position is 320 and y position is 568. The center point is (320 / 2 , 568 / 2) = (160, 284).

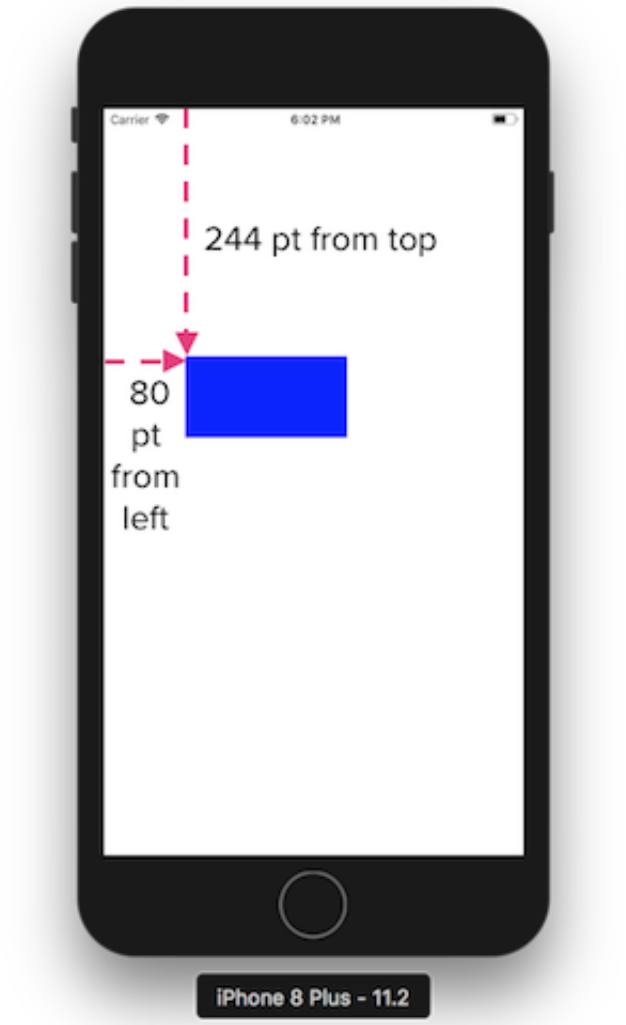
In the frame based coordinate system, we can create an UIView in code like this:

```
let blueView = UIView(frame: CGRect(x: 80.0, y: 244.0, width: 160.0, height: 80.0))
blueView.backgroundColor = UIColor.blue
self.view.addSubview(blueView)
```

This will put a blue rectangle in X position of 80 (from left), Y position of 244 (from top) and a width of 160 and height of 80. Like this :



I have precalculated the position so that the rectangle is placed in the center of the view (horizontally and vertically). It is centered nicely in iPhone SE, but when you view it on iPhone 8 Plus, it is positioned upper left from the center because the iPhone 8 Plus screen width and height is larger than iPhone SE, like this :



To accommodate for iPhone 8 plus screen, we will have to adjust the x, y position of the view to get it to the center of the screen.

```
if(screenSize == iPhoneSE){
    blueView.frame = CGRect(x: 80.0, y: 244.0, width: 160.0, height: 80.0)
} else if (screenSize == iPhone8Plus){
    blueView.frame = CGRect(x: 127.0, y: 338.0, width: 160.0, height: 80.0)
}
```

As more device with different screen size is introduced by Apple, it become troublesome to cater for each of the screen size :

```
// Don't do this! This serve as a bad example

if(screenSize == iPhoneSE){
    blueView.frame = CGRect(x: 80.0, y: 244.0, width: 160.0, height: 80.0)
} else if (screenSize == iPhone8){
    blueView.frame = CGRect(x: 107.5, y: 303.5, width: 160.0, height: 80.0)
} else if (screenSize == iPhone8Plus){
    blueView.frame = CGRect(x: 127.0, y: 338.0, width: 160.0, height: 80.0)
} // Apple suddenly launch iPhone X
else if (screenSize == iPhoneX){
// ffuuuu...
}

// what if user rotate the screen?!
// what if there's a top blue bar when Personal Hotspot is turned on?
```

To solve this problem, Apple has introduced Auto Layout with constraint based approach so you wouldn't need to manually accommodate each different screen size. **Auto Layout will do the tedious calculation of frames** for you, all you need to do is define constraints for each UI elements (UIView, UILabel, UIButton etc..) . Auto Layout will then use these constraints to calculate the position and size for each of the UI elements.

Auto Layout using constraint

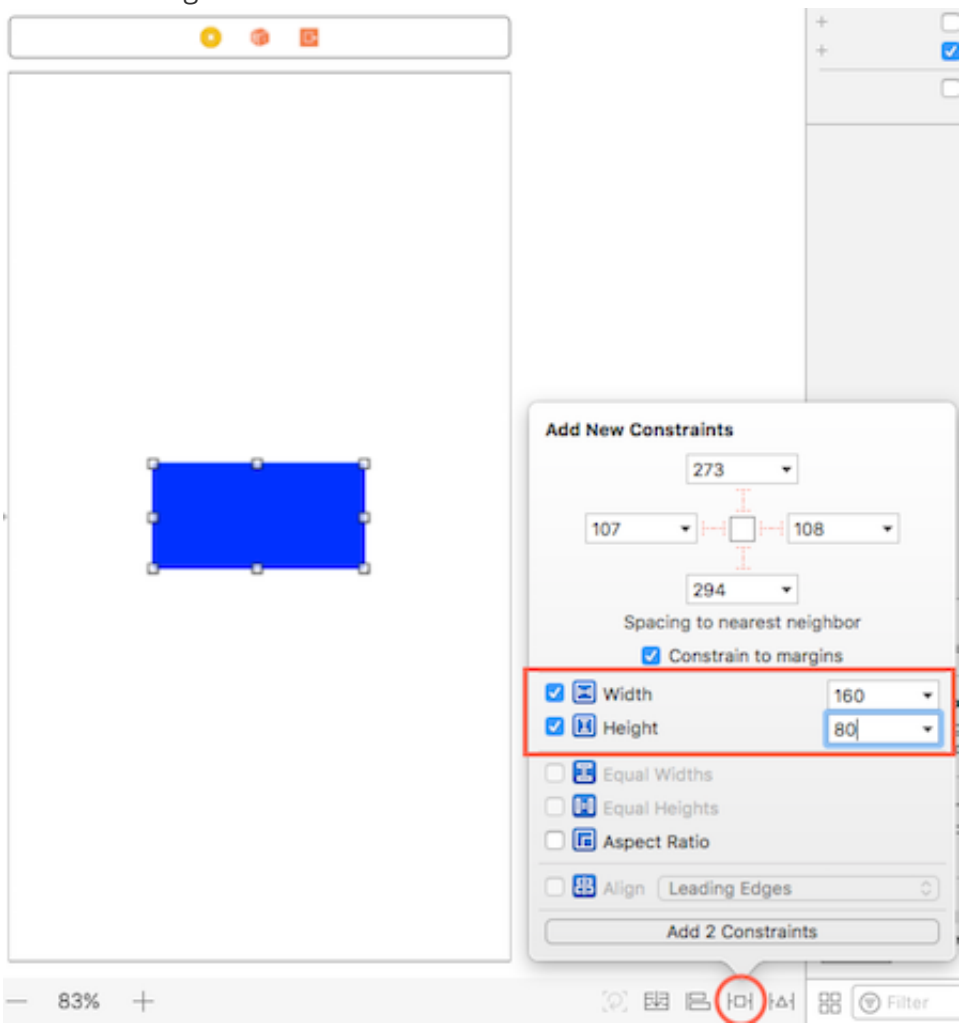
Things start to get complicated after iPhone 5 is released, then iPhone 6 and 6 plus... and then iPhone X , it is certainly possible to manually design multiple UI layout to fit each of them, but it is very time consuming and repetitive.

Auto Layout is a feature which the size and position of all views in the view hierarchy are dynamically calculated on run time, based on constraints placed on those views. "Dynamically calculated on run time" as in the size and position will be calculated every time you run the app, and recalculated again as you rotate the phone orientation or using split view in iPads etc.

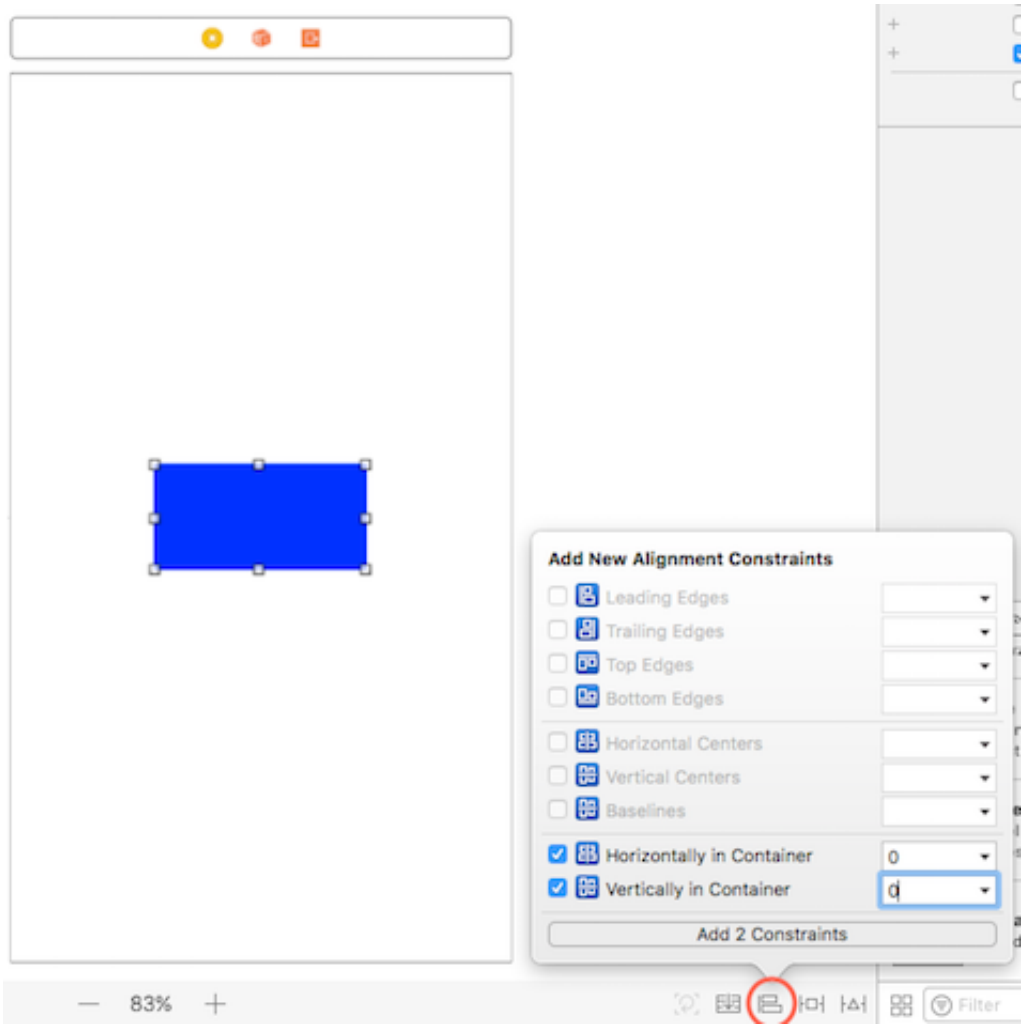
For easier understanding, let's call the thing that will dynamically calculates the size and position of all the views as **Auto Layout Engine**.

Continuing the previous example shown in frame based layout section, to position the blue rectangle to center horizontally and vertically using Auto Layout, we add constraints to the blue view in Interface Builder / Storyboard like this :

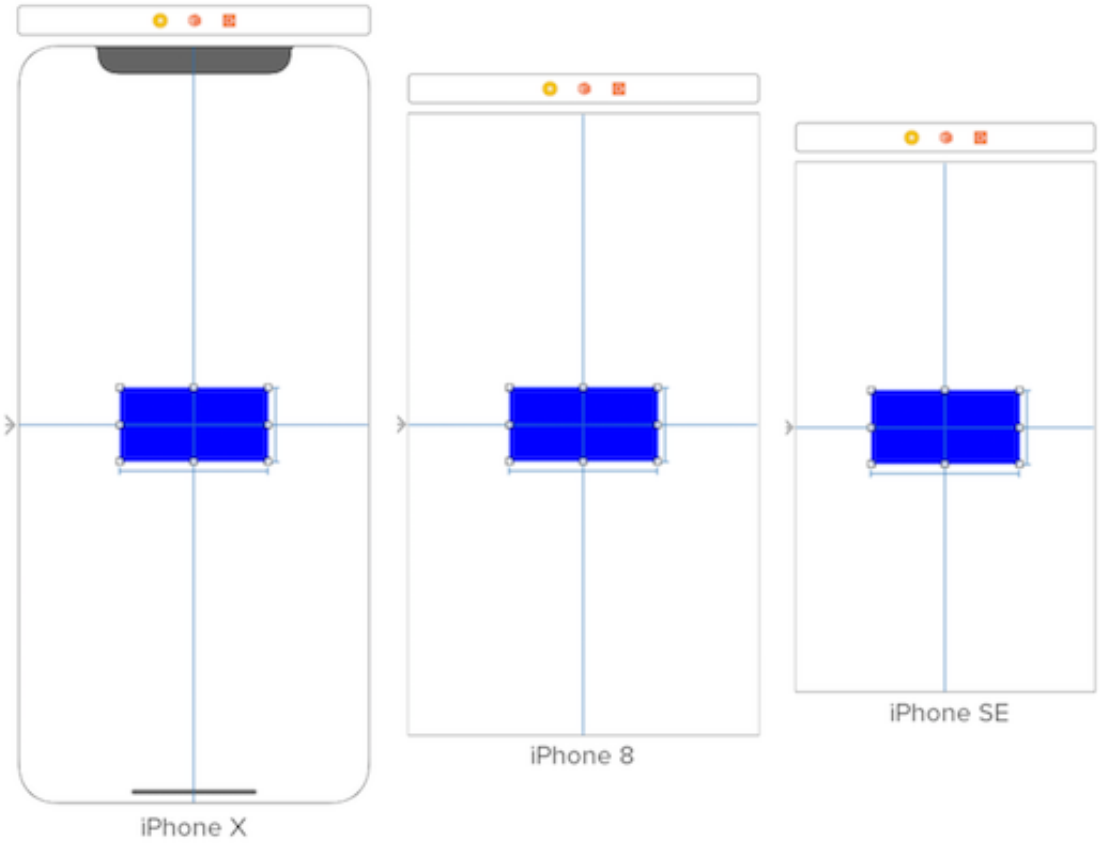
Width and height constraints:



Horizontal and vertical constraints:



This will set the blue view size (width and height) and also center it both horizontally and vertically to the screen. The constraints will work across multiple devices seamlessly like this :



Brief overview of Auto Layout calculation

Following the constraints set previously, when your app is launched in iPhone X, Auto Layout Engine know that the screen point size of iPhone X is **375** x **812**, then it will calculate the screen center position like this:

```
screenCenterX = (screen width) / 2 = 375 / 2 = 187.5
screenCenterY = (screen height) / 2 = 812 / 2 = 406
```

To position the blue view nicely in the center, Auto Layout Engine will use the width and height of the blue view to calculate what position it should place the blue view on. The calculation formula might look like this :

```
// iPhone X

screenCenterX = 187.5
screenCenterY = 406

width of blue view = 160
```

```
height of blue view = 80

// x position of blue view
// if blue view is horizontally centered, left half of the blue view should be on the left of the screenCenterX
x = screenCenterX - ( (width of blue view)/ 2 )
x = 187.5 - (160 / 2)
x = 107.5

// y position of blue view
// if blue view is vertically centered, top half of the blue view should be on top of the screenCenterY
y = screenCenterY - ( (height of blue view) / 2 )
y = 406 - (80 / 2)
y = 366
```

Now when you launch your app in iPhone 8, Auto Layout Engine will use the point resolution of iPhone 8 (**375** x **667**) and calculate the screen center points and position of the blue view again using the same formula.

```
// iPhone 8

screenCenterX = 187.5
screenCenterY = 333.5

width of blue view = 160
height of blue view = 80

// x position of blue view
// if blue view is horizontally centered, left half of the blue view should be on the left of the screenCenterX
x = screenCenterX - ( (width of blue view)/ 2 )
x = 187.5 - (160 / 2)
x = 107.5

// y position of blue view
// if blue view is vertically centered, top half of the blue view should be on top of the screenCenterY
y = screenCenterY - ( (height of blue view) / 2 )
y = 333.5 - (80 / 2)
y = 293.5
```

Since Auto Layout Engine always calculate the size and position of the view before rendering it on screen, you can rest assured that the layout will work nicely if you define all the constraint correctly.

Other than different phone size, Auto Layout will also recalculate position and size of a view when the followings happen :

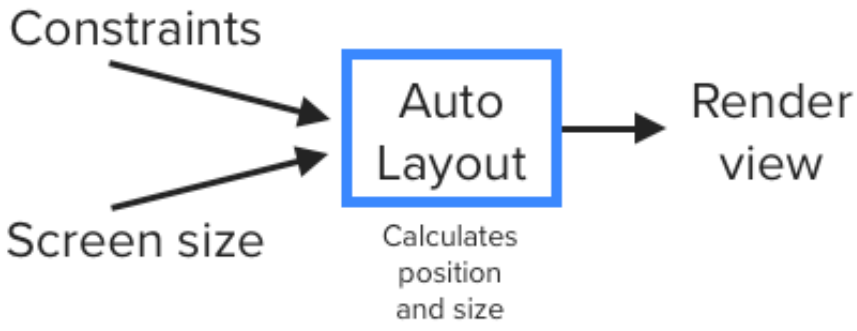
1. Device is rotated
2. Split screen view is enabled in iPad
3. The active call red top bar appears on iPhone

Imagine you have to manually write code to handle all the situation listed above, it would be a hell lot of work. Auto Layout saves us from having to deal with these.

2 - How Auto Layout calculates position and size using constraints

To recap, Auto Layout takes input of **constraints** (the constraint you created in Xcode) and **app screen size** (depending on iOS device screen resolution, orientation) and calculates **position** and **size** of each UI Elements (eg: UILabel, UIButton, UIView, etc).

Here is the official screen resolution of each iOS devices, for UI development we will use the **UIKit Size (Points)** column. **UIKit Size** is the size we get when doing user interface in Xcode .



In order for Auto Layout to render the position and size of a view correctly, Auto Layout **must know** or **able to calculate** the :

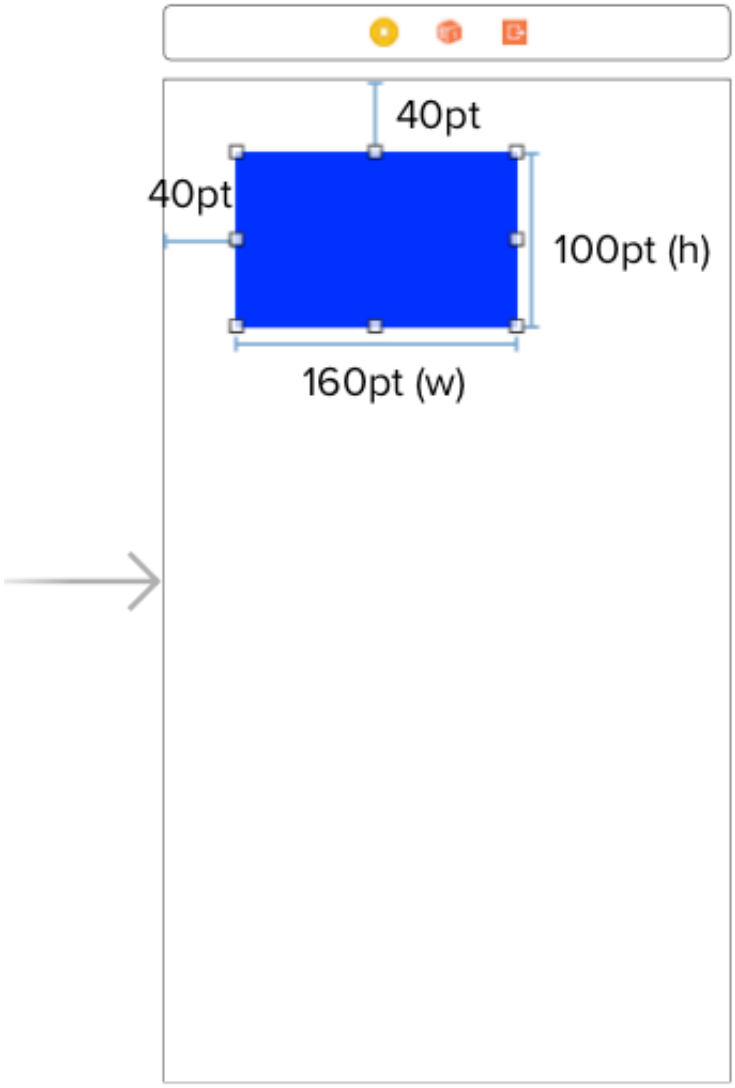
1. **x position** of a view
2. **y position** of a view
3. **width** of a view
4. **height** of a view

Defining constraint helps Auto Layout to figure all of the above. Below there will be some examples showing how Auto Layout calculates the position and size of a view using simple math.

We will discuss 4 examples in this chapter, there will be 1 exercise question at the end of this chapter.

Example 1

A straight forward example might look like this :



4 constraints are defined :

1. Distance from the screen left to the left of the blue view is 40 pt
2. Distance from the screen top to the top of the blue view is 40 pt
3. Width of the blue view is 160 pt
4. Height of the blue view is 100 pt

From the 1st constraint, Auto Layout can deduce that the x position of the view is 40.

From the 2nd constraint, Auto Layout can deduce that the y position of the view is 40.

From the 3rd constraint, Auto Layout can deduce that the width of the view is 160.

From the 4th constraint, Auto Layout can deduce that the height of the view is 100.

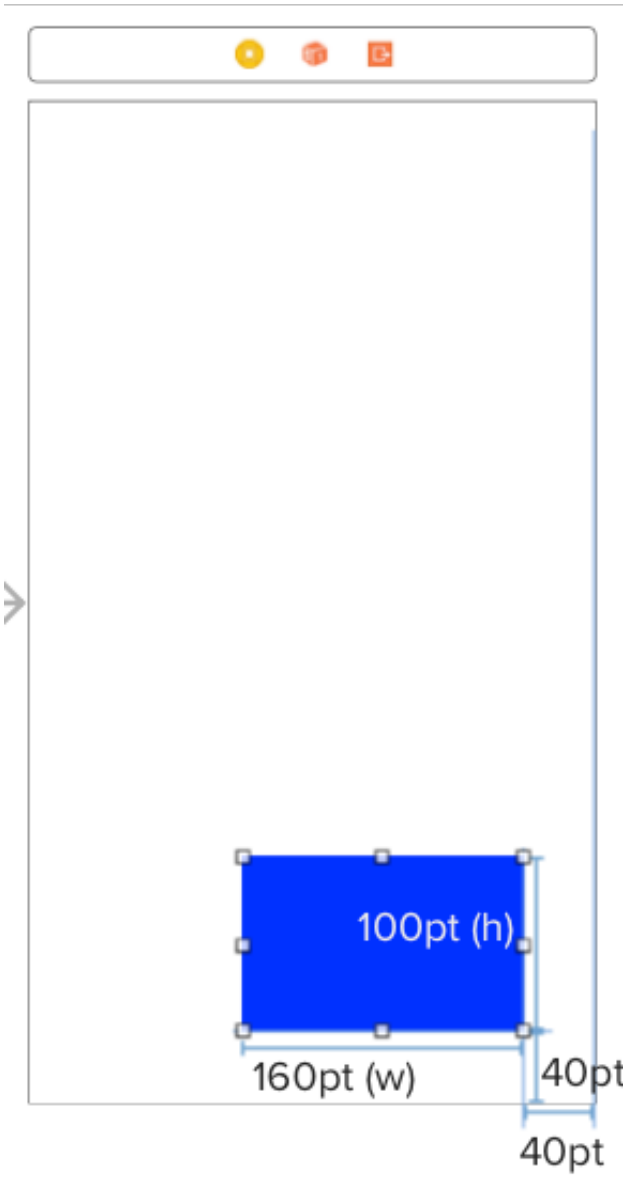
Since x, y, width and height of the blue view is known, Auto Layout will then render it nicely across multiple screen size and orientation :

iPhone SE | Portrait

iPhone SE | Landscape

Example 2

Similar to previous example, we use back the same width and height but this time we define a right and bottom constraint like this :



4 constraints are defined :

1. Distance from the screen right to the right of the blue view is 40 pt
2. Distance from the screen bottom to the bottom of the blue view is 40 pt
3. Width of the blue view is 160 pt
4. Height of the blue view is 100 pt

To calculate **x position** of the blue view, Auto Layout will use the screen width. Calculation on an iPhone SE (screen width 320 pt) will look like this :

```
// iPhone SE
screenWidth = 320

// distance of blue view from right of the screen
distanceFromScreenRight = 40

// width of blue view
viewWidth = 160

// x position of the blue view
x = screenWidth - distanceFromScreenRight - viewWidth
x = 320 - 40 - 160
x = 120
```

To calculate **y position** of the blue view, Auto Layout will use the screen height. Calculation on an iPhone SE (screen height 568 pt) will look like this :

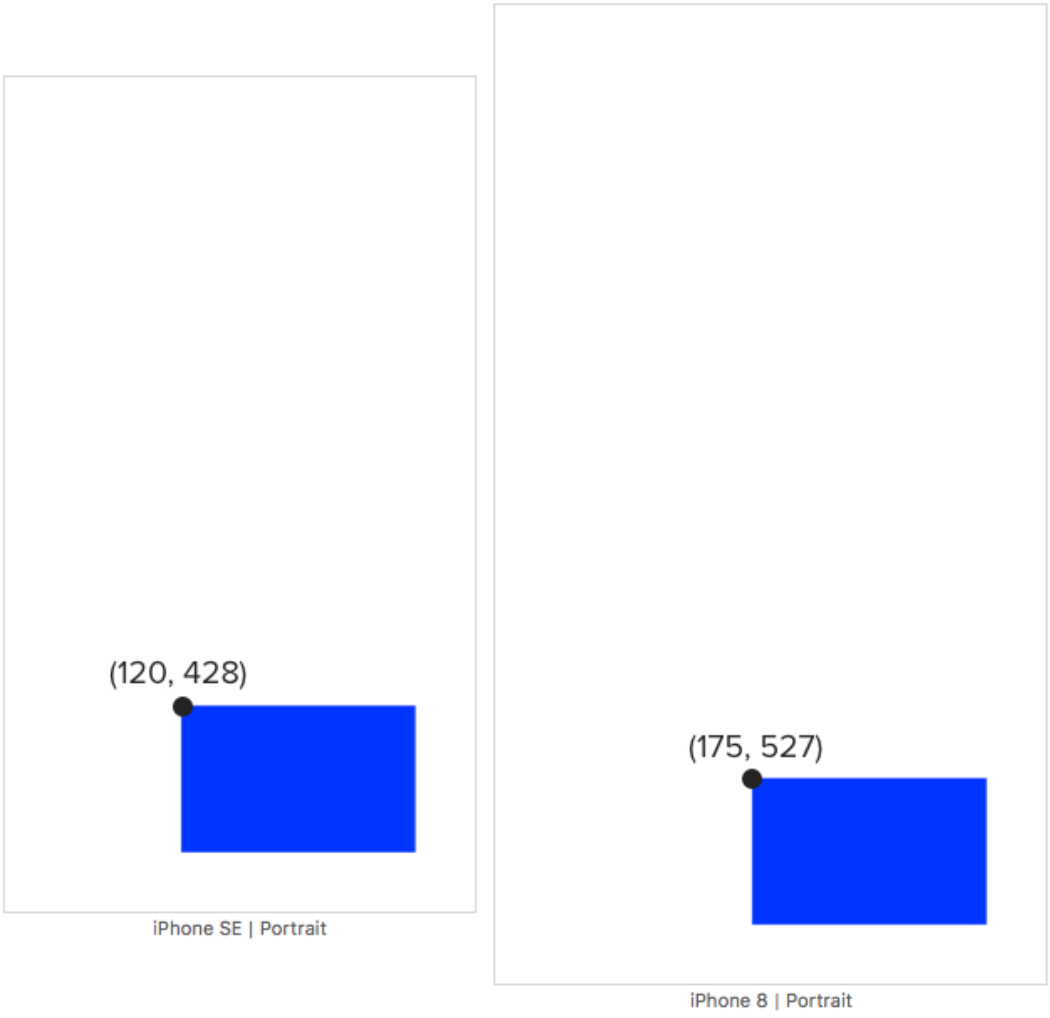
```
// iPhone SE
screenHeight = 568

// distance of blue view from bottom of the screen
distanceFromBottom = 40

// height of blue view
viewHeight = 100

// y position of the blue view
y = screenHeight - distanceFromBottom - viewHeight
y = 568 - 40 - 100
y = 428
```

Using the same calculation formula, iPhone 8 (screen width 375, height 667) will calculate the x position as 175, y position as 527.

```
// iPhone 8
screenWidth = 375
screenHeight = 667

// distance of blue view from right of the screen
distanceFromScreenRight = 40

// distance of blue view from bottom of the screen
distanceFromBottom = 40

// width of blue view
```

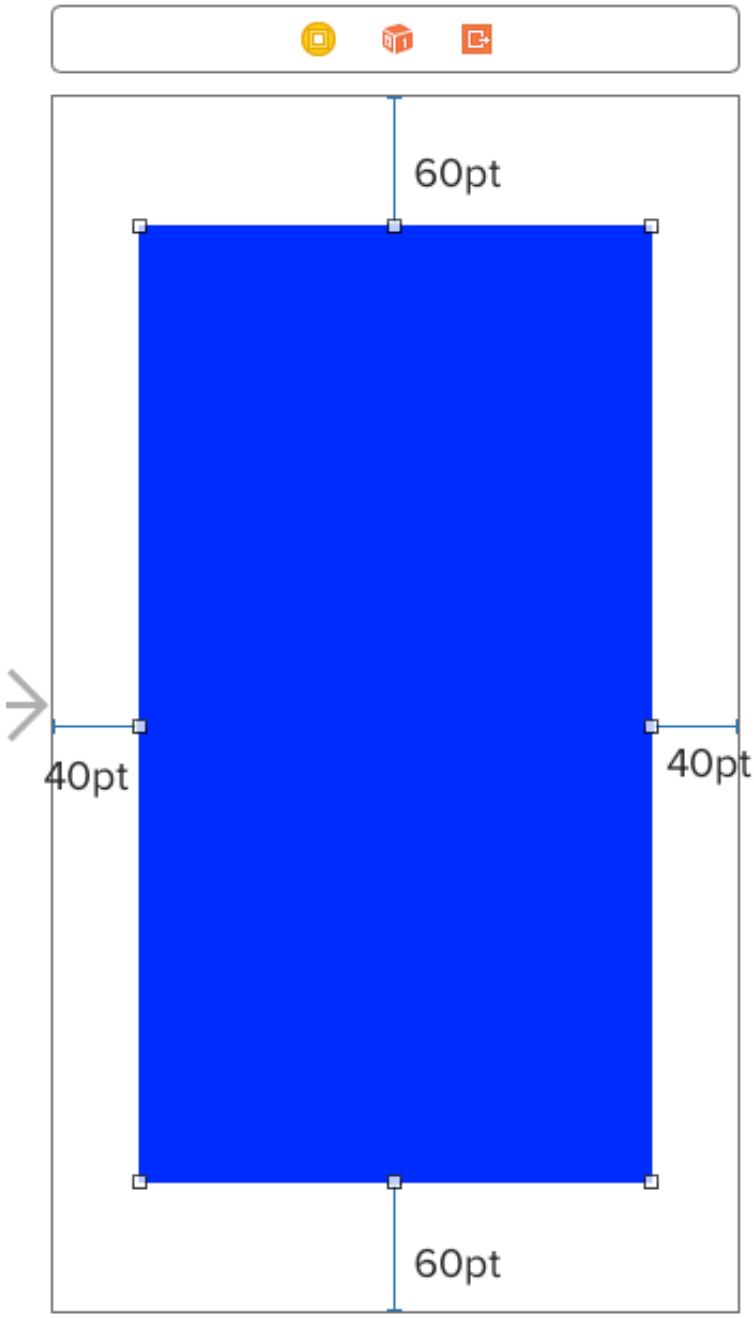
```
viewWidth = 160
// height of blue view
viewHeight = 100

// x position of the blue view
x = screenWidth - distanceFromScreenRight - viewWidth
x = 375 - 40 - 160
x = 175

// y position of the blue view
y = screenHeight - distanceFromBottom - viewHeight
y = 667 - 40 - 100
y = 527
```

Now x, y, width and height of the view is known, Auto Layout will render it nicely across multiple screen size and orientation :



Example 3

In this example, there is no width or height constraint defined, instead we will set the top, left, right and bottom constraint from the blue view to the screen :



4 constraints are defined :

1. Distance from the screen top to the top of the blue view is 60 pt
2. Distance from the screen left to the left of the blue view is 40 pt
3. Distance from the screen right to the right of the blue view is 40 pt
4. Distance from the screen bottom to the bottom of the blue view is 60 pt

From the 1st constraint, Auto Layout can deduce that the y position of the view is 60.

From the 2nd constraint, Auto Layout can deduce that the x position of the view is 40.

To calculate **width** of the blue view, Auto Layout will use the screen width.

Calculation on an iPhone SE (screen width 320 pt) will look like this :

```
// iPhone SE
screenWidth = 320

// distance of blue view from the left of screen
distanceFromLeft = 40

// distance of blue view from the right of screen
distanceFromRight = 40

// width of the blue view
viewWidth = screenWidth - distanceFromLeft - distanceFromRight
viewWidth = 320 - 40 - 40
viewWidth = 240
```

To calculate **height** of the blue view, Auto Layout will use the screen height. Calculation on an iPhone SE (screen height 568 pt) will look like this :

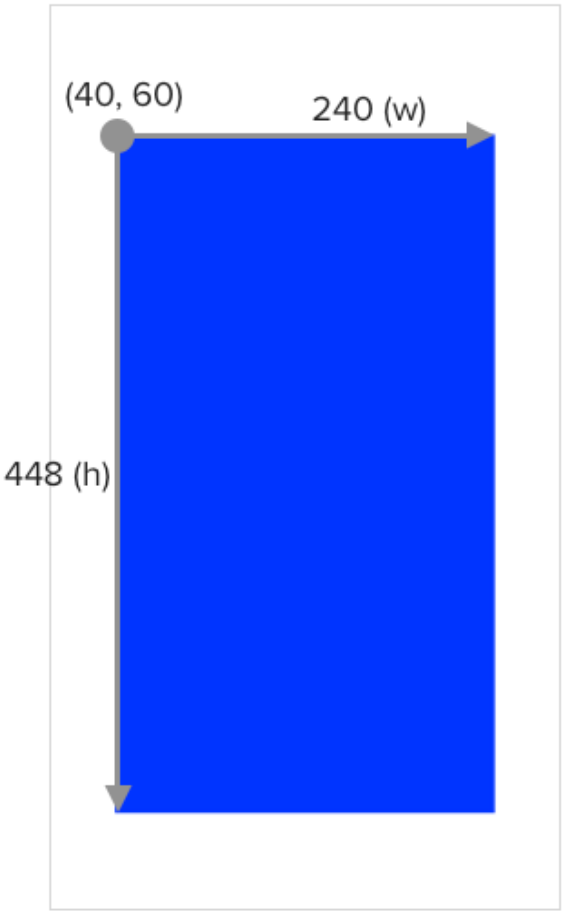
```
// iPhone SE
screenHeight = 568

// distance of blue view from the top of screen
distanceFromTop = 60

// distance of blue view from the bottom of screen
distanceFromBottom = 60

// height of the blue view
viewHeight = screenHeight - distanceFromTop - distanceFromBottom
viewHeight = 568 - 60 - 60
viewHeight = 448
```

Using the same calculation formula, iPhone 8 (screen width 375, height 667) will calculate the width as 295, height as 547.

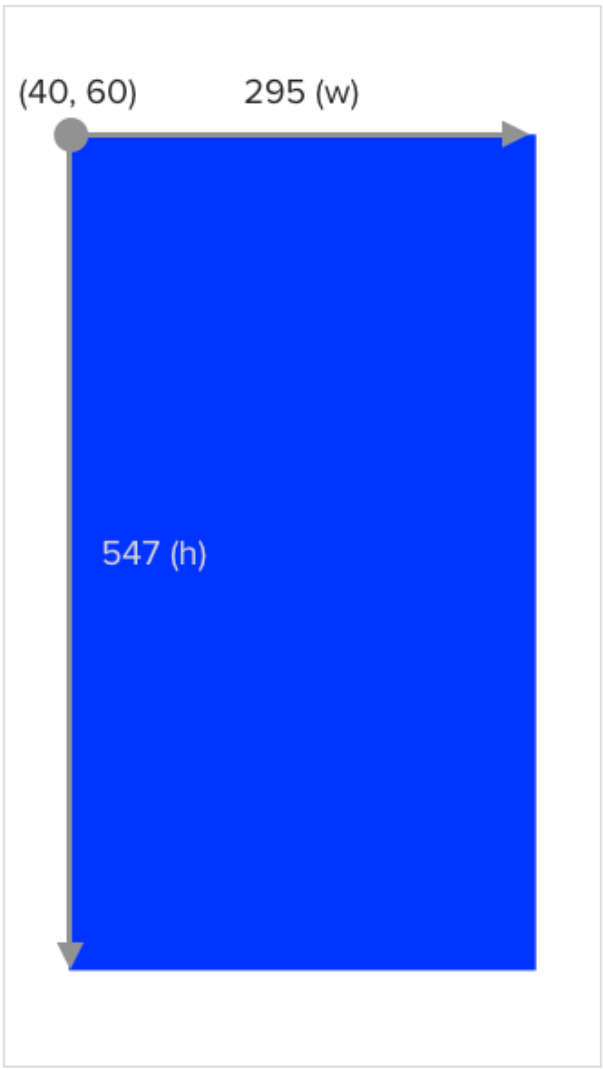
```
// iPhone 8
screenWidth = 375
screenHeight = 667
```

```
// distance of blue view from the left of screen
distanceFromLeft = 40

// distance of blue view from the right of screen
distanceFromRight = 40

// distance of blue view from the top of screen
distanceFromTop = 60

// distance of blue view from the bottom of screen
distanceFromBottom = 60

// width of the blue view
viewWidth = screenWidth - distanceFromLeft - distanceFromRight
viewWidth = 375 - 40 - 40
viewWidth = 295

// height of the blue view
viewHeight = screenHeight - distanceFromTop - distanceFromBottom
viewHeight = 667 - 60 - 60
viewHeight = 547
```

Now x, y, width and height of the view is known, Auto Layout will render it nicely across multiple screen size and orientation :



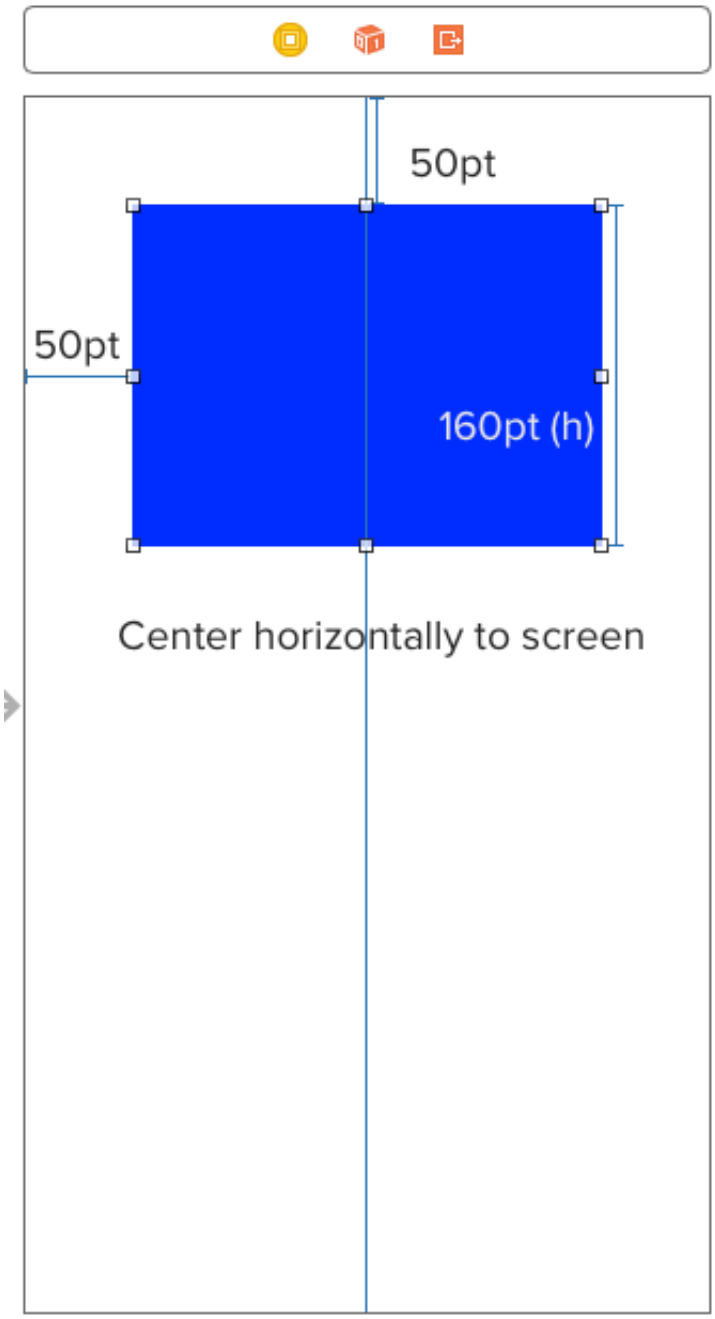
iPhone SE | Portrait



iPhone 8 | Portrait

Example 4

This example will be a bit more complex than previous ones, the constraints are defined like this :



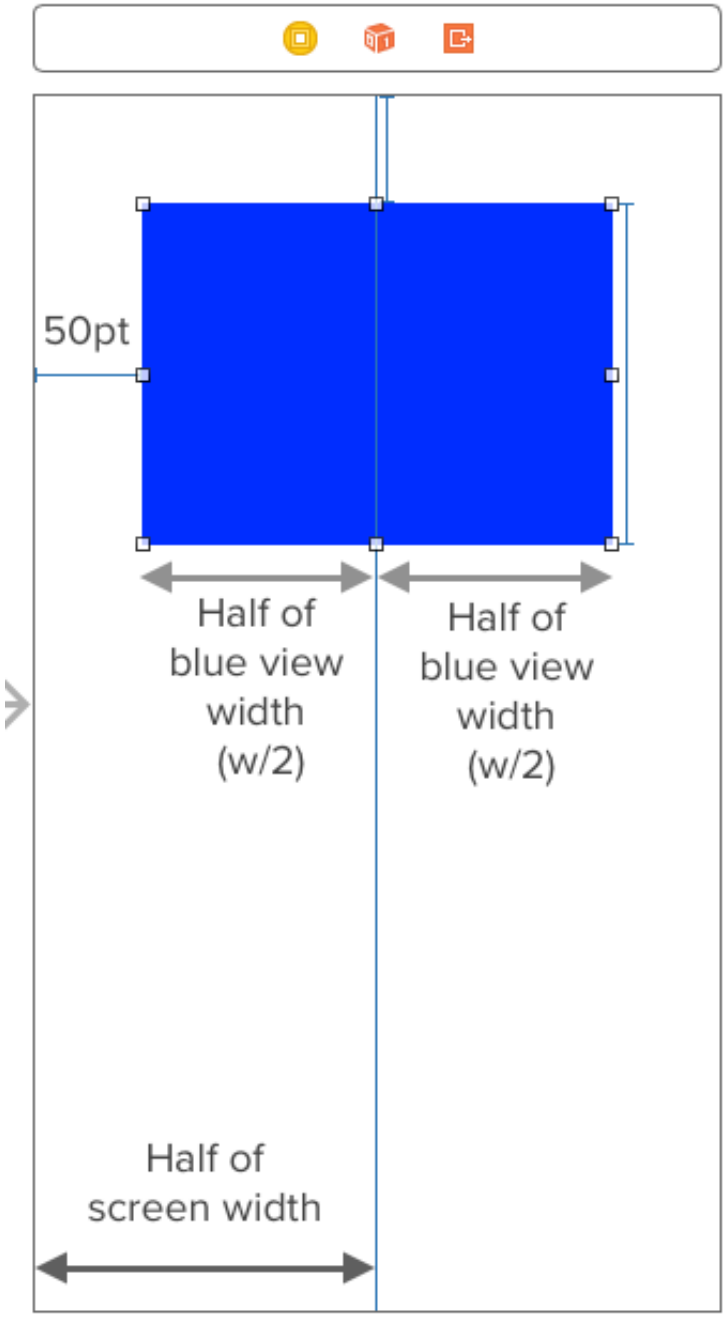
4 constraints are defined :

1. Distance from the screen top to the top of the blue view is 50 pt
2. Distance from the screen left to the left of the blue view is 50 pt
3. Height of the blue view is 160 pt
4. Blue view is **horizontally centered** to the screen

From the 1st constraint, we can deduce that the y position of the view is 50.

From the 2nd constraint, we can deduce that the x position of the view is 50.

The remaining step is to calculate the width of the blue view since height is given. Since blue view is horizontally centered, it means that the width of its left half is same as the right half, like this :



From the above diagram, we can deduce that :

```
half of screen width = 50 + half of blue view width
0.5 * screenWidth = 50 + ( viewWidth * 0.5 )
```

To calculate **width** of the blue view, Auto Layout will use the screen width. Calculation on an iPhone SE (screen width 320 pt) will look like this :

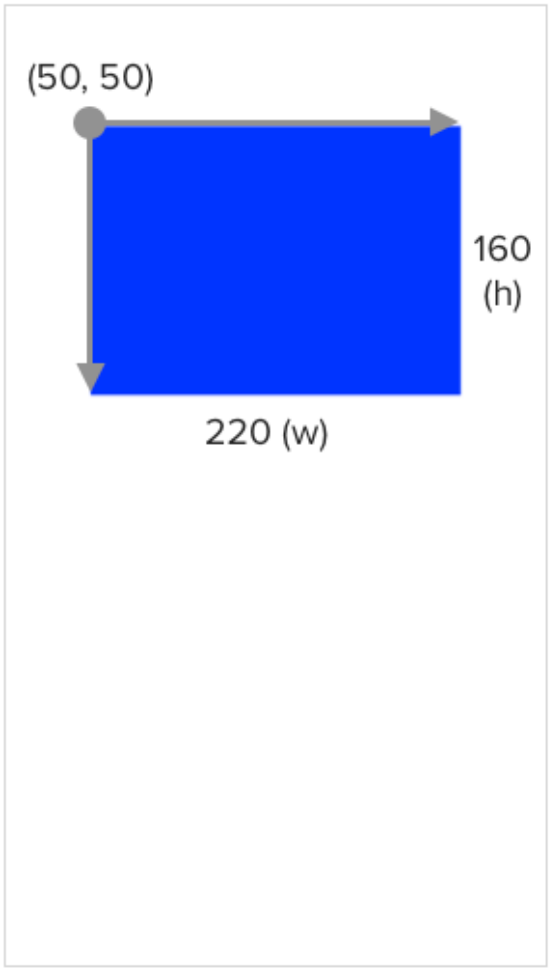
```
// iPhone SE
screenWidth = 320
```

```
// distance of blue view from the left of screen
distanceFromLeft = 50

distanceFromLeft + (viewWidth / 2 ) = screenWidth / 2
50 + (viewWidth / 2) = 320 / 2
50 + (viewWidth / 2) = 160
viewWidth / 2 = 160 - 50
viewWidth / 2 = 110
viewWidth = 110 * 2
viewWidth = 220

// width of blue view is 220
```

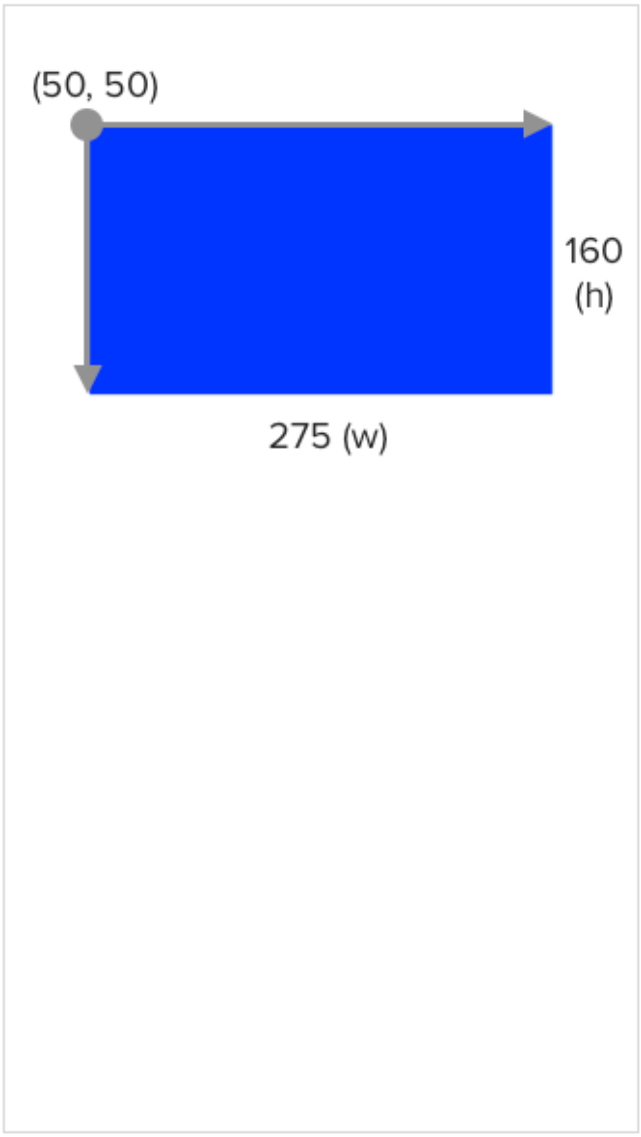
Using the same calculation formula, iPhone 8 (screen width 375, height 667) will calculate the width as 275.

```
// iPhone 8
screenWidth = 375

// distance of blue view from the left of screen
distanceFromLeft = 50

distanceFromLeft + (viewWidth / 2 ) = screenWidth / 2
50 + (viewWidth / 2) = 375 / 2
50 + (viewWidth / 2) = 187.5
viewWidth / 2 = 187.5 - 50
viewWidth / 2 = 137.5
viewWidth = 137.5 * 2
viewWidth = 275

// width of blue view is 275
```

Now x, y, width and height of the view is known, Auto Layout will render it nicely across multiple screen size and orientation :



iPhone SE | Portrait



iPhone 8 | Portrait

This example is useful for cases where you want to center a view with equal left spacing and right spacing. Instead of setting left space constraint and right space constraint separately to the same number, you can just set the left space constraint and set a horizontally center constraint on the view. It makes it easier when you want to change the spacing distant in the future, where you only need to edit one constraint instead of two.

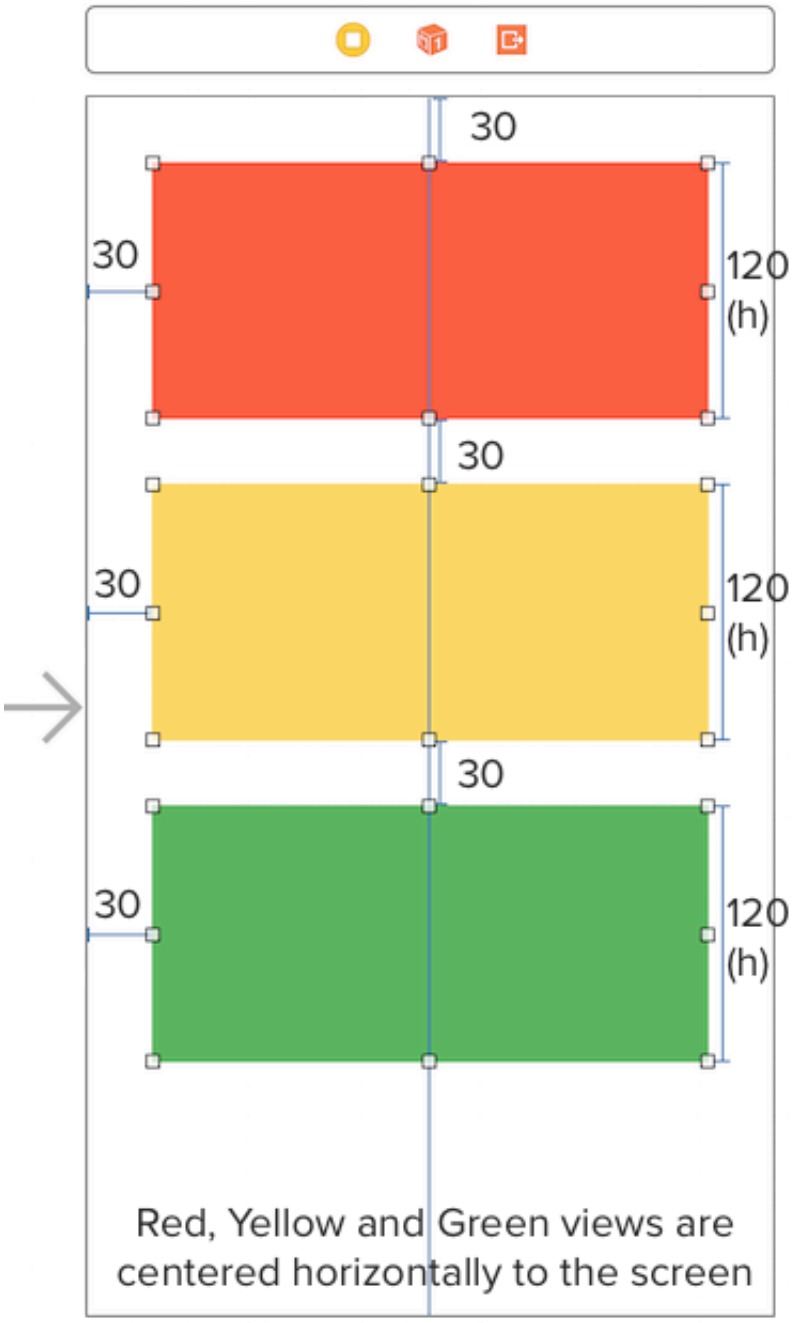
Exercise - Multiple View

We have discussed various examples and now its time to do some hands on calculation 📝! Although previous examples only used one view, the same concept and calculation still applies for multiple view.

Here is the exercise question :

Calculate the **x position**, **y position**, **width** and **height** of red, yellow and green view. Assuming iPhone SE screen size is used (**320**pt x **568** pt)

The constraints are defined as follow :

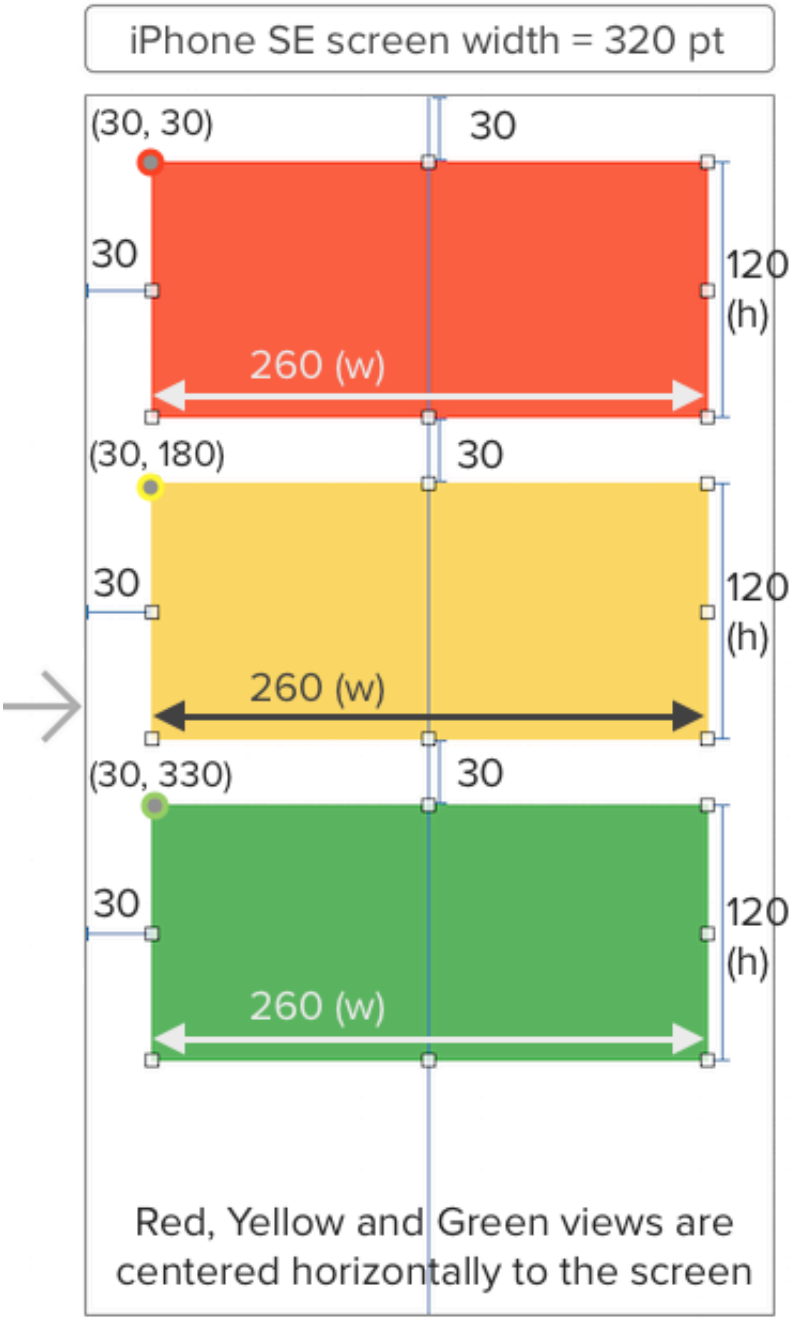


1. Distance from the screen top to the top of red view is 30 pt
2. Distance from the screen left to the left of red view is 30 pt
3. Height of red view is 120 pt
4. Distance from bottom of red view to top of yellow view is 30 pt
5. Distance from the screen left to the left of yellow view is 30 pt
6. Height of yellow view is 120 pt
7. Distance from bottom of yellow view to top of green view is 30 pt
8. Distance from the screen left to the left of green view is 30 pt
9. Height of green view is 120 pt
10. Red view is horizontally centered on the screen
11. Yellow view is horizontally centered on the screen
12. Green view is horizontally centered on the screen

Hint : Start calculating from the view at the top first then slowly proceed to bottom.

Exercise Answer

On iPhone SE screen :



Red View :

1. X position is 30
2. Y position is 30
3. Width is 260
4. Height is 120

Yellow View :

1. X position is 30
2. Y position is 180
3. Width is 260
4. Height is 120

Green View :

1. X position is 30
2. Y position is 330
3. Width is 260
4. Height is 120

This is how it looks like in multiple screen sizes :

Summary

The takeaway of this chapter is that if Auto Layout can calculate all of the views' position and size using the given constraints, it will render them nicely. If there is not enough constraint, Auto Layout will not be able to calculate position/size of some view, then it will attempt to 'guess' their position/size and thats the reason why you see some layout going haywire when constraints are not defined correctly.

Here is the checklist to ensure Auto Layout can render the views correctly.

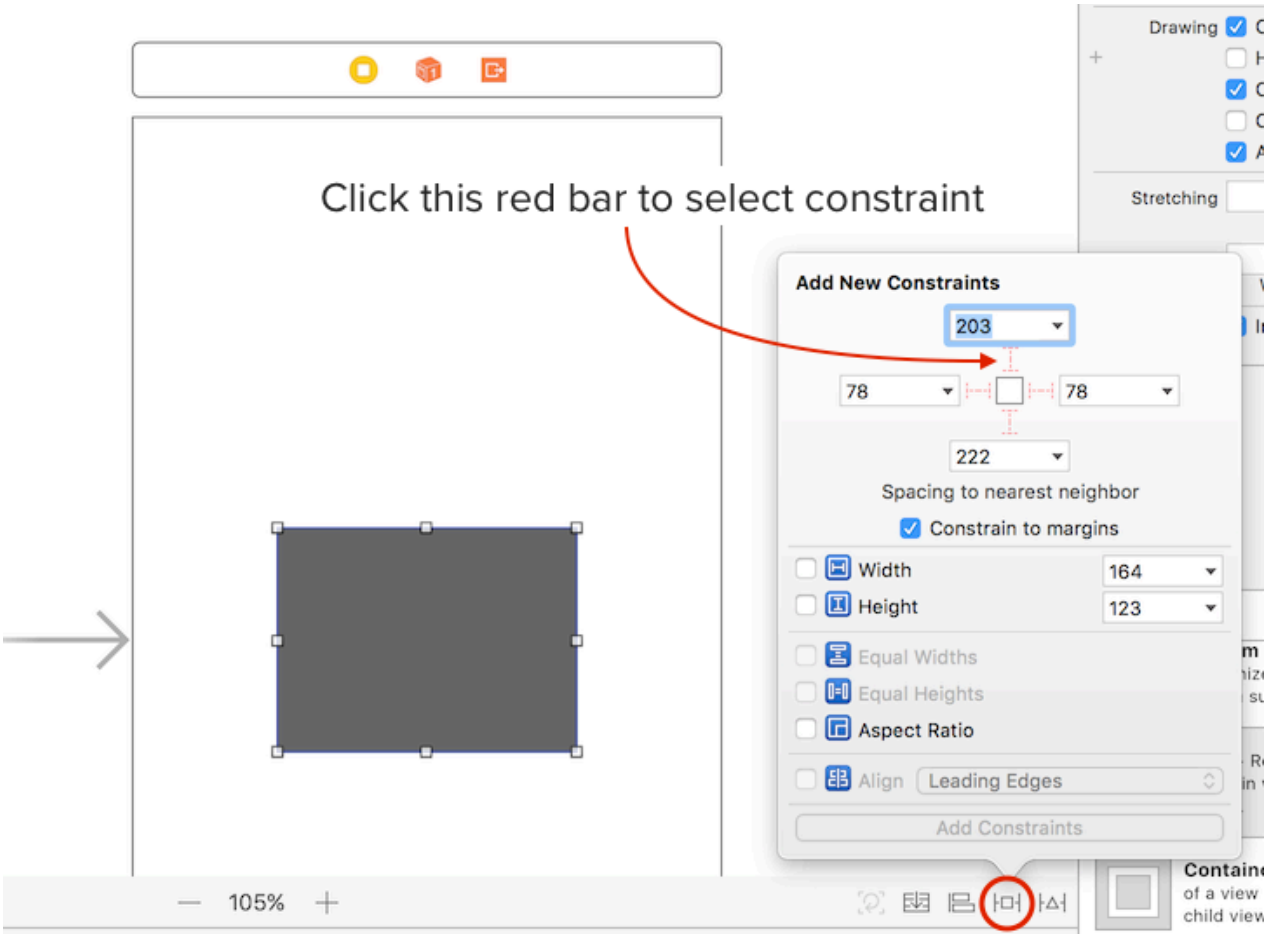
> Does the constraints defined allow Auto Layout to know / calculate :
>
> 1. **x position** of all view ✅
> 2. **y position** of all view ✅
> 3. **width** of all view ✅
> 4. **height** of all view ✅

One quick way to determine if Auto Layout is able to calculate the position and size is to calculate it yourself, by using pencil and paper. If you can't deduce them using math, Auto Layout can't either. Once you get a hang of it, you won't need to calculate these by hand, as long as you know how to define constraints that enable Auto Layout to calculate these, your layout will look great on every device.

3 - How to add, edit and delete constraint

This chapter will demonstrate how to set constraint in the interface builder in Xcode, feel free to skip this chapter if you already know how to do it. In interface builder, you can add constraint using the constraint button or by holding `control` key + drag the UI element to other UI element.

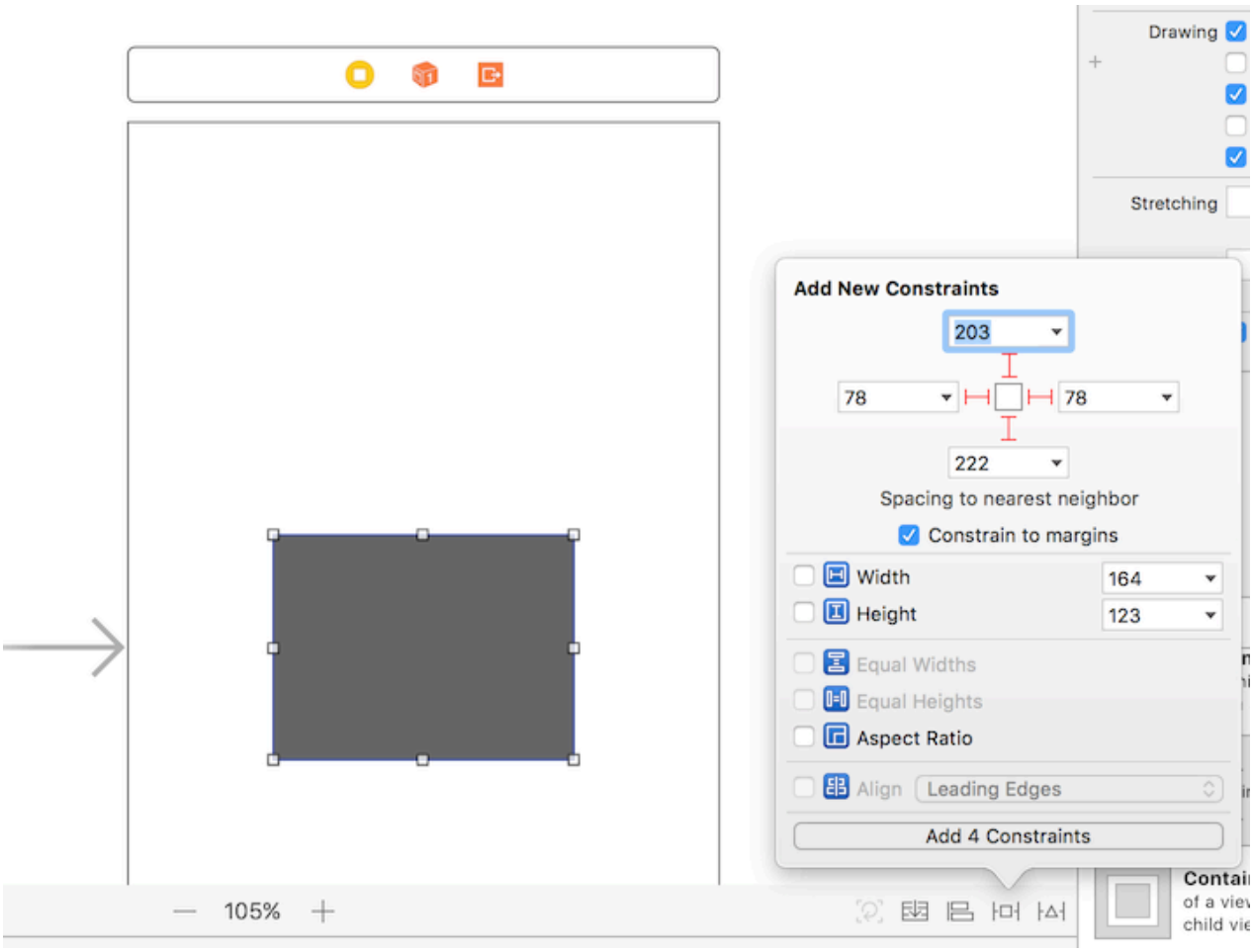
Add constraint using constraint button



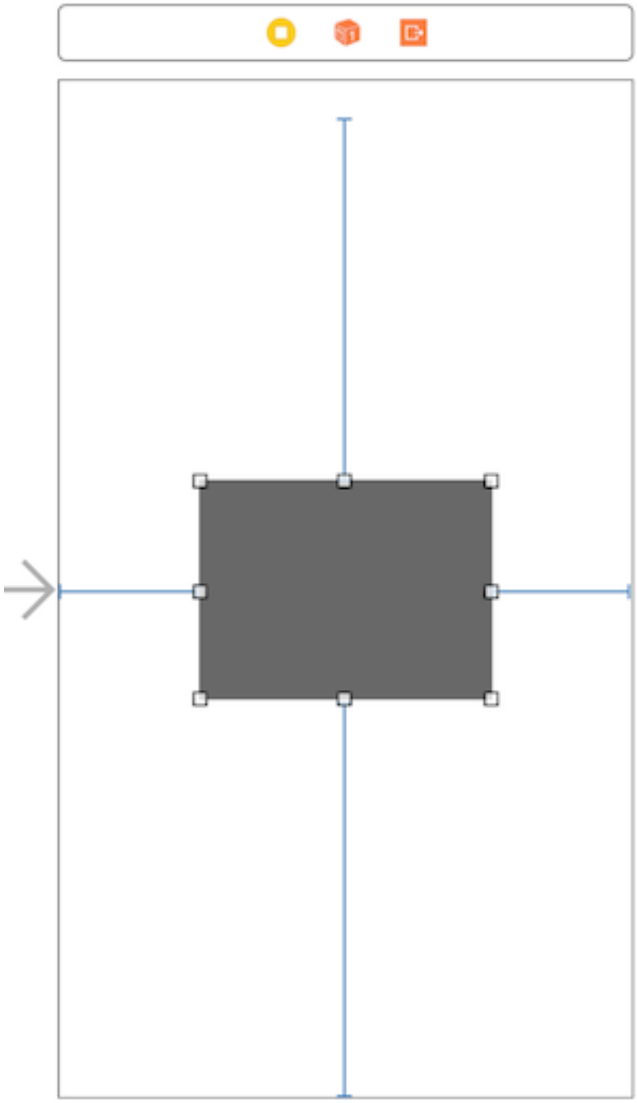
Click the **circled icon** ("Add new constraint") as shown above, then select the side of constraint you want by clicking on the red bar or change the number value of the red bar (Xcode will auto select the red bar after you change the value).

The four red bar means distance of four sides (top , left, right, bottom) from the selected view to its neighbour. If there is no surrounding neighbour, then it will be the distance of the view to the **safe area**. We will explain more about safe area later in this chapter.

To demonstrate, if you select four side of the red bars like this and click "Add 4 Constratins" :

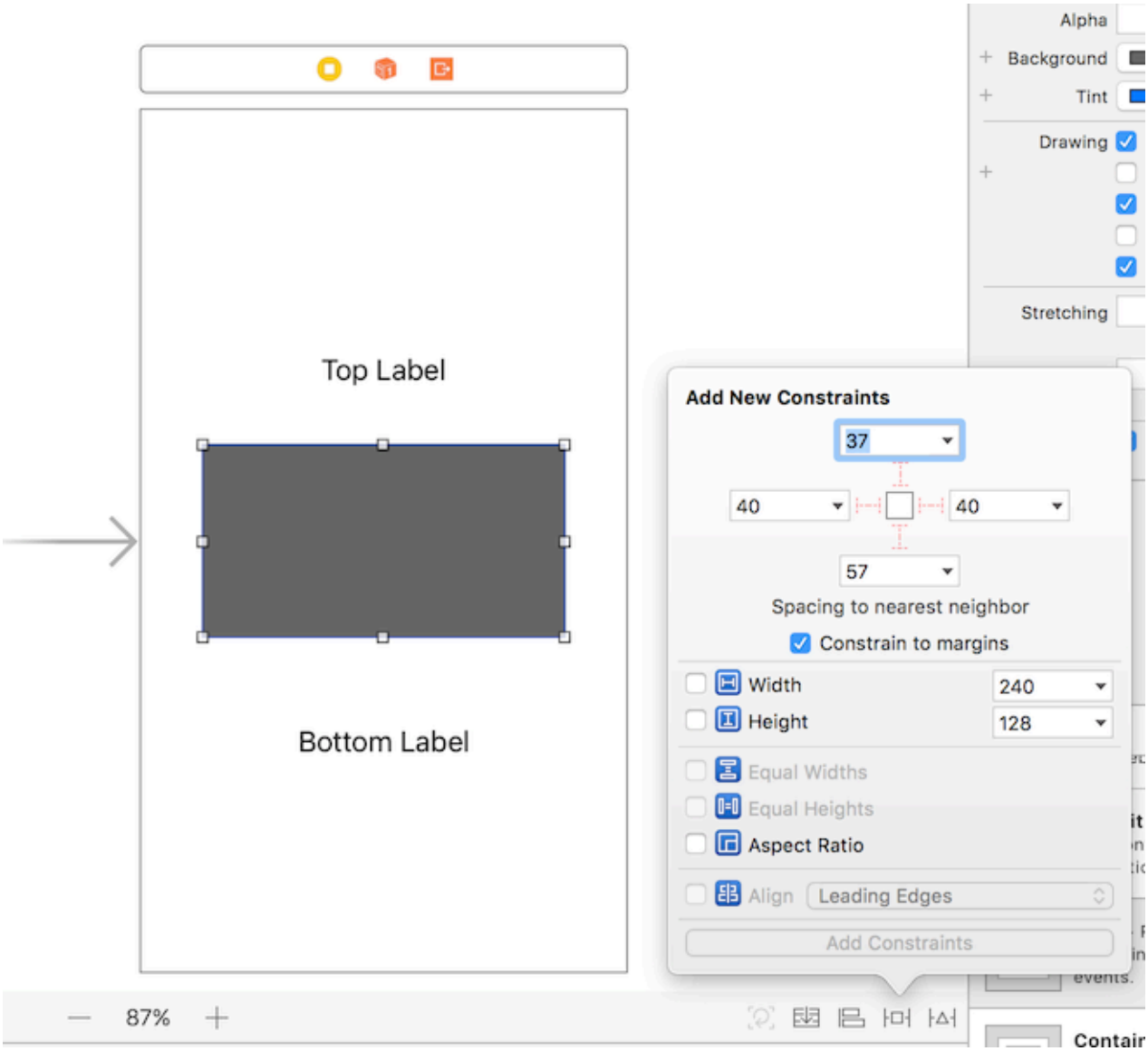


Then Xcode will create the leading (left), top, trailing (right) and bottom constraint for the view like this :

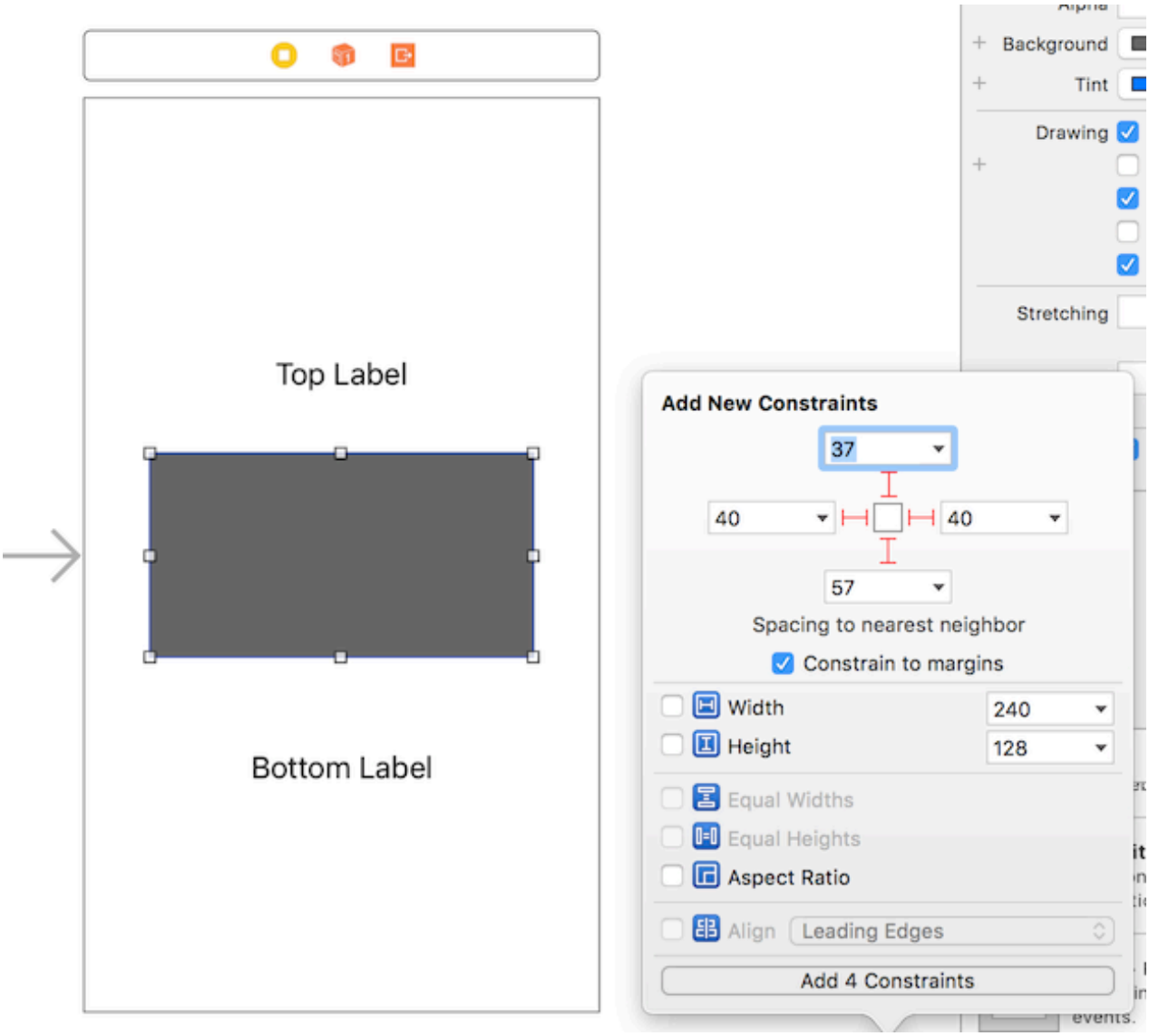


Constraint to nearest neighbour

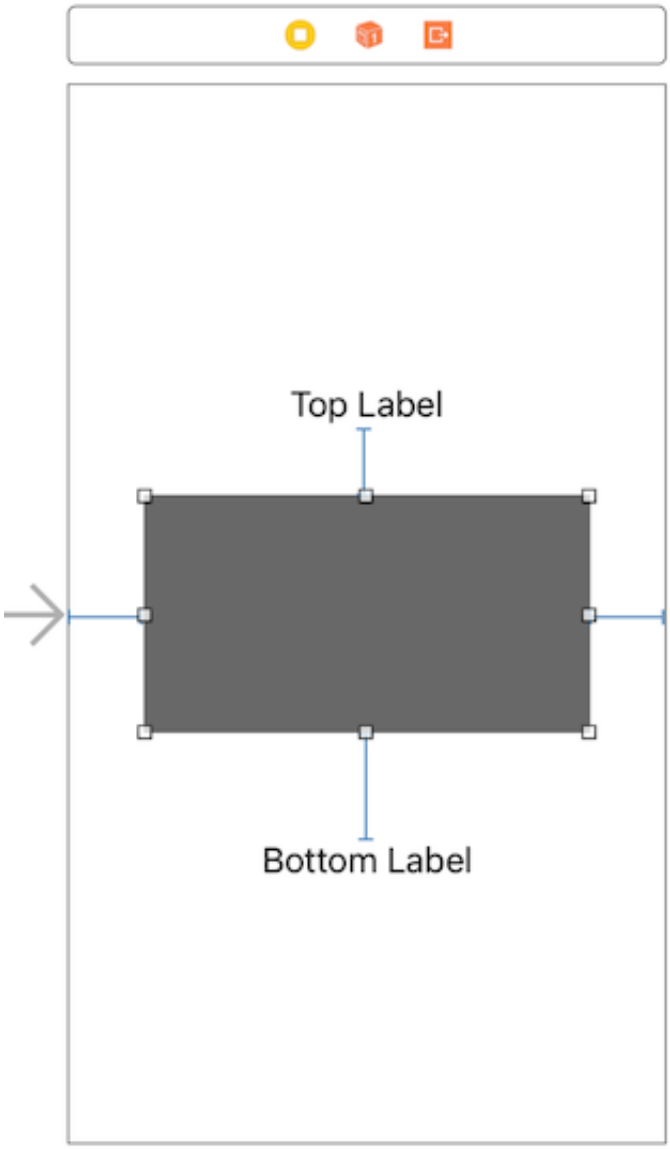
If there are neighbouring UI elements , then the constraint distance will be set relative to the neighbour UI element. For the example below, there are two labels on the top and bottom of the gray view:



Let's select the middle gray view and add constraint to it:



After clicking "Add 4 Constraints", we can see that the top and bottom constraint is set relative to the top label and bottom label as they are the nearest neighbouring element to gray view.

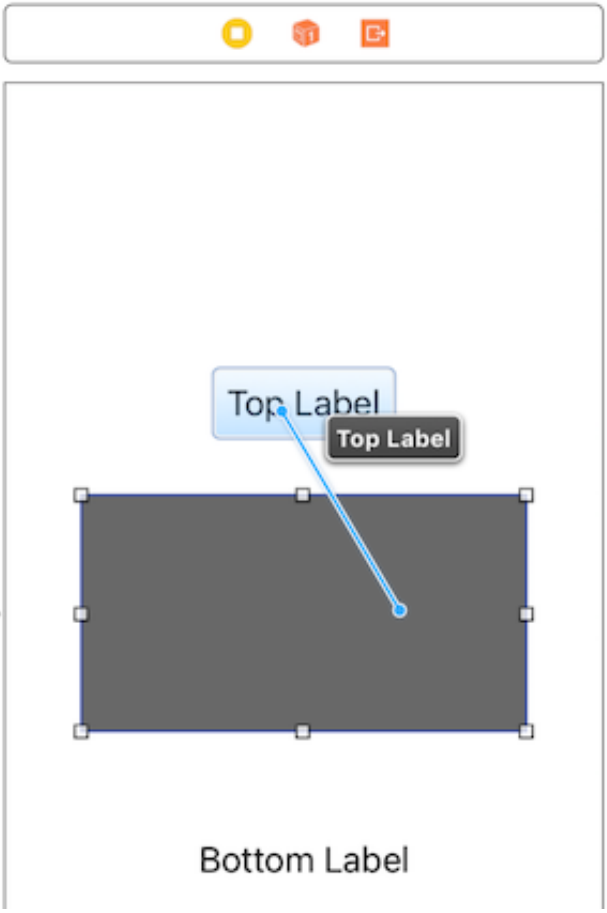

Top Label
Bottom Label

Add constraint by holding control key and drag

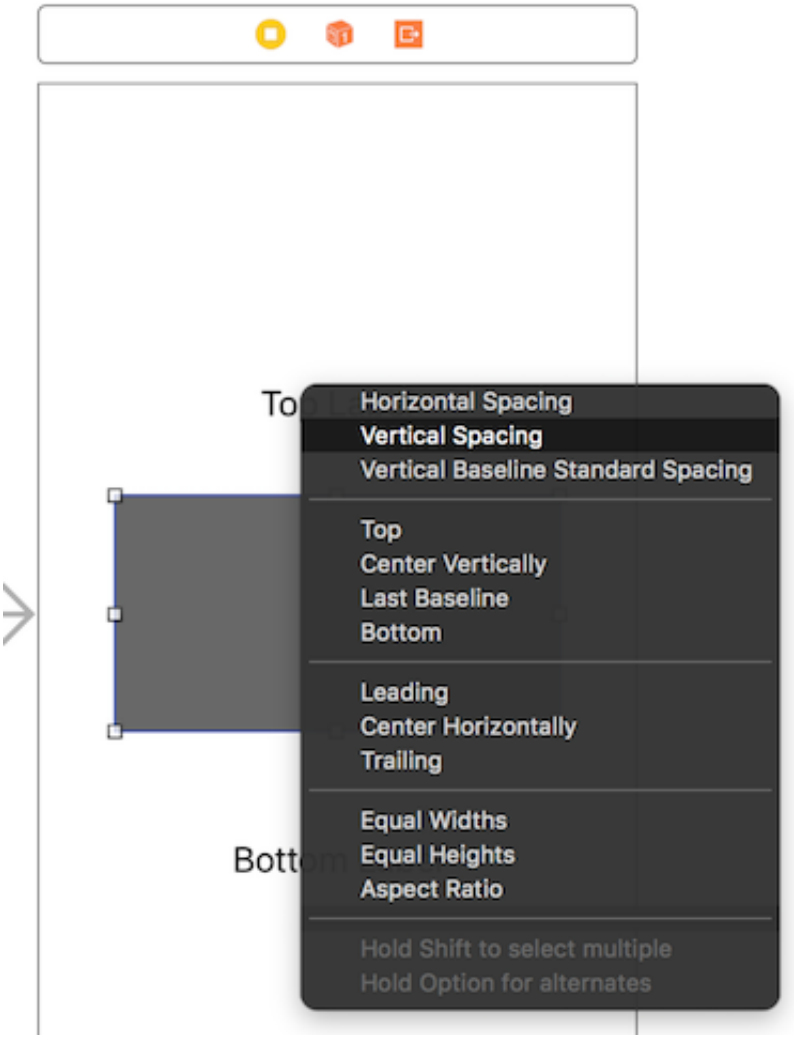
You can also add constraint to an UI element by dragging it to other UI element.

For the example below, we will add the vertical distance constraint from gray view to the top label.

Select the gray view, then press and hold `control` key, then drag the mouse cursor (while still holding `control` key) from gray view to the top label.



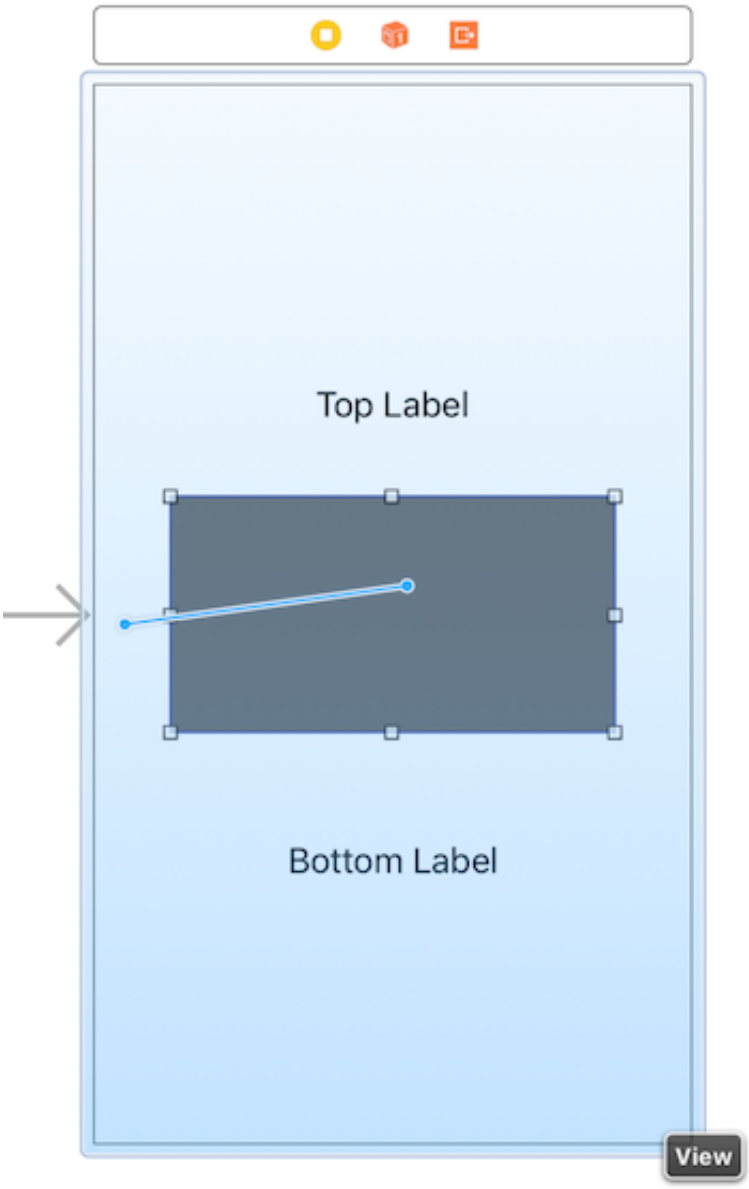
After letting go the mouse click / trackpad, a menu will appear that let you choose what constraint to add.



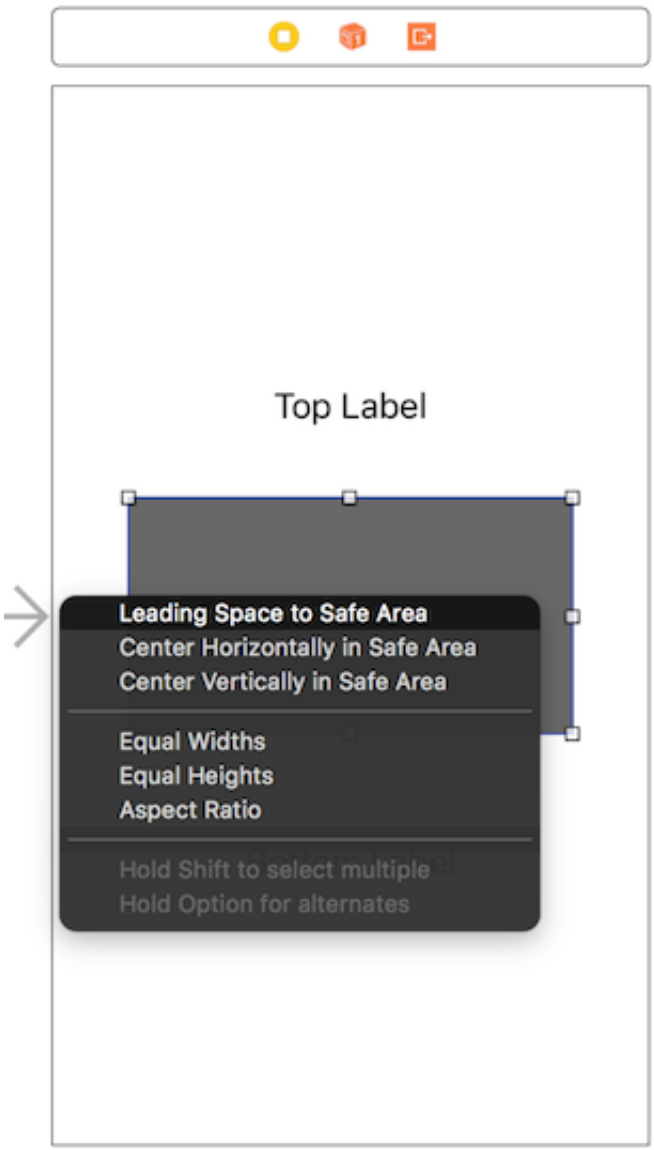
In this case we will select "Vertical spacing" as we will want to add the vertical distance constraint between the gray view and the top label. You can try out other constraint too.

Dragging to parent view

You can also hold `control` key + drag the UI element to its parent too, like this :



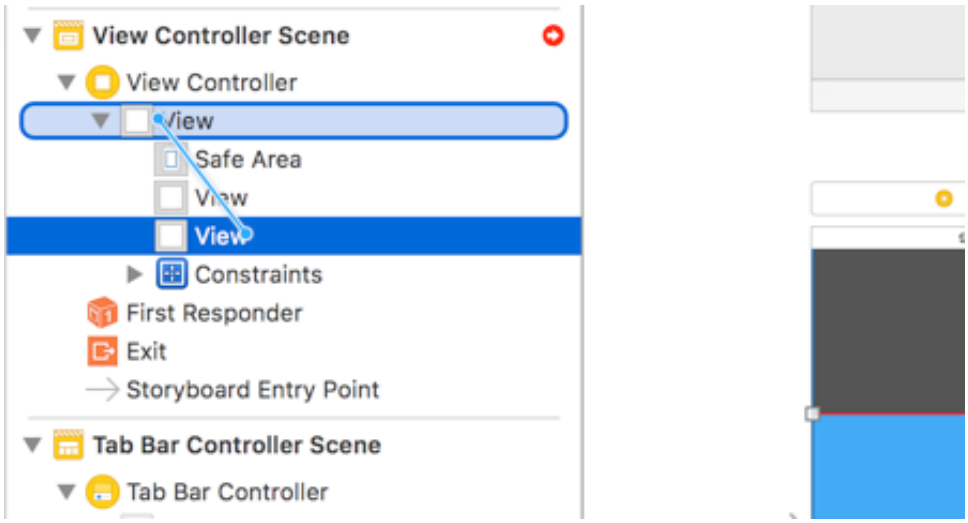
Then a similar menu will appear after you let go of the mouse drag.

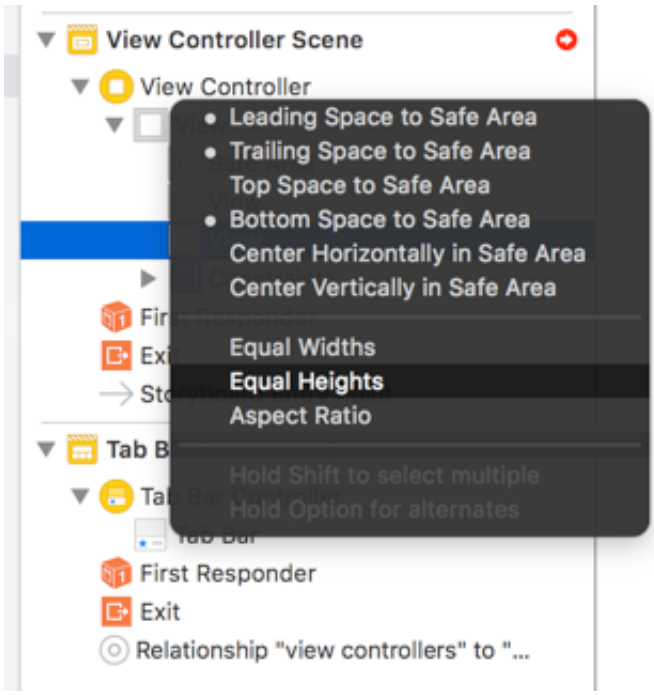


Selecting "Leading Space to Safe Area" will add a constraint from the left of the screen edge to the left of the gray view.

Dragging in Document Outline

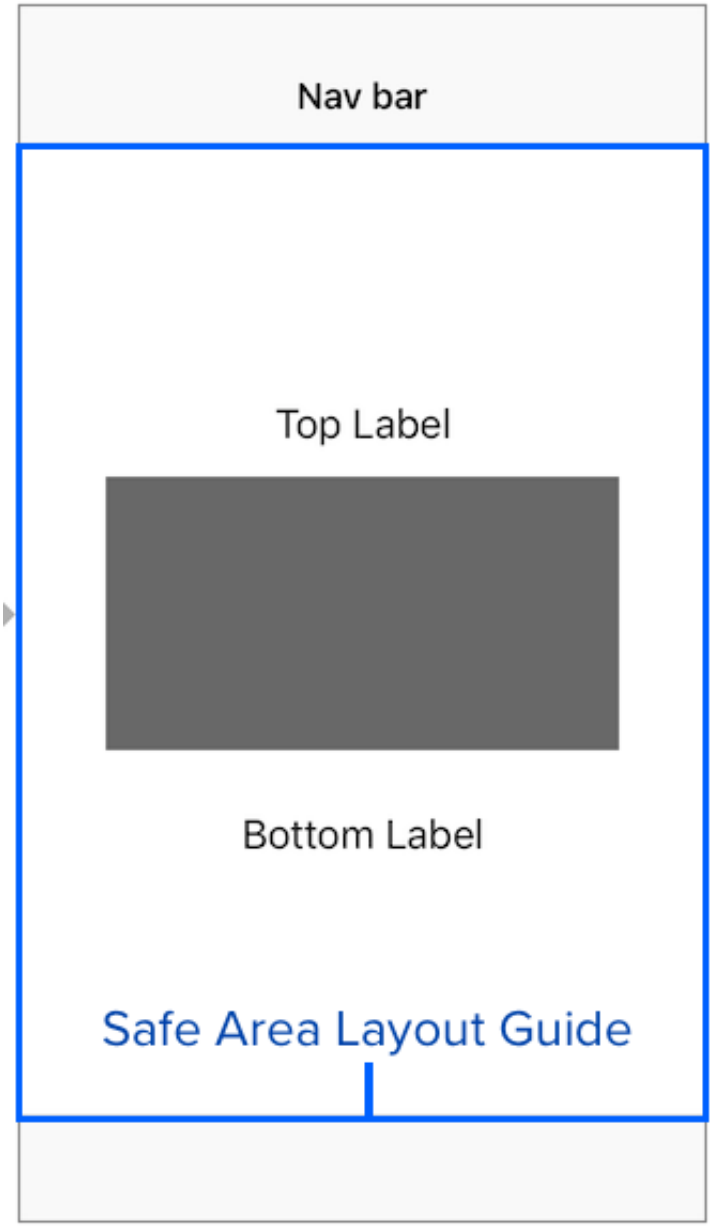
Sometimes there might be multiple views which overlap in Interface Builder which make it hard to control + drag to the correct view. In situation like this, we can drag (Hold control + drag) in the document outline directly :





Safe Area layout guide

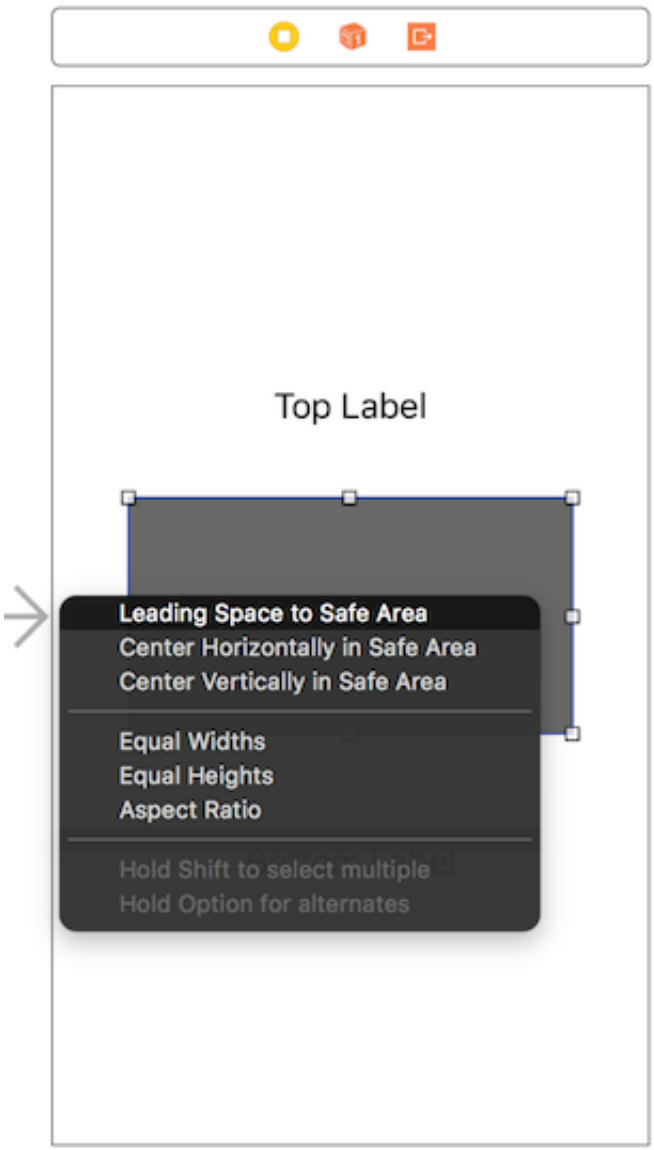
Safe Area layout guide is introduced in iOS 11. Setting constraint to safe area will ensure that the UI element won't get hidden beneath nav bar, tab bar or the rounded corner section of iPhone X.



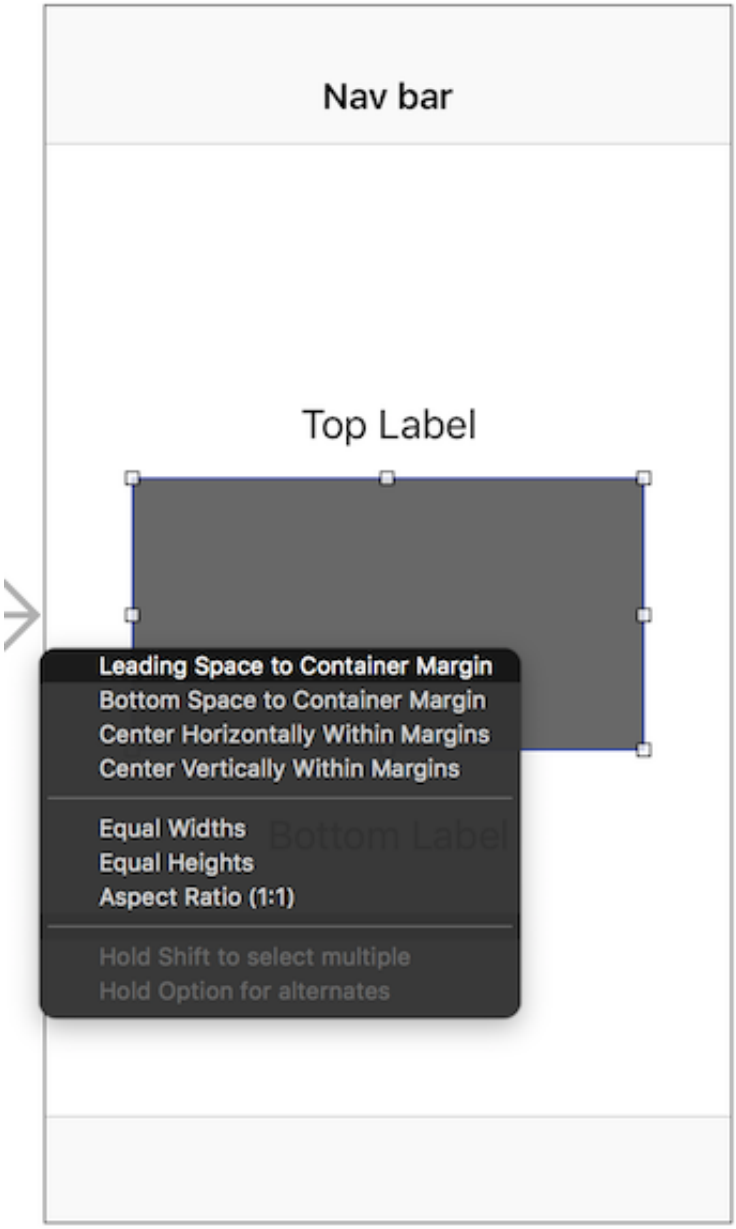
In iPhone X / XR / XS screen, the safe area looks like this :

Alternate option in menu

As mentioned in previous section, after releasing mouse drag, you will see a menu showing list of available constraint like this:



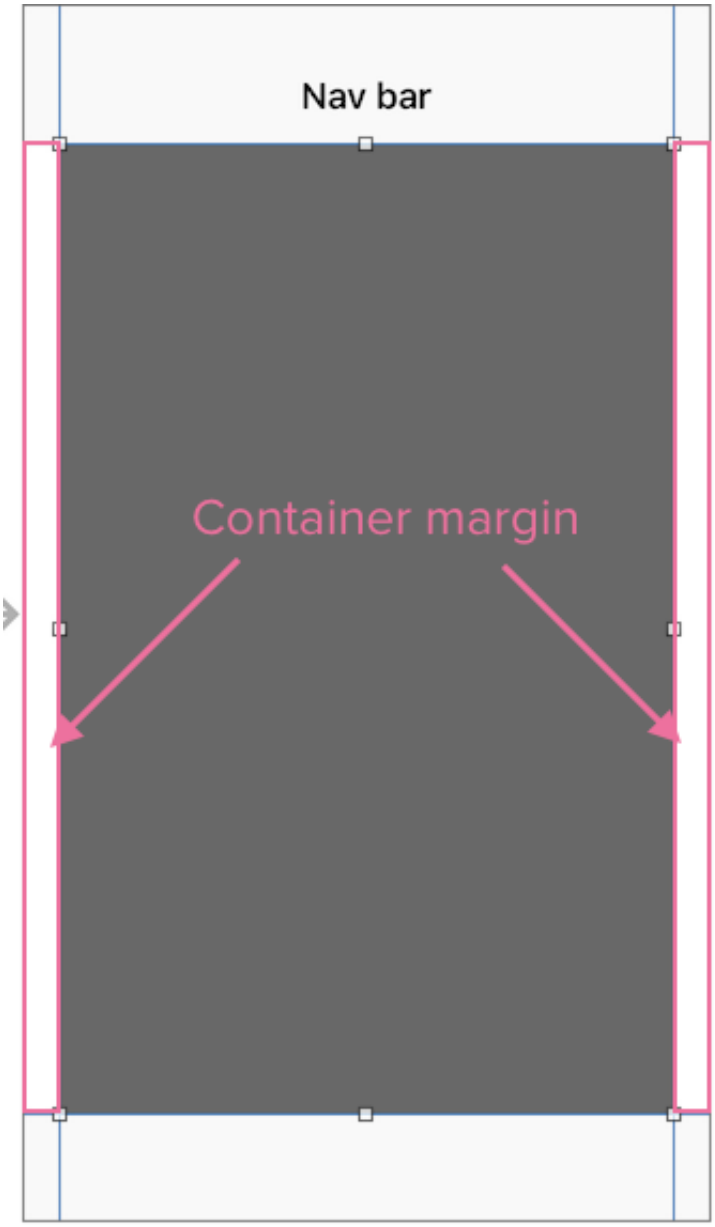
Now if you press and hold `alt` key, you will see a different menu being shown:



Container margin

What is container margin you ask? It is like Safe Area, but with a system defined margin space put in between the safe area and the view inside safe area.

I tried to set constraints with 0 distance from the gray view to four side of the container margin, the result looks like this:

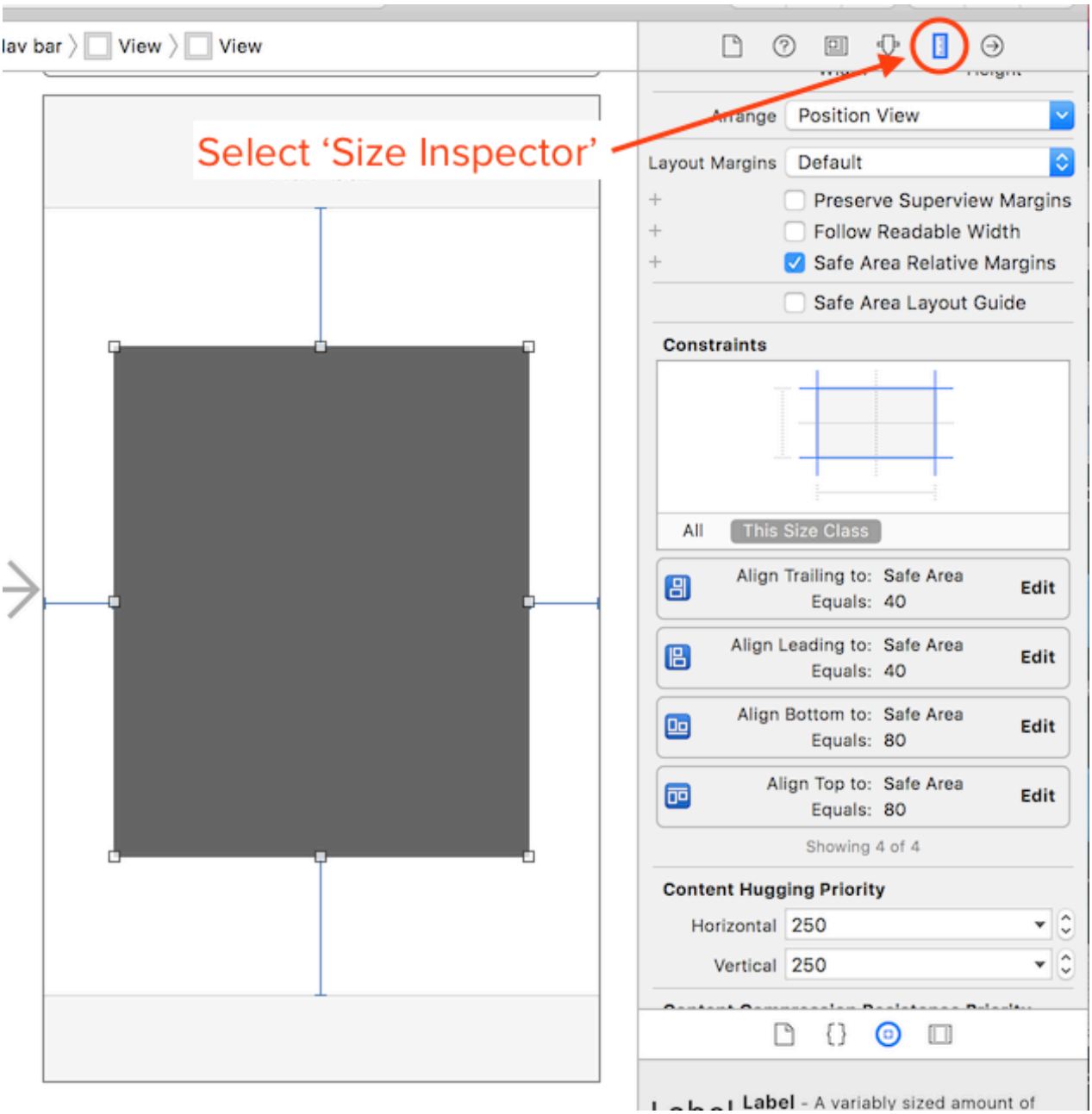


In iOS 11 iPhone SE, the margin is 16pt on both the left side and right side of the safe area. However, the margin for both top and bottom side is 0pt. This margin value / distance is system defined, Apple might change this margin value in future release of iOS. Keep it in mind if you are planning to use container margin.

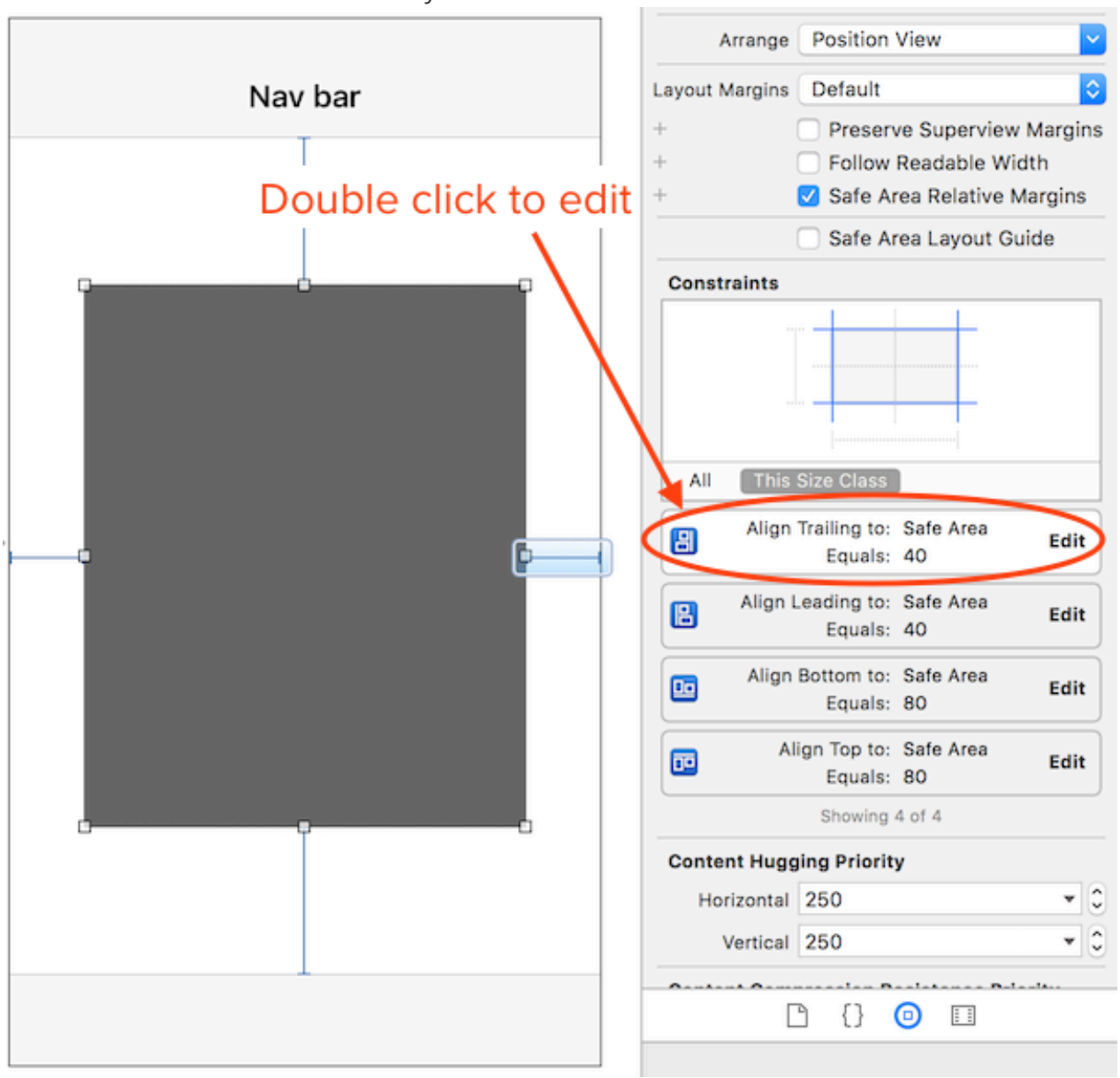
Editing constraint

Size Inspector

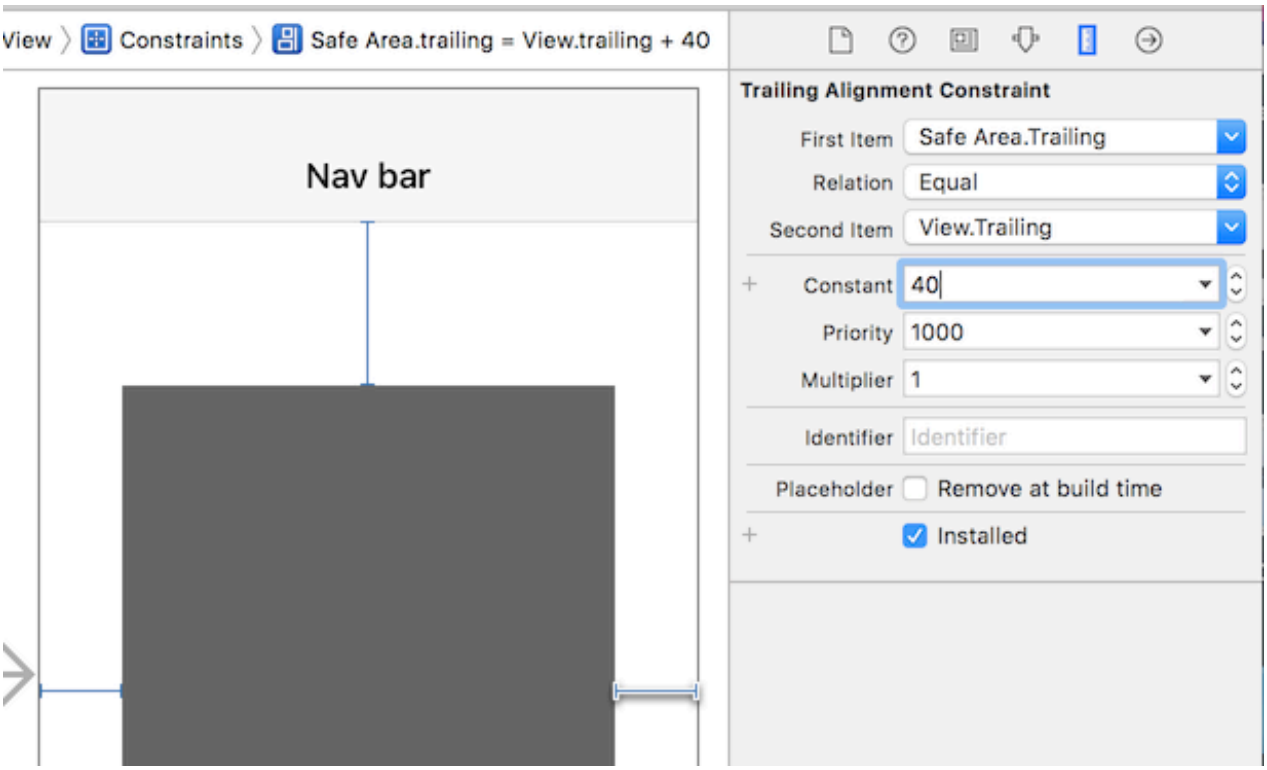
To edit a constraint value after you have created it, select the UI element and open **Size inspector** :



Then double click into the constraint you want to edit:

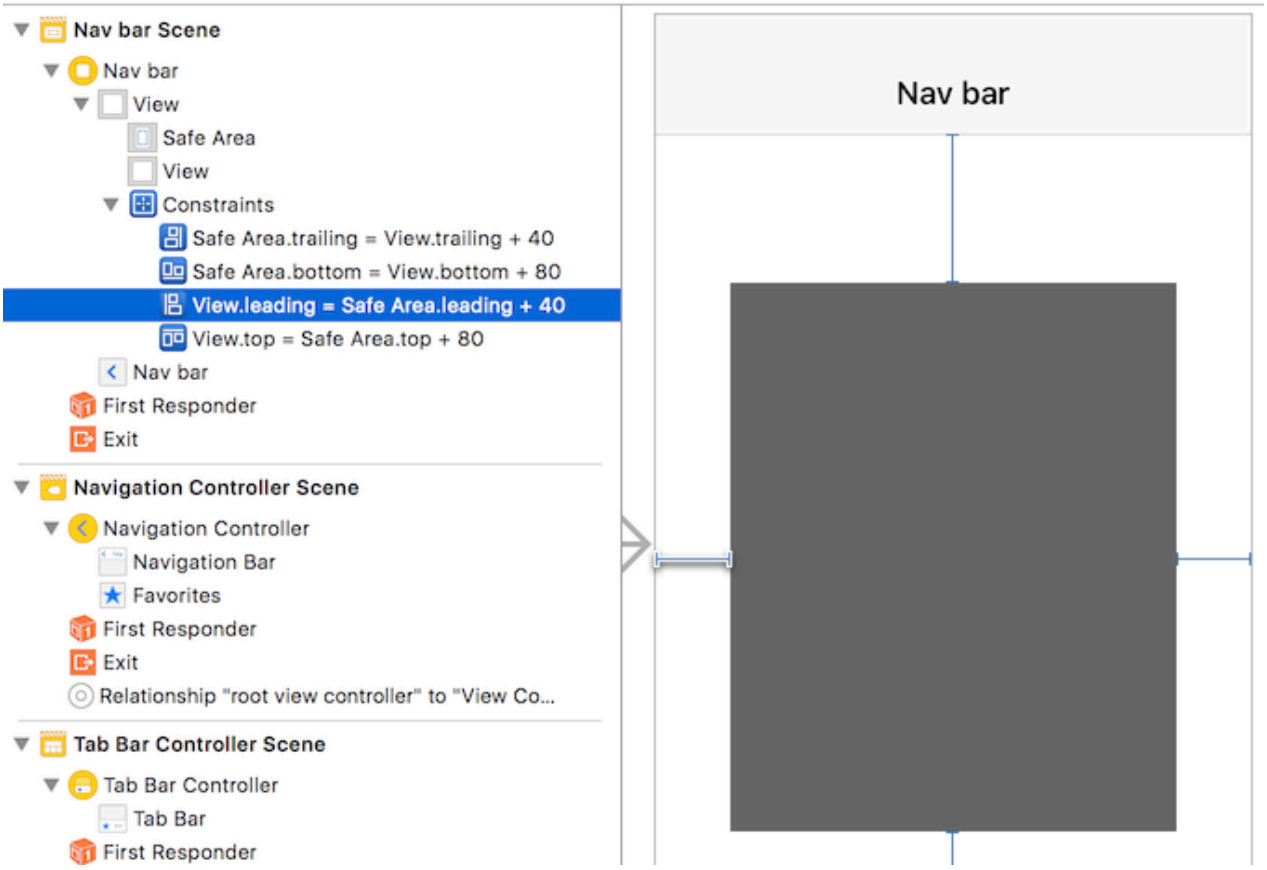


You can edit the value for the constraint, **constant** is the one you would like to edit usually. We will go into more detail about **Priority** in another chapter.



Document Outline

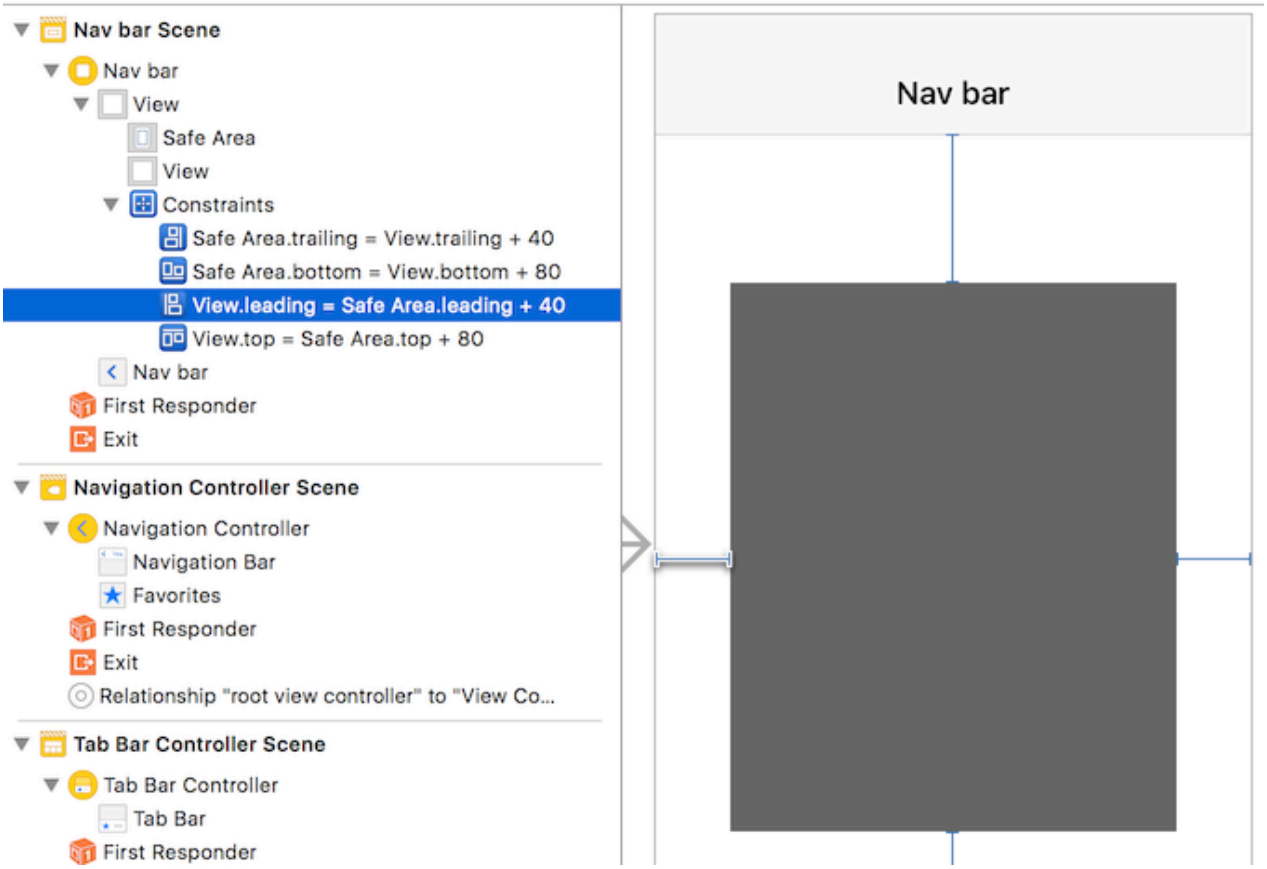
You can also select the constraint directly from the document outline on the left:



And edit the values similarly as shown in previous section.

Deleting Constraint

In the document outline, select the constraint and press `delete` to delete the constraint.



4 - Why missing constraints appear and how to solve them

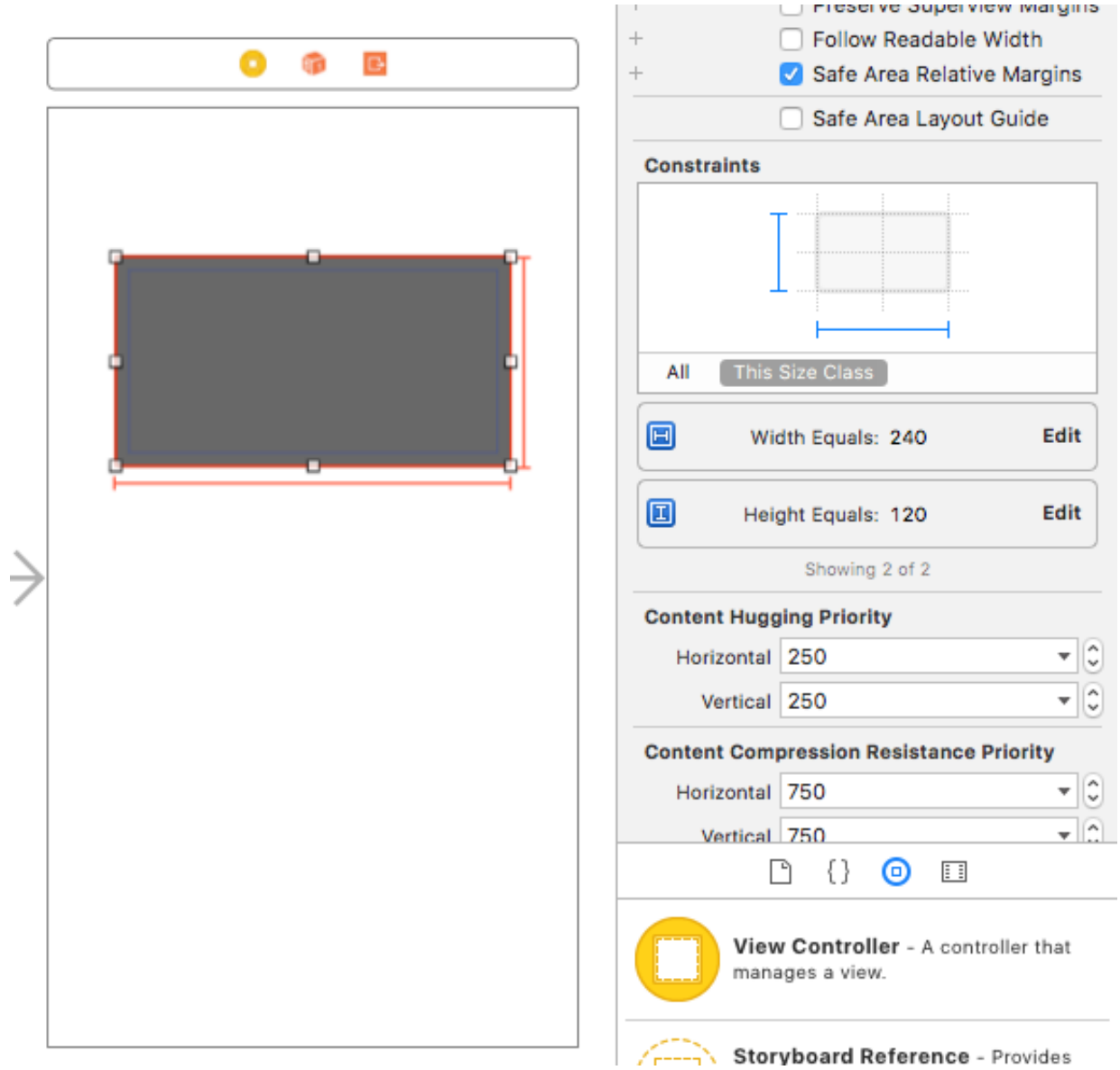
This chapter will require some knowledge from chapter 2 (How Auto Layout calculates position and size using constraints).

To recap, Auto Layout needs to know enough constraints of a view so that it can calculate :

1. **x position** of a view
2. **y position** of a view
3. **width** of a view
4. **height** of a view

Xcode shows you red line in the Interface Builder (the place where you drag and drop view controller in storyboard) **when it doesn't know where it should place the view** or/and **what size is the view**.

We will use a simple example to explain why missing constrain appear and how to solve them :

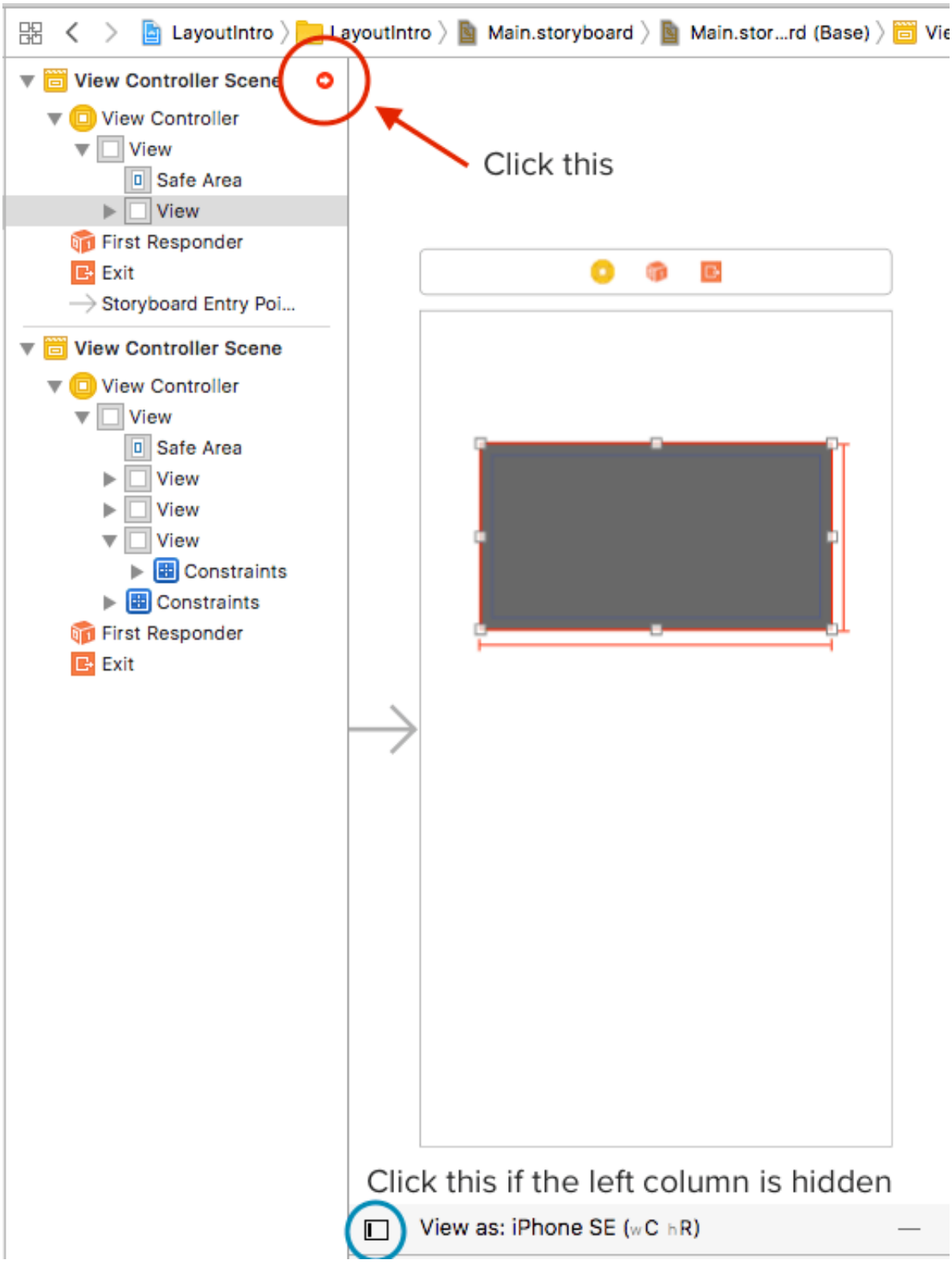


The gray view has 2 constraints placed :

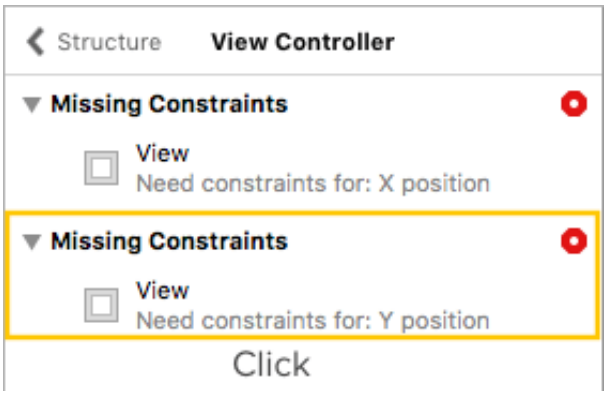
1. Width is 240
2. Height is 120

The size of it is defined but there is red line surrounding it, this is because Auto Layout doesn't know where to position it. There is no constraint that tells Auto Layout what position (x, y) it should place the gray view.

We can troubleshoot what are the missing constraints by clicking the white arrow icon inside red circle button :



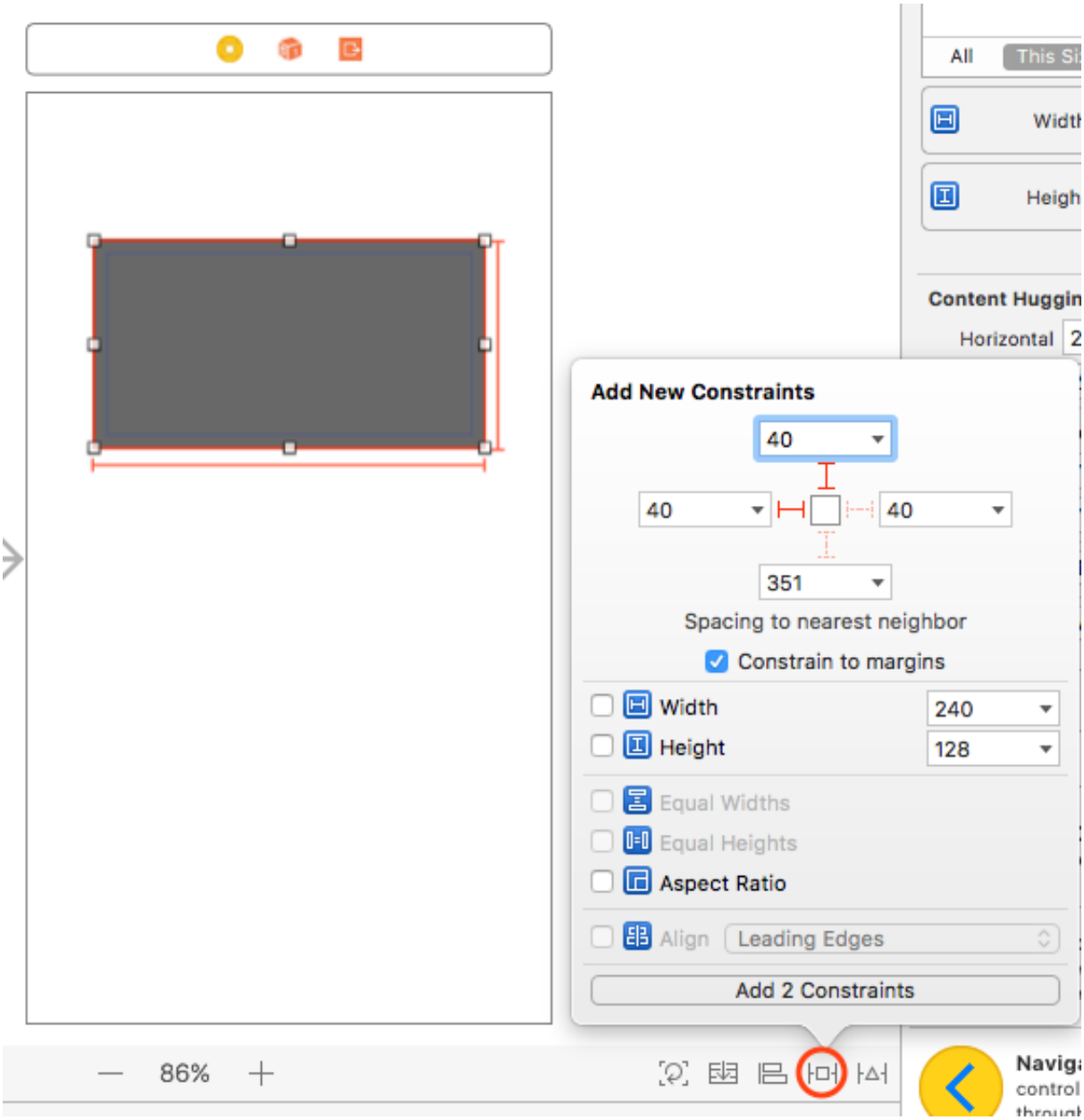
When you click on the missing constraints error, it will highlight the view that has missing constraint on Xcode.



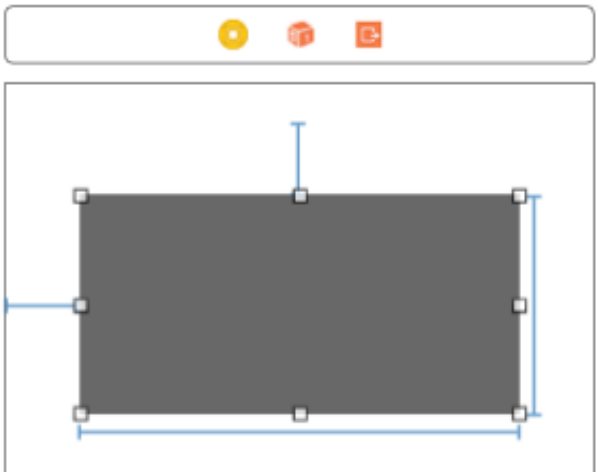
You will see Xcode complain that it needs constraints that allow it to calculate the X position and Y position of the view. Remember Auto Layout needs to know or able to calculate a view's **X position**, **Y position**, **width** and **height** to render it correctly?

But now there is no constraint which enable Auto Layout to calculate the X position and Y position. That's why Xcode surrounded the view with red lines to notify you to fix it.

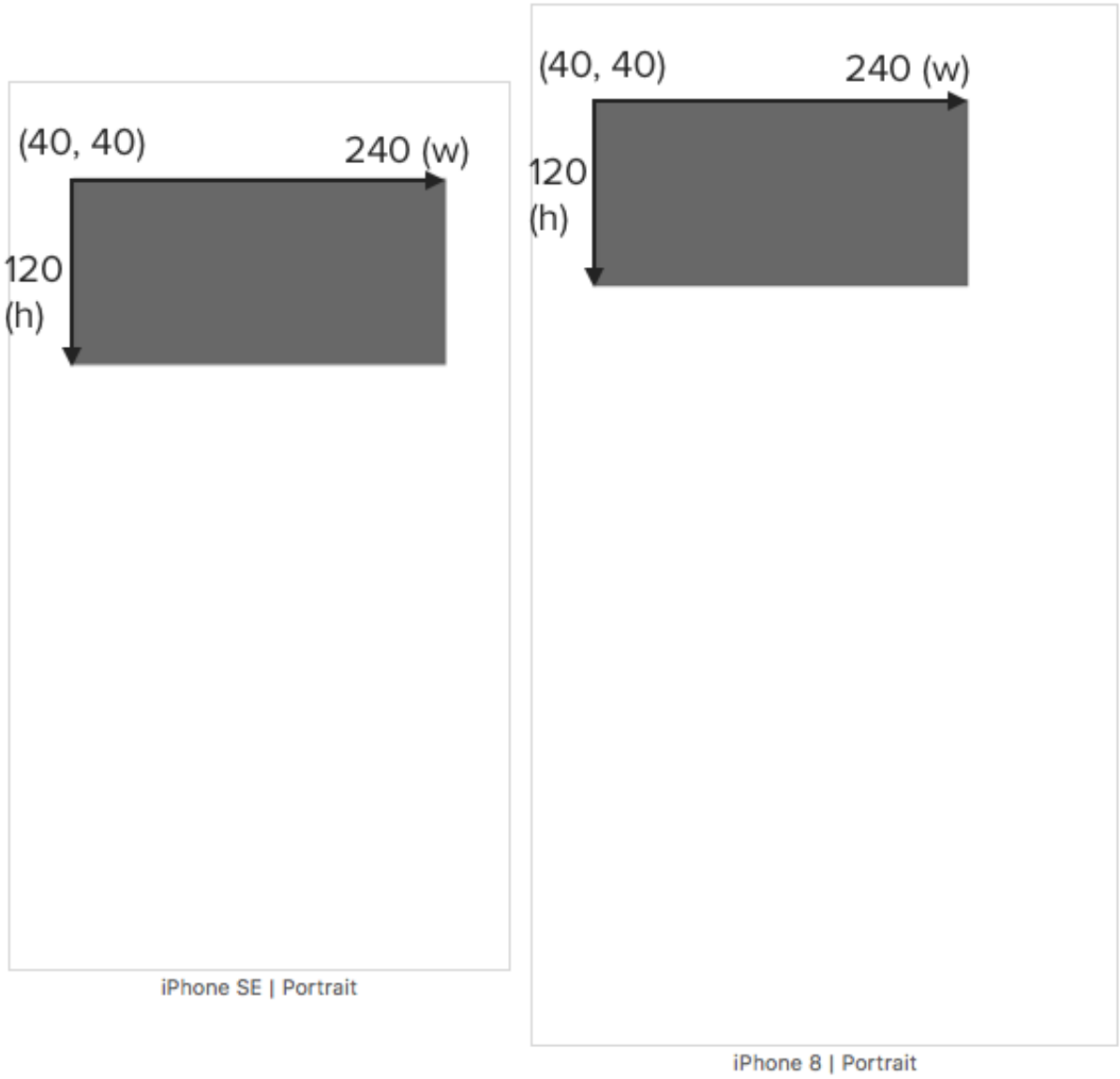
There's many way to fix it (refer to chapter 2 - how auto layout calculates position and size), the simplest way to fix it is adding a top and left constraint like this :



After adding these two constraints, you will see the red lines change into blue lines as Xcode now know where to position the view, yay!



Auto Layout now know that the view X position is 40, Y position is 40, width is 240 and height is 120. Hence it will render it nicely across devices like this :



When you stumble across red lines next time, check if there is enough constraint defined that allows Auto Layout to calculate the X, Y , width and height of that UI element.

Summary

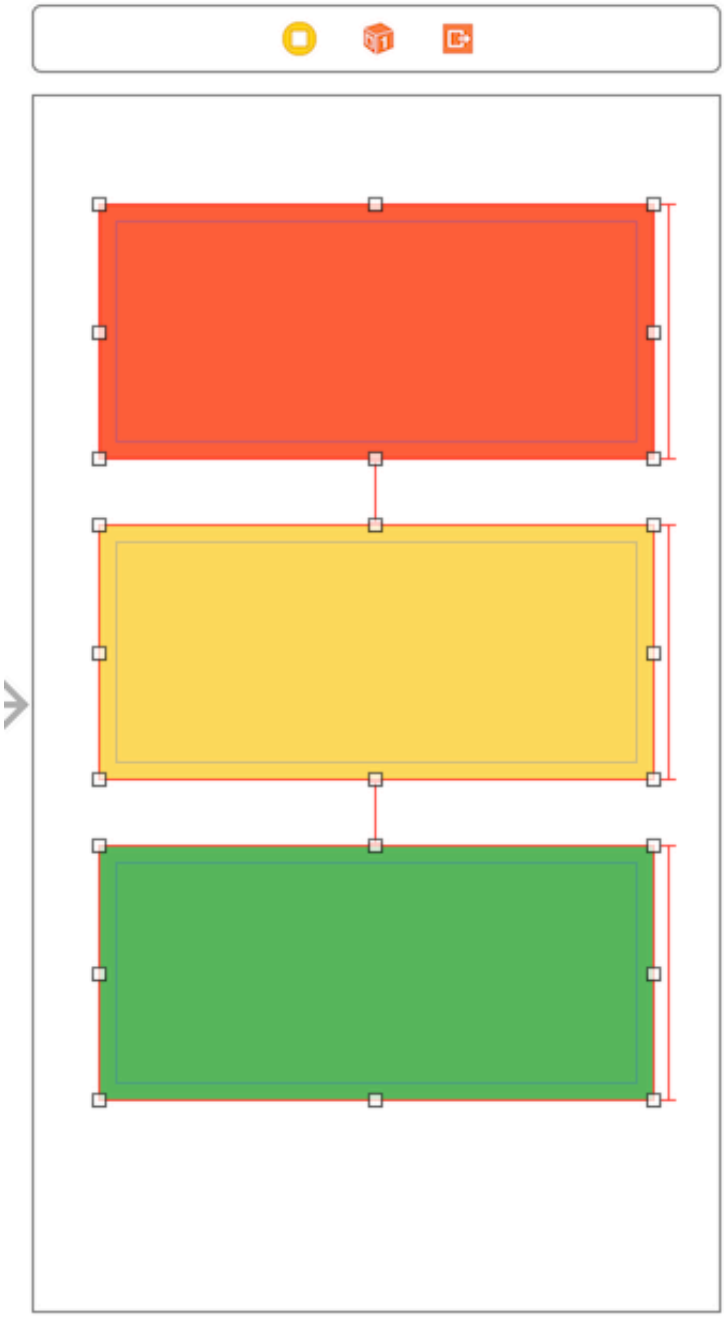
The example used only 1 view and I know that your app surely will have more than 1 view, but the cause of Xcode showing you red lines will always be that Xcode / Auto Layout doesn't know **what size is your view** or **where to place your view**.

Always add enough constraints to all the views so Xcode / Auto Layout can know or able to calculate what size is the view and where to place the view.

Exercise Project

There is an exercise Xcode project for this chapter, it is located in `exercises/missingConstraint/missingConstraint.xcodeproj` .

The project contains two storyboards, **Main.storyboard** and **Answer.storyboard**. Open the **Main.storyboard** and you will see a view controller with three views like this :



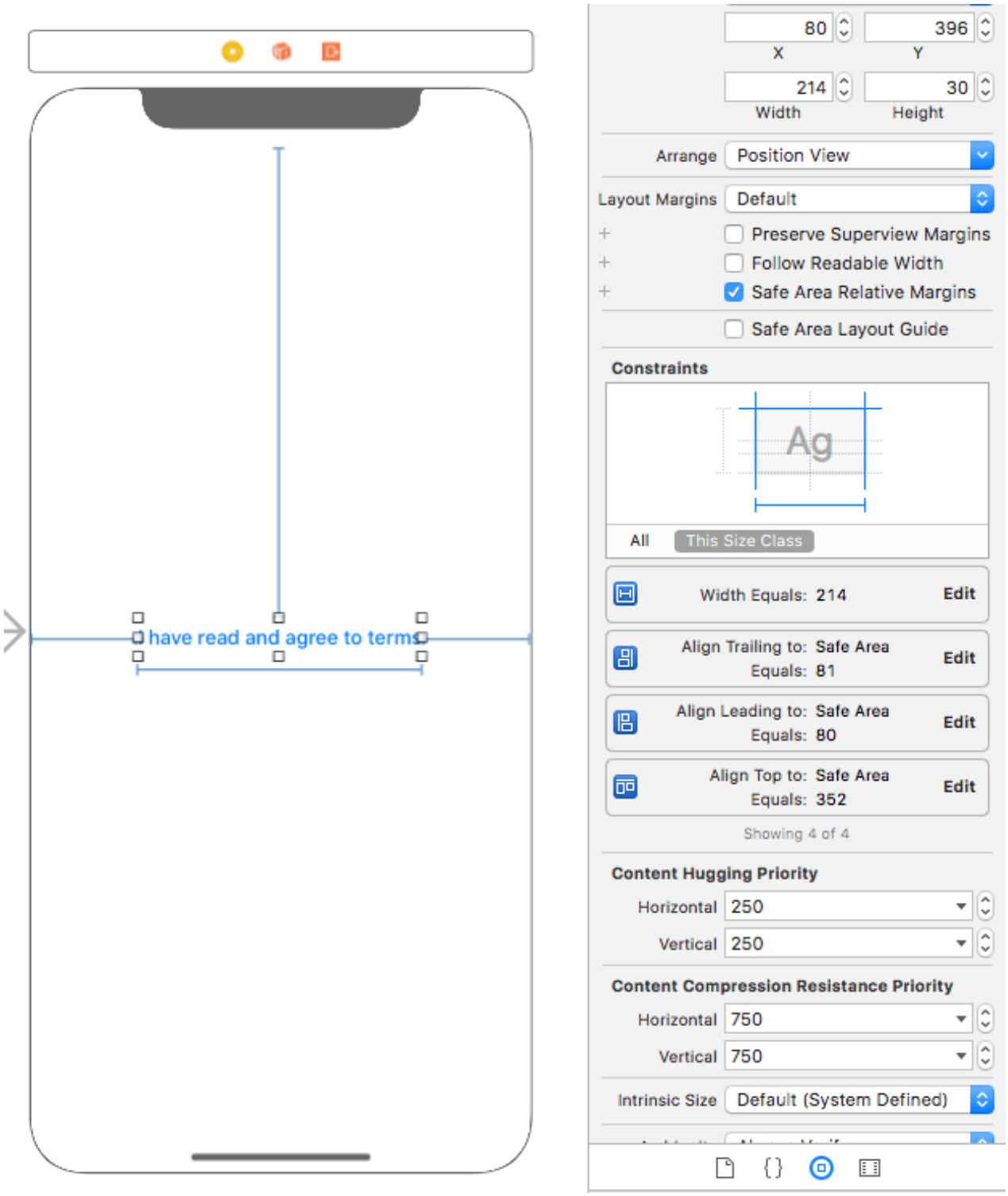
There are three views and all of them have some constraints missing, your task is to **add those missing constraints** so that they look like this in iPhone SE, iPhone 8 and iPhone 8 plus :

Once you are done, you can check the answer in **Answer.storyboard** ! It is fine if your constraint is not exactly the same as the answer, as long as the three views look like above in different phone screen (and also no red lines) then it is correct.

5 - Why conflicting constraints happen and how to solve them

Ever experience that you finally got rid of all the red lines and it looks good on your device, but when you view your app on another device in simulator, it looks totally off?

This is probably because you have conflicting constraints in the layout, we will use a simple example to explain below. This example contain a vertically centered button :

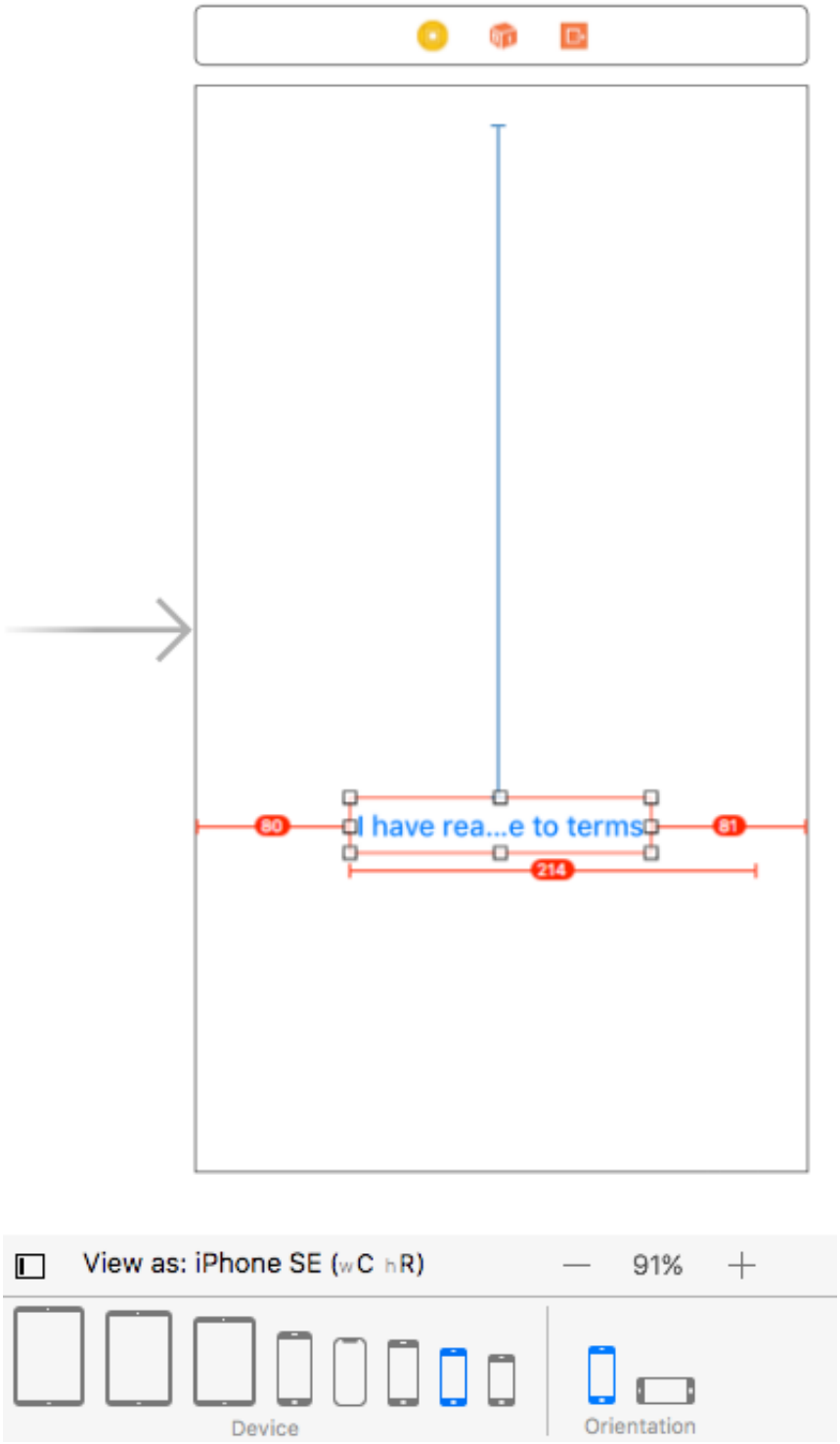


4 constraints are defined:

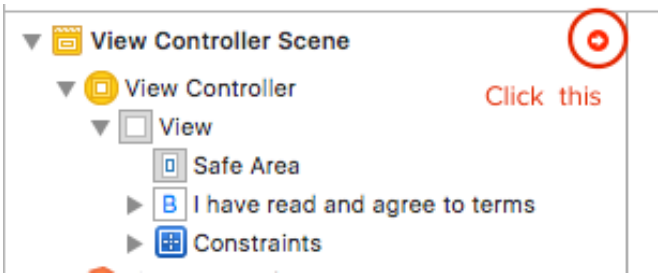
1. Width of the button equals 214 pt
2. Distance from the screen left to the left of the button is 80 pt
3. Distance from the screen right to the right of the button is 81 pt
4. Distance from the screen top to the top of the button is 352 pt

It looks good and all the lines are blue, yay! It seems like sufficient constraints are defined, Auto Layout is able to calculate the x position, y position, width and height of the button.

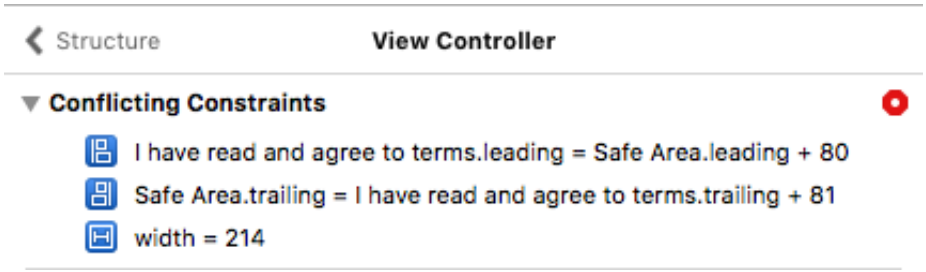
It looks good on iPhone X, but once you switch the view to iPhone SE, there's red lines everywhere with numbers 😱 (this means that there's conflicting constraints), and the button text got squeezed :



Similar to solving missing constraint, we can troubleshoot conflicting constraints by clicking the white arrow inside red circle button :



And we can see that Xcode complains to us that there's conflicting constraints :



Hmmm... so what's with the conflicting constraints? Remember how Auto Layout calculate size of a view? (Refer to chapter 2 - how auto layout calculates position and size) We will first discuss why the layout works fine on iPhone X.

Why it works fine on iPhone X

iPhone X has a screen width of 375 point,

From the leading and trailing constraints, Auto Layout can deduce the width of the button with this calculation :

```
// iPhone X
ScreenWidth = 375

LeadingConstraint = 80
TrailingConstraint = 81

widthOfButton = ScreenWidth - LeadingConstraint - TrailingConstraint
widthOfButton = 375 - 80 - 81
widthOfButton =  214
```

Using the leading and trailing constraints, Auto Layout calculates that the width of button equals to 214.

But we have also explicitly defined the width of the button to 214 using the width constraint.

```
// explicitly defined width constraint
widthOfButton = 214
```

Since the explicitly defined width is equal to the width derived from using leading and trailing constraints, Xcode doesn't complain and Auto Layout happily lay it out with no fuss.

Why it doesn't work on iPhone SE (and also other screen width)

iPhone SE has a screen width of 320 point,

Using back the same calculation with leading and trailing constraints but with different screen width :

```
// iPhone SE
ScreenWidth = 320

LeadingConstraint = 80
TrailingConstraint = 81

widthOfButton = ScreenWidth - LeadingConstraint - TrailingConstraint
widthOfButton = 320 - 80 - 81
widthOfButton =  159
```

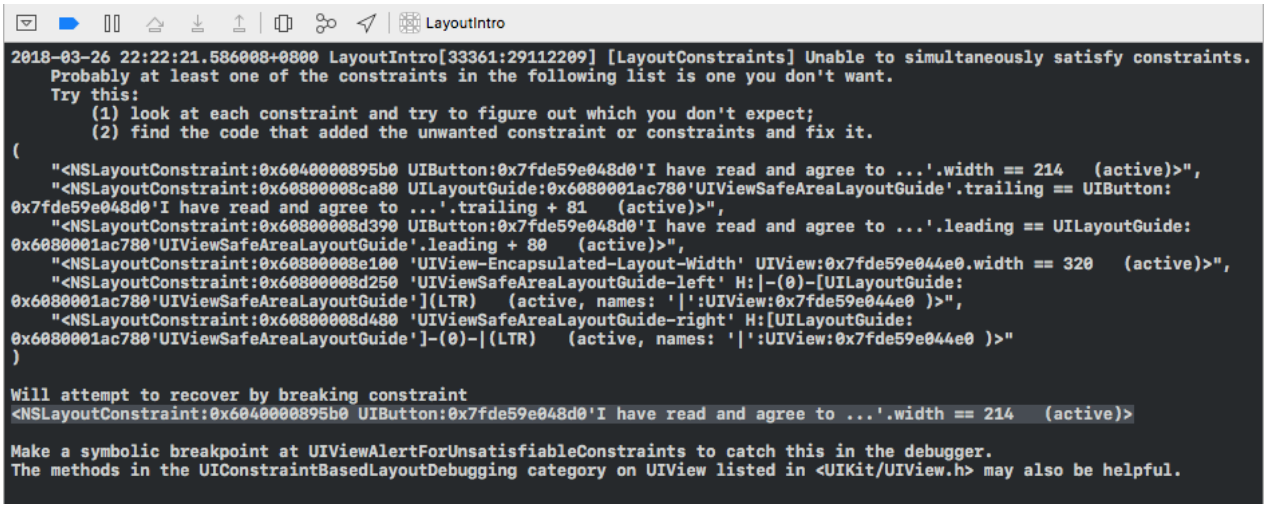
Using the leading and trailing constraints, Auto Layout calculates that the width of button equals to 159.

But we also have another width constraint which explicitly defined the width of button to 214.

```
// explicitly defined width constraint
widthOfButton = 214
```

Since there is two possible different width value, Xcode / Auto Layout got confused ("Should I use 159 as width or 214? they are conflicting !") thus showing you red lines with numbers.

When you run the app with conflicting constraints, you will see some error message in the console like this :



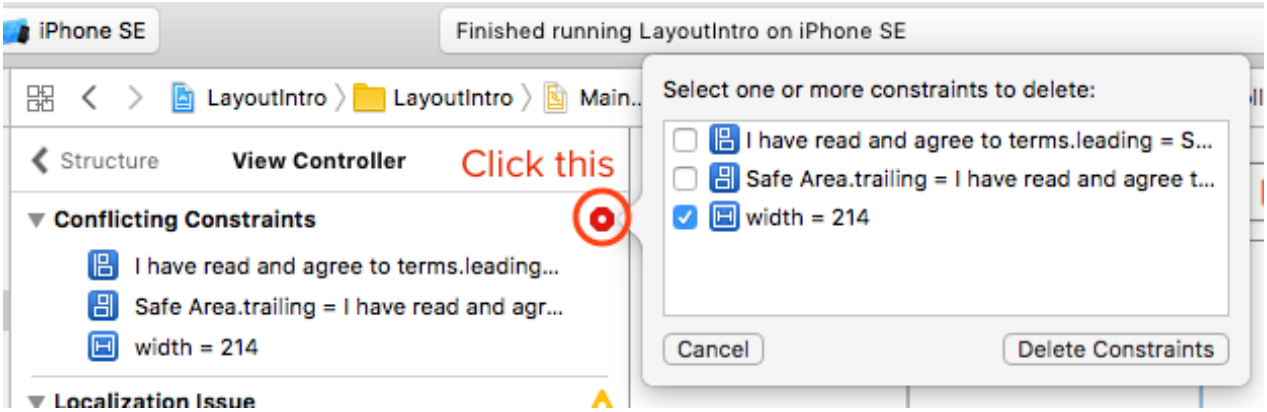
Auto Layout can't satisfy all the constraints as it is impossible to set the width of the button to both 159 and 214 at the same time. Thus to render the button on screen, Auto Layout will break (ignore) one of the constraints. (The order which Auto Layout choose a constraint to break isn't random, but I don't really know its order 😅)

In this case, the explicit width constraint of 214 will be ignored. After breaking (ignoring) the width 214 constraint, Auto Layout will use 159 as width of the button (from the calculation of leading and trailing constraints).

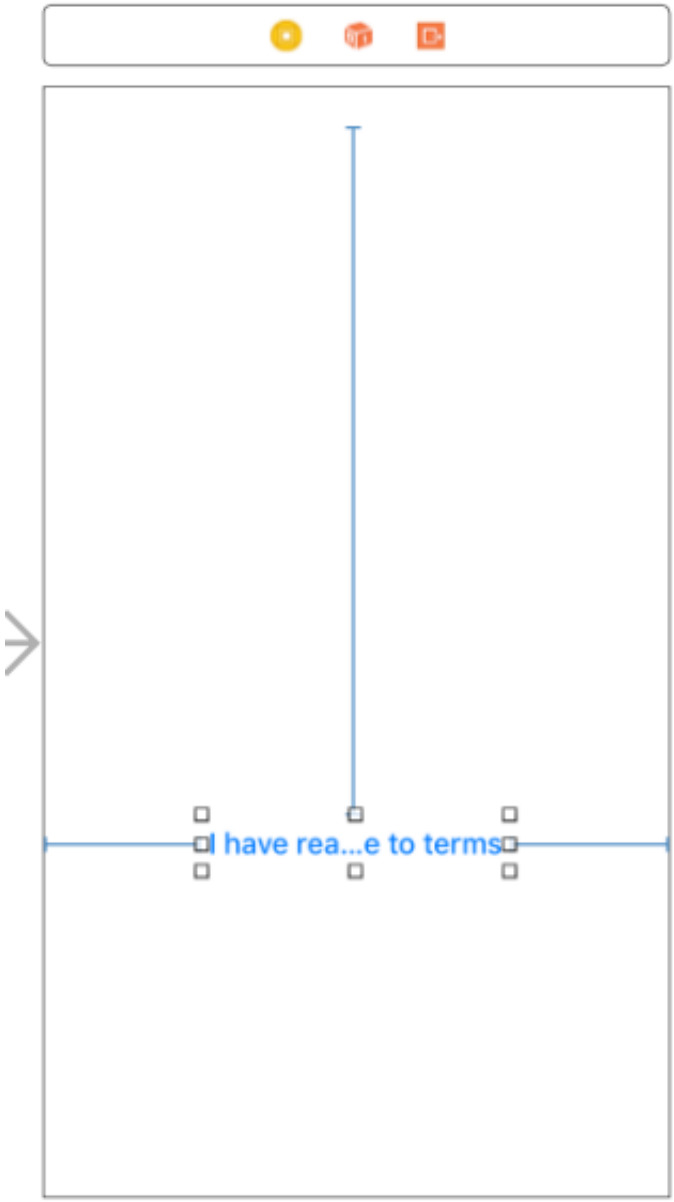
When there are multiple conflicting constraints while the app is running, Auto Layout will attempt to break constraints one by one with a certain order until there is no conflict in the layout calculation (size and position) then only will proceed to render the layout.

Fix

To resolve conflicting constraint, click on the red icon with white dot and select the constraint to delete. In this example, we will delete the explicit width 214 constraint so that the width of the button can be dynamic depending on the screen width.



After deleting the conflicting constraint, Xcode doesn't confuse anymore and show us blue lines on the iPhone SE screen (and other screen size too) , yay!



The text still got squeezed because the width derived from calculation is too short, we will need to reduce the value of leading constraint and trailing constraint. This is pretty straightforward so I will skip it now.

Summary

Conflicting constraints usually happen when you have defined too many constraint on an UI element causing Auto Layout unable to satisfy all of them.

As a general rule of thumb, try to **use as least constraints as possible** when designing layout, to reduce the chance of creating conflicting constraint and also make it easier to manage later on.

If you design UI with only one device size (usually the iPhone model you are using) in the Interface Builder during development, you might be prone to creating conflicting constraints unknowingly (blue lines in the device you selected, but might have red lines in other device).

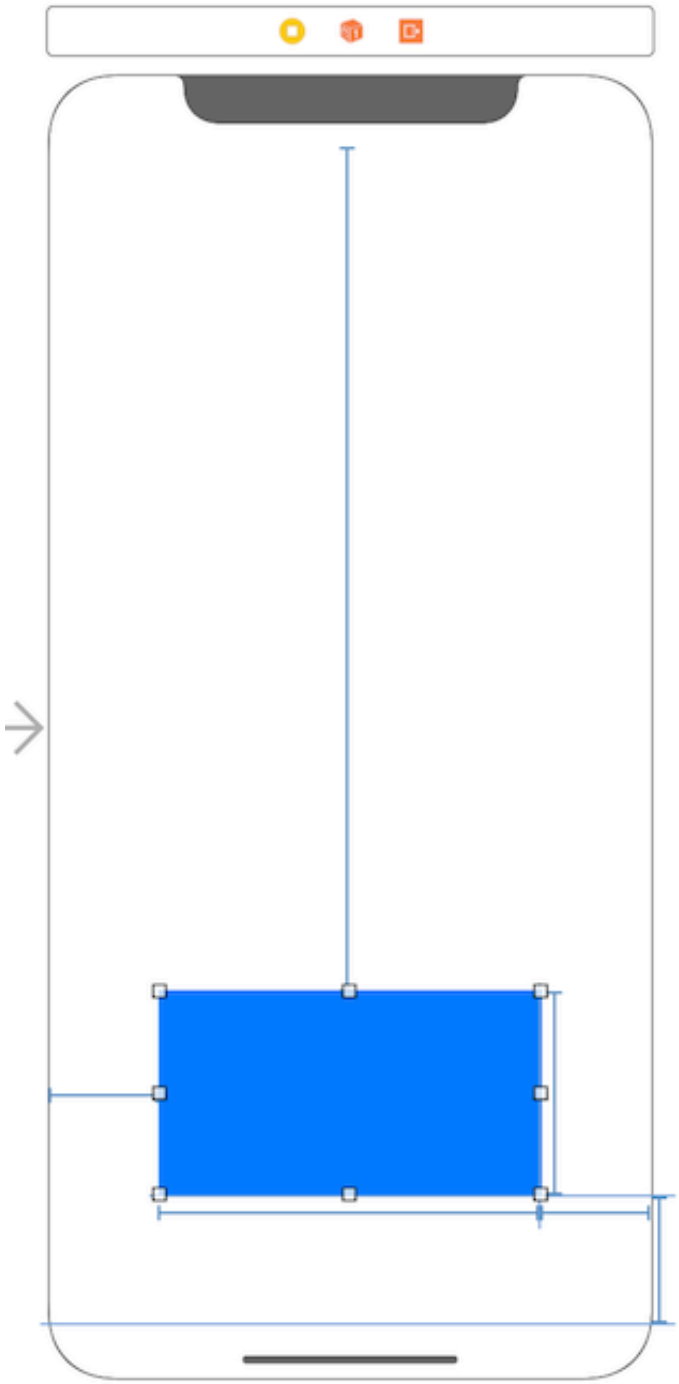
To spot accidental conflicting constraint, be sure to switch between different iPhone / iPad model when placing a new constraint/ deleting a constraint in the Interface Builder.

You can switch the device using the device list in interface builder:

Exercise

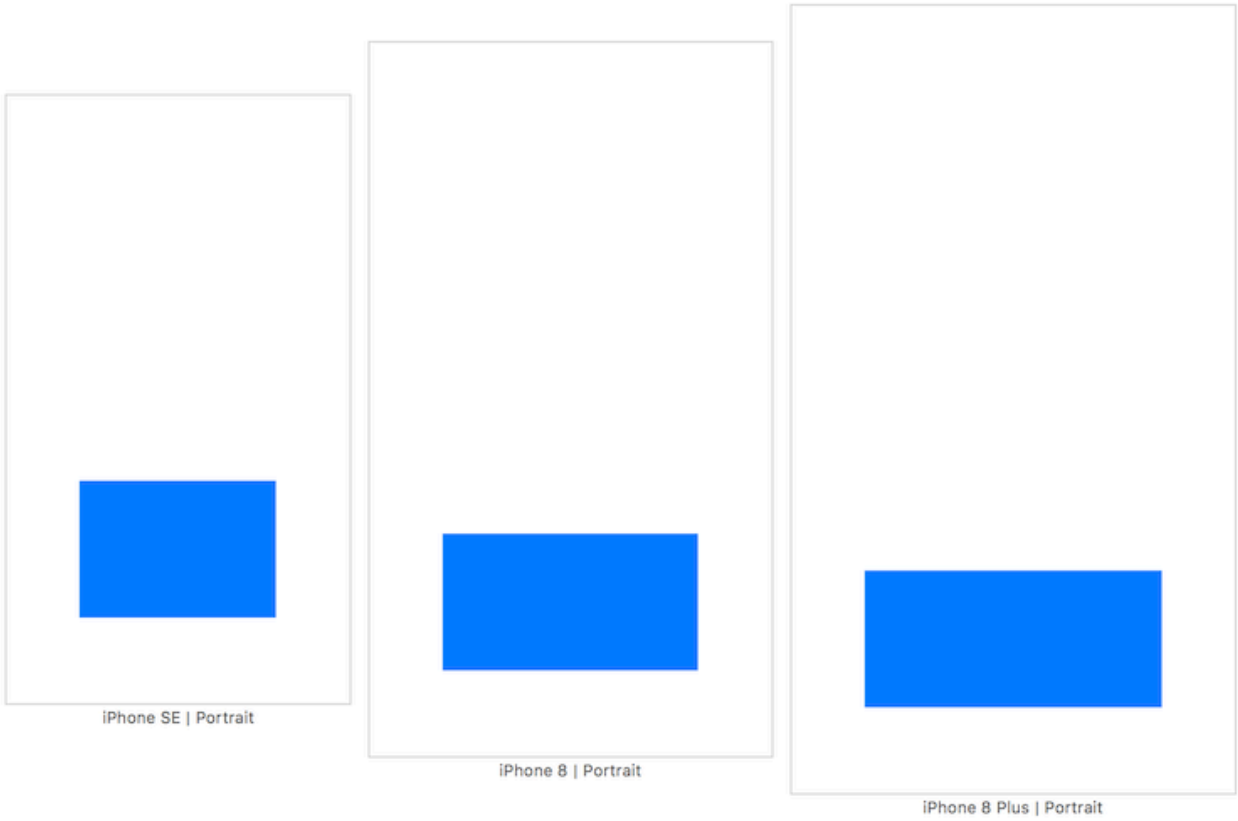
There is an exercise Xcode project for this chapter, it is located in `exercises/conflictingConstraint/conflictingConstraint.xcodeproj` .

The project contains two storyboards, **Main.storyboard** and **Answer.storyboard**. Open the **Main.storyboard** and you will see a view controller with three views like this :



There is one blue view in the view controller. At first glance everything seems good with blue lines and all, but there's multiple conflicting constraints group for the blue view, if you switch the device to iPhone 8 plus or SE, you will see plenty of red lines.

Your task is to choose and **remove the constraints that cause conflict** so that it will look like this in iPhone SE, iPhone 8 and iPhone 8 plus :



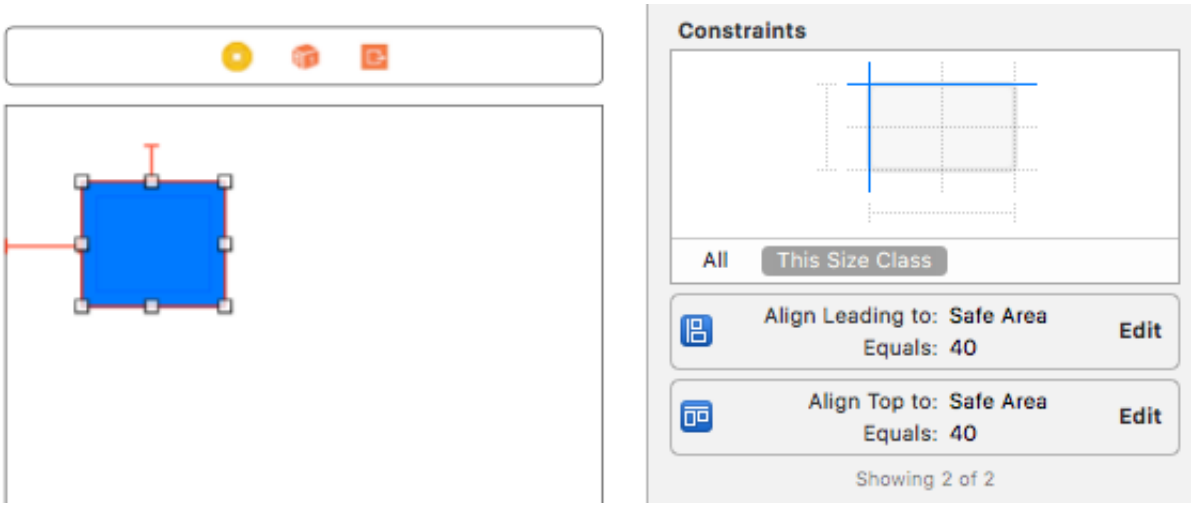
Once you are done, you can check the answer in **Answer.storyboard**. Be sure to select the correct constraint to remove so that the end result looks like above in different phone screen size!

6 - What is intrinsic content size and the importance of it

As a recap, in order for Auto Layout to render the position and size of a view correctly, Auto Layout **must know** or **able to calculate** the :

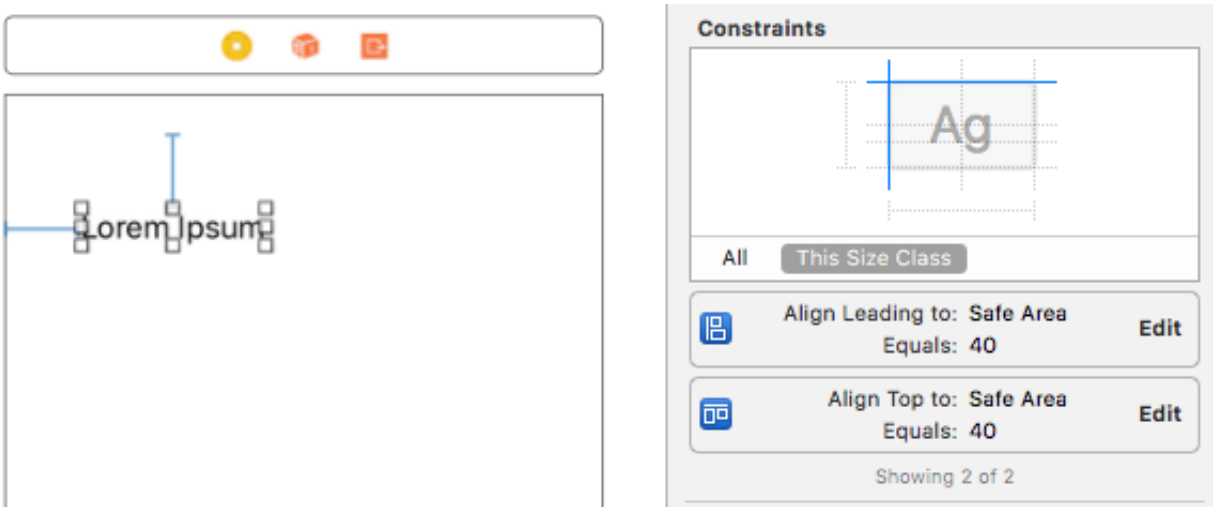
1. **x position** of a view
2. **y position** of a view
3. **width** of a view
4. **height** of a view

If you place a view **without** enough constraint that Auto Layout is unable to calculate its width and height, Xcode would complain by showing you red lines like this :



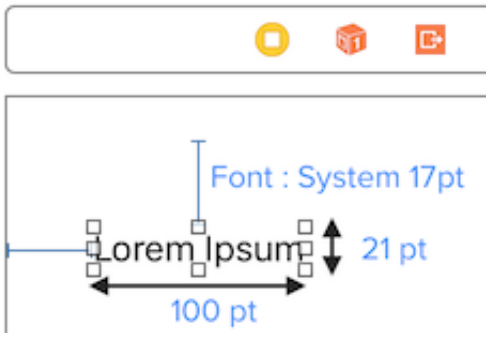
The blue rectangle view has a leading (40 pt) and top (40 pt) constraints, meaning its x position and y position is known, but Auto Layout doesn't know its width and height. Hence Xcode is showing you the red lines.

Now lets place a label and define the same leading and top constraints :



Wait... Xcode doesn't complain even though the size is not defined and it show us blue line, what happened? It turns out that Auto Layout is able to calculate the label size (width and height) based on its content (text of the label) and the font type / size of the label.

For example, a label that is using System font type, font size 17pt and contain text "Lorem Ipsum", its intrinsic size is as follows :



If you increase the font size of the label or insert more text to the label, its intrinsic size will become bigger. Conversely, if you decrease the font size of the label or shorten the text in the label, its intrinsic size will become smaller.

Intrinsic content size means the size of an UI element **derived based on its content**. For label, its content is the text inside.

UIView doesn't have intrinsic content size, there is no way to calculate its width and height using the content inside UIView (there's no text inside UIView!), hence you have to manually define constraints that enable Auto Layout to calculate its size.

Label / UILabel has intrinsic content size, its width and height can be derived from the text of the label and the font type / size used.

Button and Textfield have intrinsic content size and its calculation is similar to label.

An empty image view doesn't have intrinsic content size, but once you add an image to the image view, its intrinsic content size is set to the image size.

Here is a table of intrinsic content size for common UI element from Apple official documentation.

Intrinsic content size of Label

By default, the intrinsic content width of a label increases as the text gets longer but its intrinsic content height stays fixed as single line tall. If you have a label containing long text without constraint specifying its width and height, it will grow horizontally and eventually off the screen like this :

To prevent the label from going off the screen, we can specify constraint that enable Auto Layout to calculate its width and this value will replace its default intrinsic content width. Let's add a trailing constraint to the label and it will look like this :

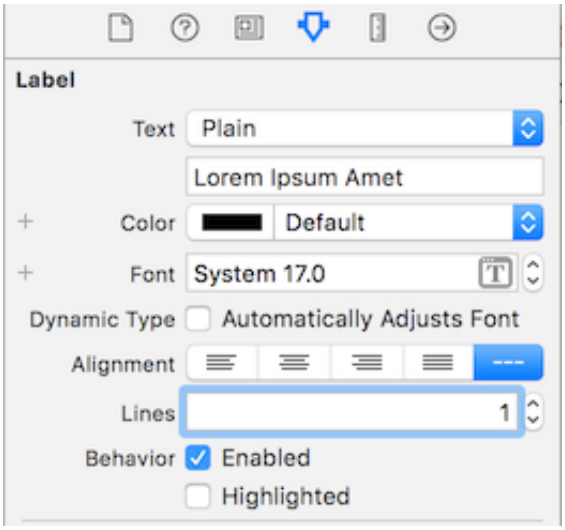
Note: Explicit width/height constraint or combination of constraints (eg: leading + trailing constraint) that allows Auto Layout to calculate width / height will override and replace the intrinsic content size of an UI element.

For iPhone SE, Auto Layout will calculate the width for the label shown above like this :

```
// iPhone SE
screenWidth = 320
leadingConstraint = 40
trailingConstraint = 40

labelWidth = screenWidth - leadingConstraint - trailingConstraint
labelWidth = 320 - 40 - 40
labelWidth = 240
```

As Auto Layout is able to calculate the label width using constraints, it will use this calculated value instead of its original intrinsic width. But now the text is truncated as its intrinsic height is still 1 line tall, how can we change its intrinsic height?

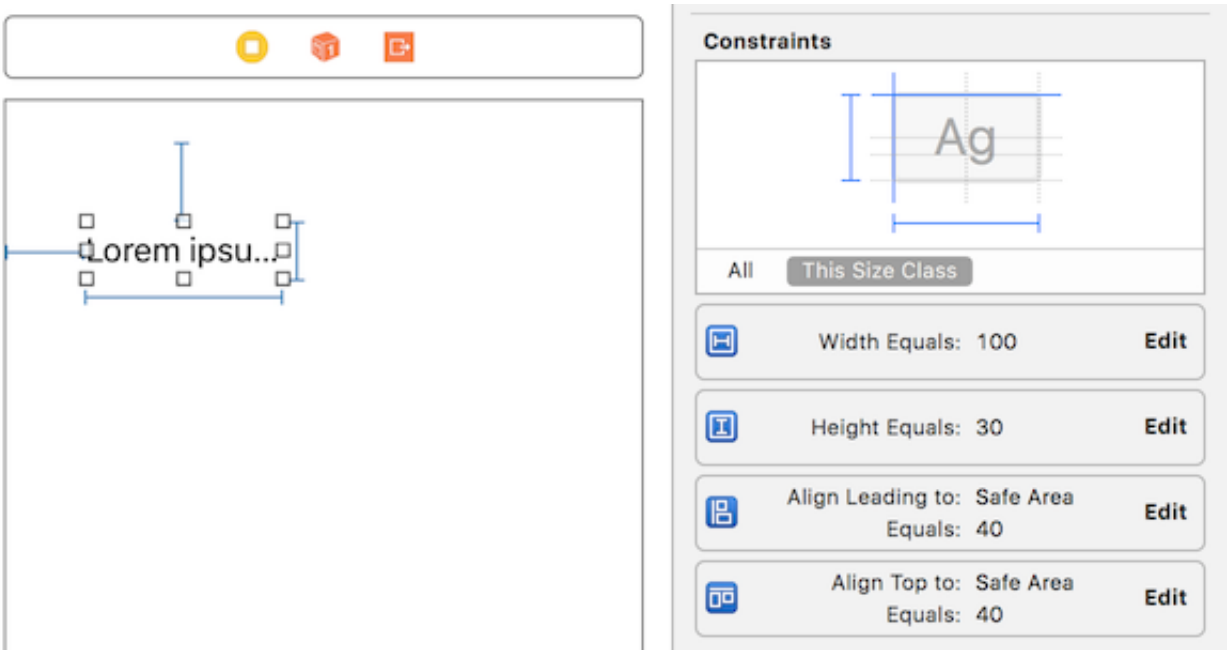
We can change its intrinsic height by changing the number of lines. In **the attributes inspector**, select 'Lines'. This number means the **maximum number of lines** the label height will expand to. If your label text occupies two lines and you set the line numbers to 2, it will show 2 lines. If your label text only occupies 1 line and you set the line numbers to 2, it will just show only 1 line, which your text occupied, it won't show extra blank line space below.



There's a special case for this, if the **number of lines of the label is 0**, it means that the label can **infinitely** expand in height as its content text grows longer. We can change the number of lines and update the frame of the label like this:

(Check the included video in `videos/6/changeNumberOfLinesZero.mp4`)

If you set both explicit width and height constraint, or combination of leading/trailing and top/bottom constraint for the label, the explicit defined size or calculated size from constraints will replace the intrinsic content size making it fixed and non flexible. This causes problem if the text of label exceed the size and unable to expand, causing the text to be cropped :



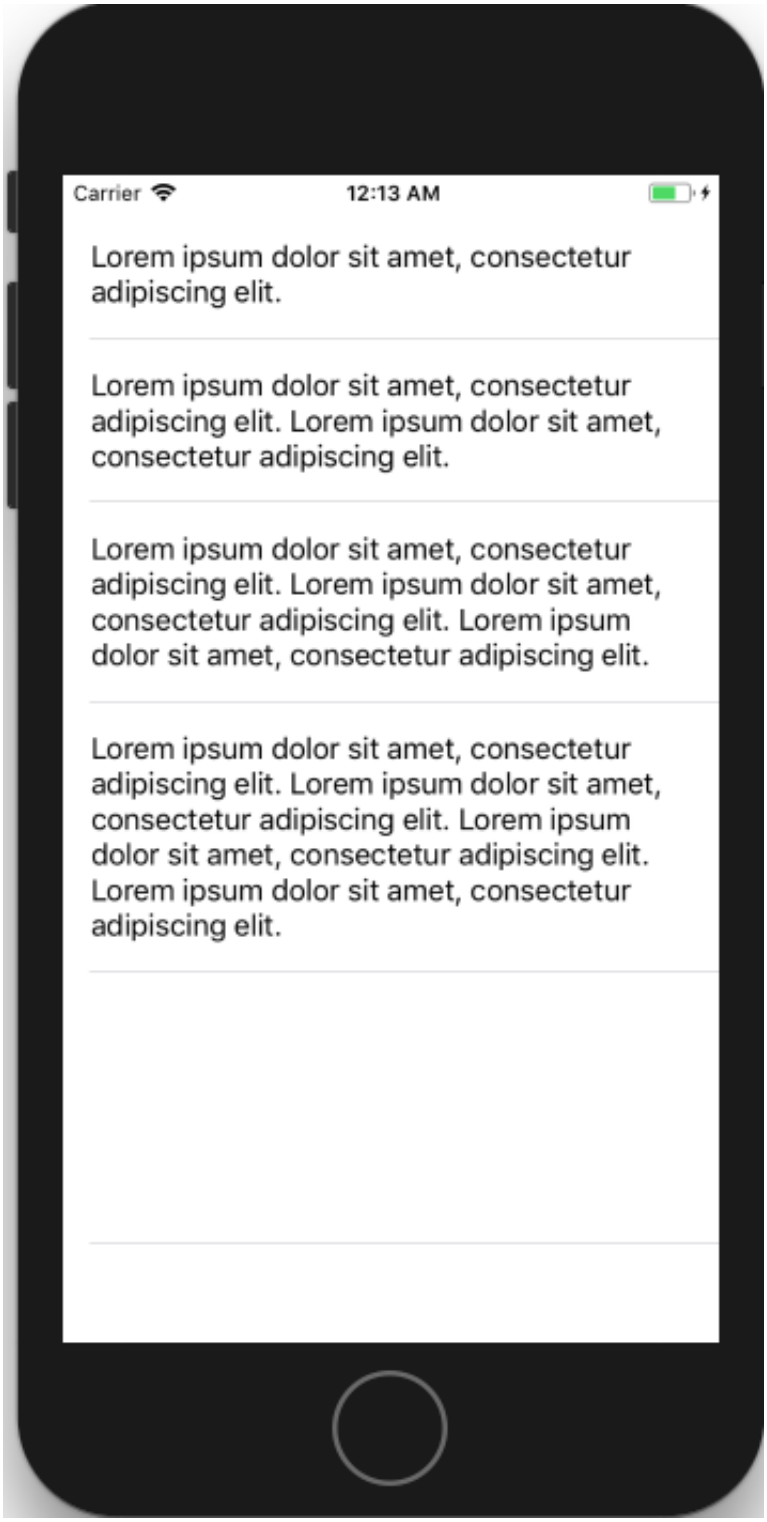
In most cases, you will only need to set constraints to let Auto Layout calculate the width of the label and let its intrinsic height expand based on its text content.

The same concept can be applied on other UI element that have similar intrinsic content size calculation as label such as button and textfield.

Usefulness of intrinsic content size

Intrinsic content size allows the size of an UI element to be dynamically expanded based on its content, it is especially useful on **designing a scroll view containing dynamic content** (content loaded from user input or HTTP request) or **self sizing table view cells** (tableview containing rows of different height based on the content of the label inside cell). We will explain more about scroll view and table view in the upcoming chapters.

Self sizing table view cells look like this :



Notice how the tableview cell expand to cater for longer length of text (content) inside it. Since the width is fixed (`label width = tableview width – leading constraint – trailing constraint`), the text label intrinsic height increases as text got longer. This is useful if you are showing comments with varying length (like Reddit app) on table view.

7 - Using Tableview for content with dynamic size

In previous chapter, we have talked about what is intrinsic content size and how it can dynamically expand based on a label's text content. We will apply the intrinsic content size on a tableview for this chapter.

Create a simple dynamic height table view cell

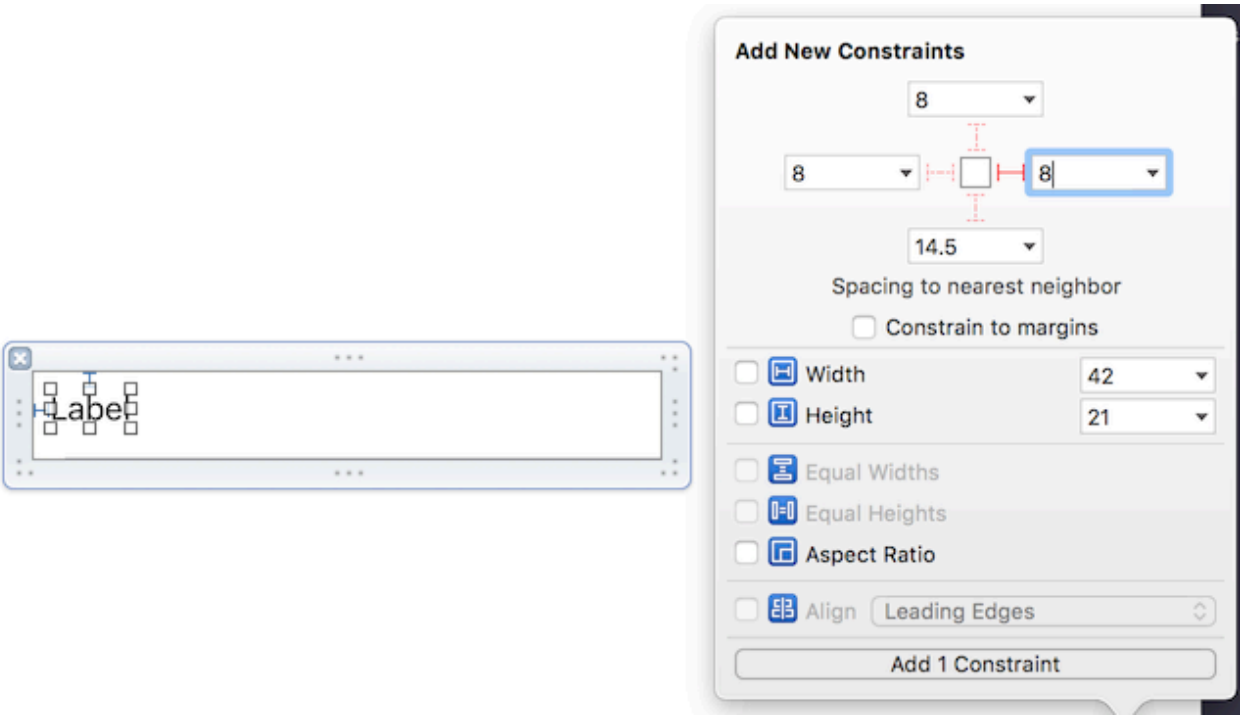
We will use a starter Xcode project with a preset tableview and custom tableview cell for this chapter. You can find the project in `exercise/tableView/tableView.xcodeproj` . When you open and run the project, you should see a tableview with some cell like this:

Notice that the text in second and third row is too long and stretched across the screen? This is because the tableview cells have a fixed height. We are going to fix that.

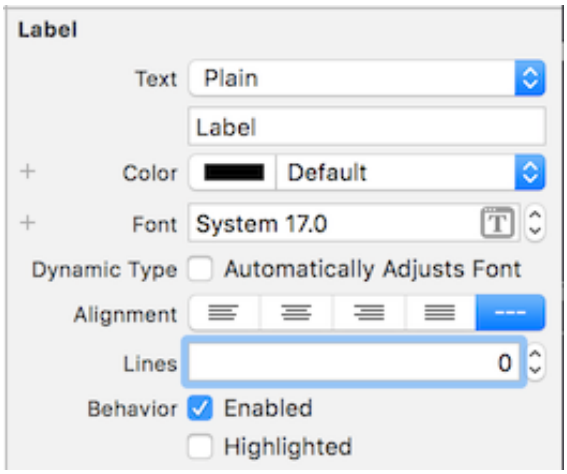
Open `CommentTableViewCell.xib` (Inside the 'View' group) , you will see there's a label with top and leading constraint defined:

CommentTableViewCell.xib

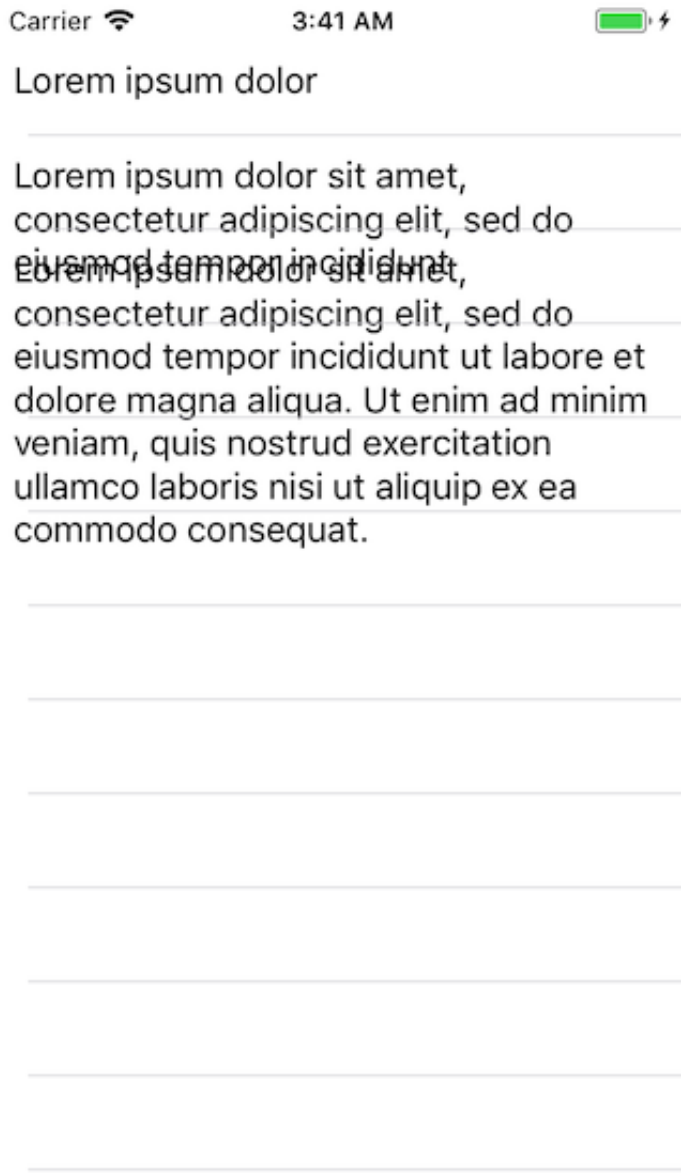
There is no trailing constraint for the label hence it will keep growing to the right (eventually off the screen) as the text gets longer, let's set a trailing constraint to it. Let's set its value to 8, and remember to uncheck the "Constraint to margins".



We will also set its number of lines to 0 so it can expand downward infinitely as its text increase:



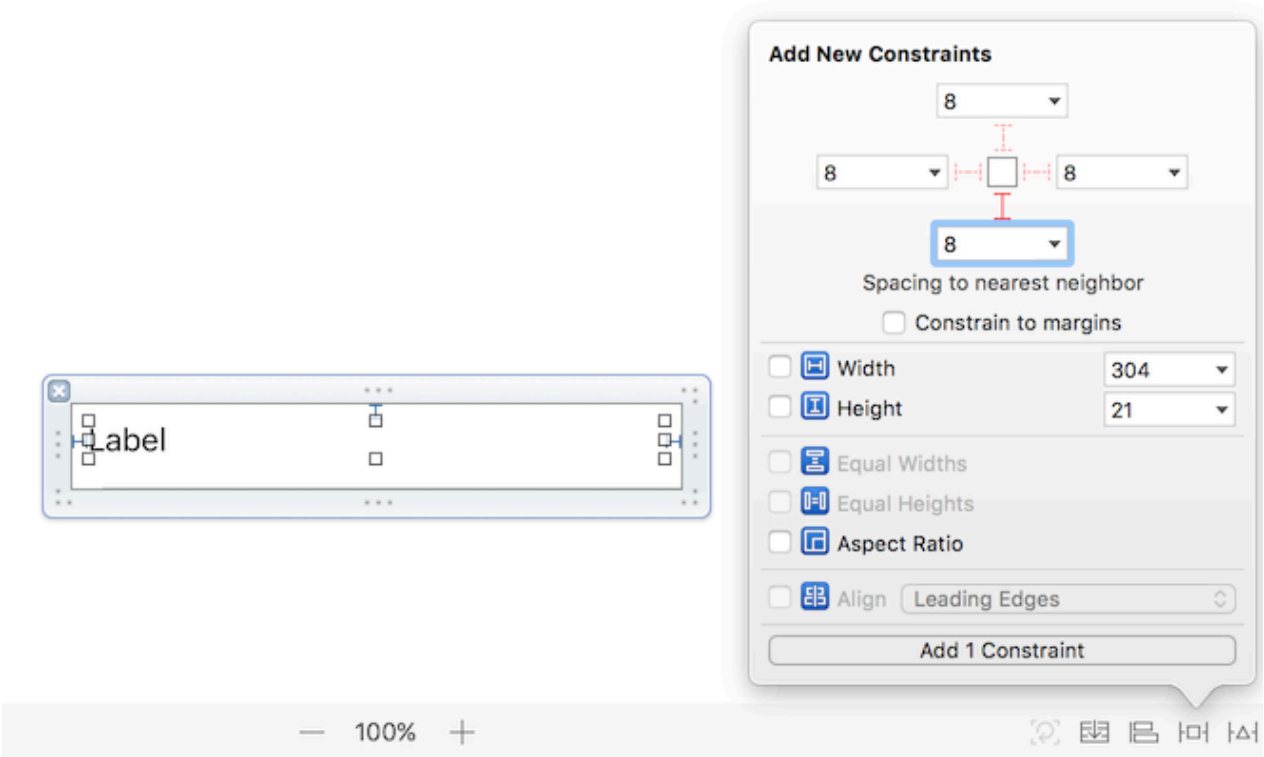
Let's run it again and see the result. You should see a result like this:



The labels are getting overlapped! 😱 The label text on second row has extended and overlapped with the text on third row! This is happening because although the label has become dynamic, the cell height still remain fixed for each row.

We will now make the cell expand its height based on the intrinsic content height of the label.

Add a bottom constraint to the label in `CommentTableViewCell.xib`, set its value to 8 and remember to uncheck "Constraint to margins":



This will ensure that the label never cross the bottom boundary of the cell and overlaps the next cell, we will further explain it later.

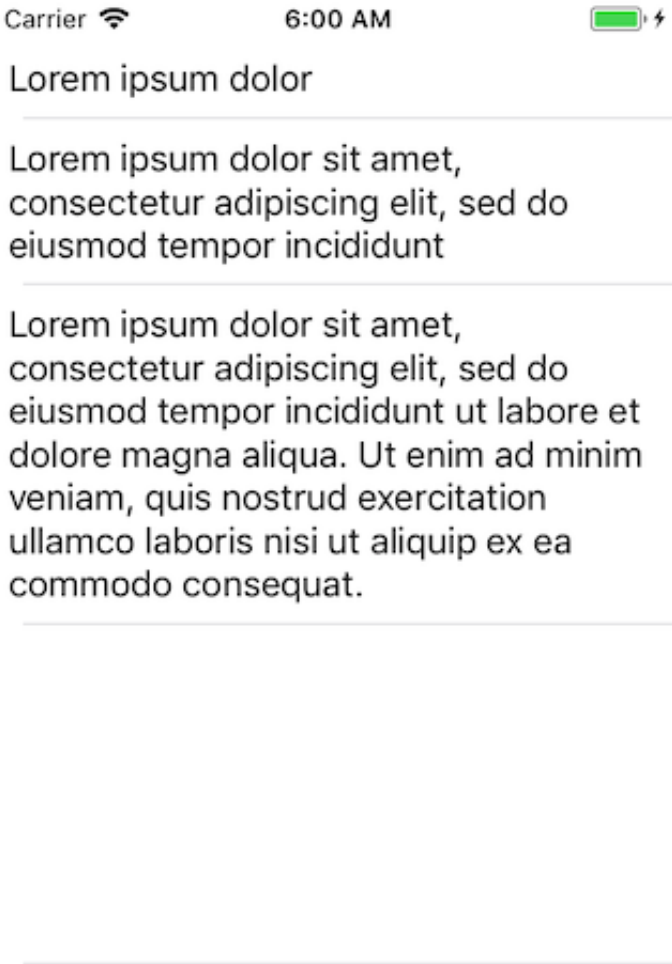
And then open `ViewController.swift`. Find the line `self.tableView.rowHeight = 44.0` inside `viewDidLoad()`, replace the fixed height value `44.0` to `UITableViewAutomaticDimension`. This tells the tableview that its row height will be dynamic depending on the size derived from the content inside.

```
// ViewController.swift
// Inform tableview to dynamically calculate height for each row
self.tableView.rowHeight = UITableViewAutomaticDimension
```

We will also add `self.tableView.estimatedRowHeight = 44.0` on the next line. The `estimatedRowHeight` is the placeholder height value for the cell before the layout/text is loaded in it. This is useful when you are fetching text data from web API and before the text data response is received, there's a default height for the cell. Once the text data response is received and put into the label, tableview will automatically adjust the height for the row.

```
// ViewController.swift
// Inform tableview to dynamically calculate height for each row
self.tableView.rowHeight = UITableViewAutomaticDimension
self.tableView.estimatedRowHeight = 44.0
```

Now build and run the app again, you should see each row has a different height depending on its text content:



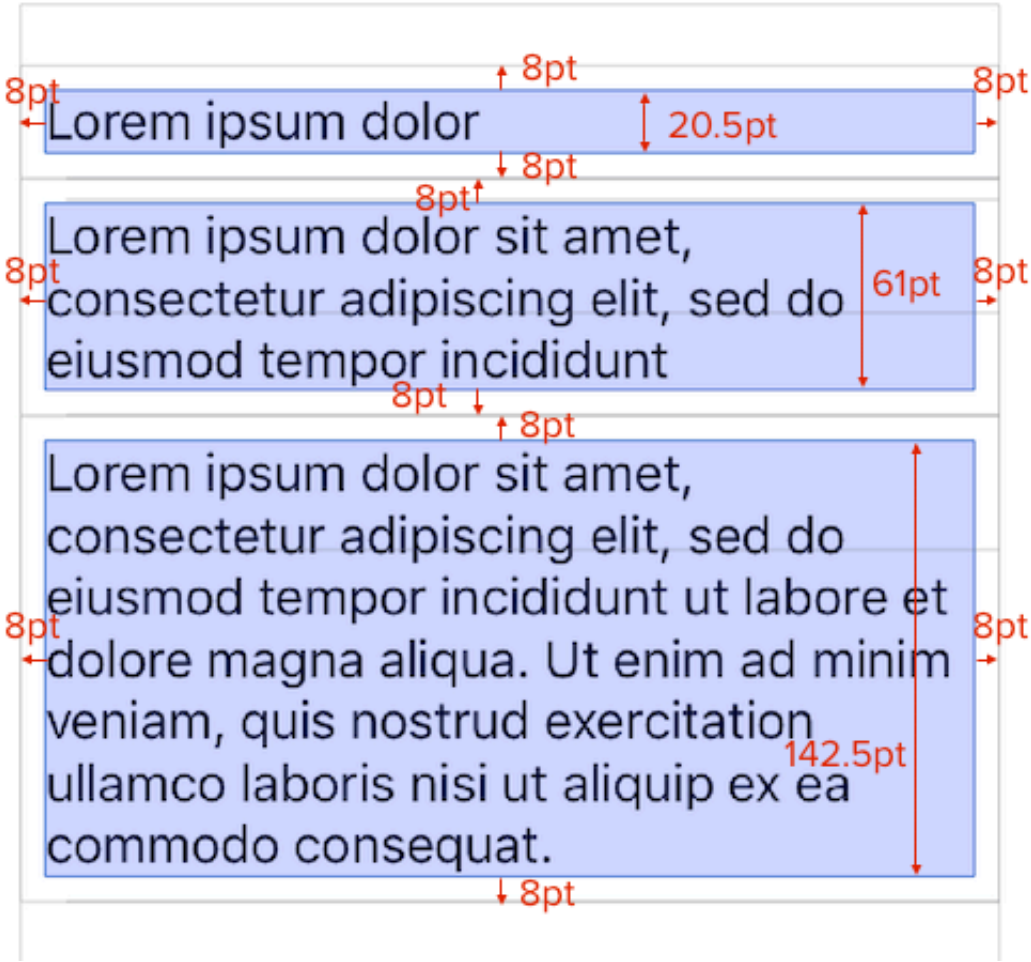
Yay! Now we have a tableview with dynamic cell height (also known as "**self-sizing tableview cell**", use this phrase if you want to google thing related to it). We will explain more about how the tableview calculates the cell height using auto layout constraint below.

Notes from Apple official guide regarding self-sizing tableview cell:

> To enable self-sizing table view cells, you **must set** the table view's `rowHeight` property to `UITableViewAutomaticDimension`. You **must also assign a value** to the `estimatedRowHeight` property. As soon as both of these properties are set, the system uses Auto Layout to calculate the row's actual height.

Calculation of cell height

Here is the constraint and intrinsic content height of label in each cells:



By using the auto layout calculation formula we have learnt previously, the label width derived from leading and trailing constraint will replace its original intrinsic content width.

```
// iPhone SE
tableviewWidth = screenWidth = 320

labelWidth = tableviewWidth - leadingConstraint - trailingConstraint
labelWidth = 320 - 8 - 8
labelWidth = 304
```

Then we let the label to infinitely expand its intrinsic content height based on its text content by setting its "Lines" property to 0.

The row height is calculated from its inner elements. In this case, the label's intrinsic content height and top / bottom constraint is used to calculate the row height.

```
topConstraint = 8
bottomConstraint = 8

firstRowHeight = firstLabelIntrinsicHeight + topConstraint + bottomConstraint
firstRowHeight = 20.5 + 8 + 8
firstRowHeight = 36.5

secondRowHeight = secondLabelIntrinsicHeight + topConstraint +
bottomConstraint
secondRowHeight = 61 + 8 + 8
secondRowHeight = 77

thirdRowHeight = thirdLabelIntrinsicHeight + topConstraint + bottomConstraint
thirdRowHeight = 142.5 + 8 + 8
thirdRowHeight = 158.5
```

You don't need to know the exact value of intrinsic content height of label as its very difficult to calculate by hand, you just need to know that the height of cell will be affected by the intrinsic content height of its inner UI elements.

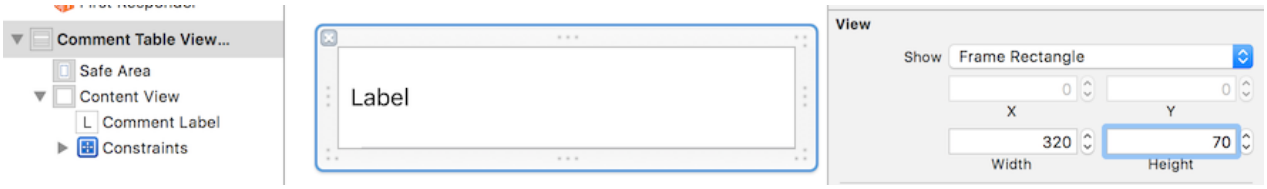
The key of making self-sizing tableview cell work is that **Auto Layout must be able to calculate the height of the cell** using the **constraints** defined and also the **intrinsic content size of UI elements** inside it.

Usually you would define four side of constraints to a label inside a tableview cell, leading/ trailing so that the label wont exceed the left/right boundary, top/bottom so that the label wont exceed the top/bottom boundary.

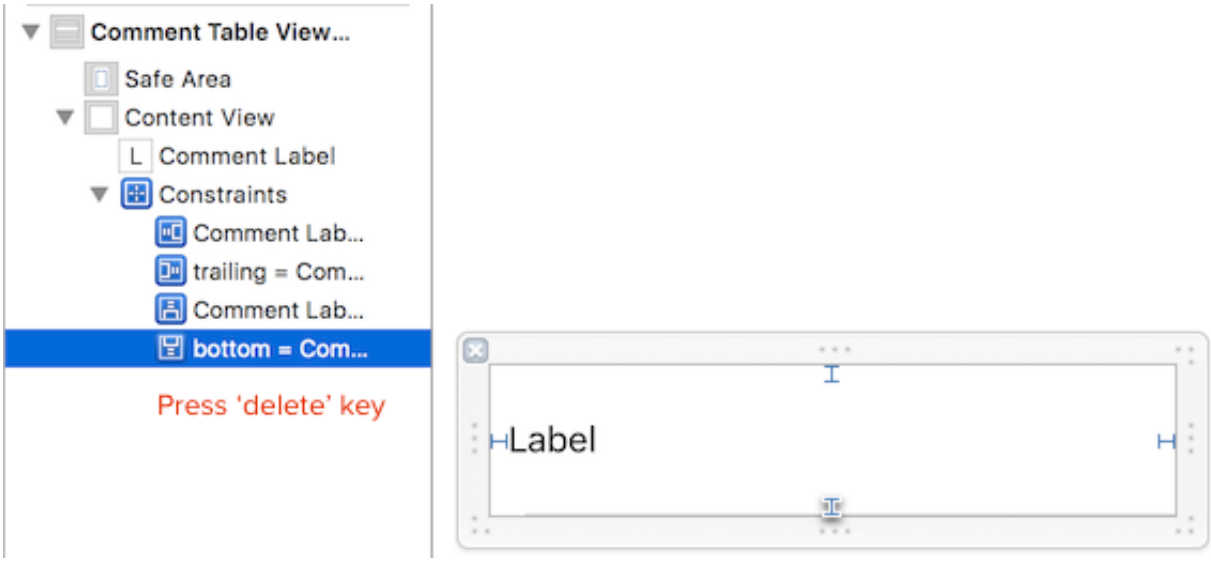
Adding additional UI elements

Let's say now we want to add another label inside the cell to show the username of the commenter.

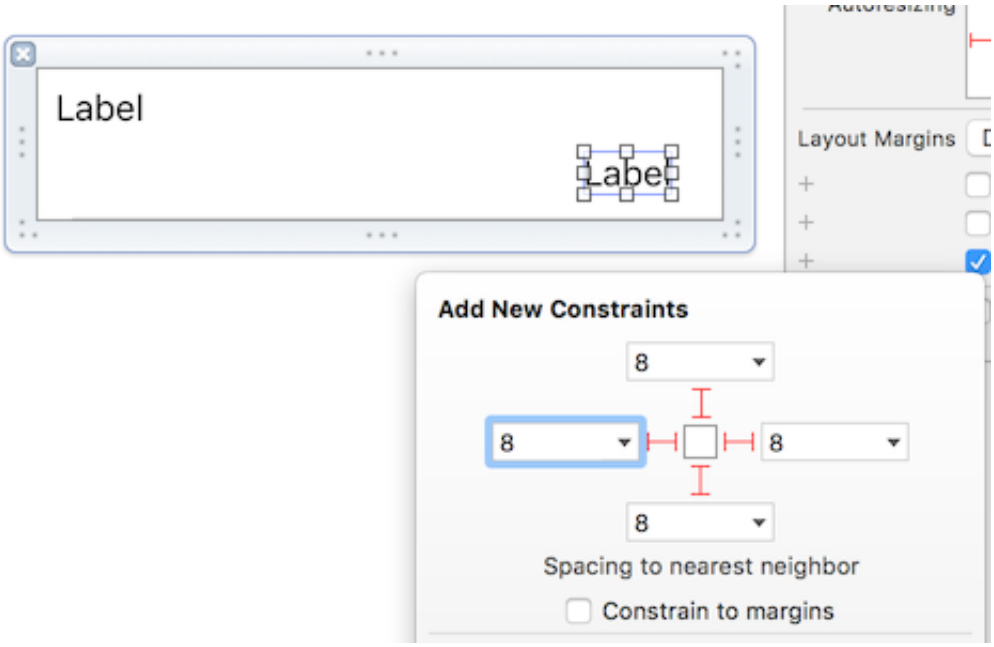
Open `CommentTableViewCell.xib` (Inside the 'View' group) , and increase its height to 70 so we have more space to put another label:



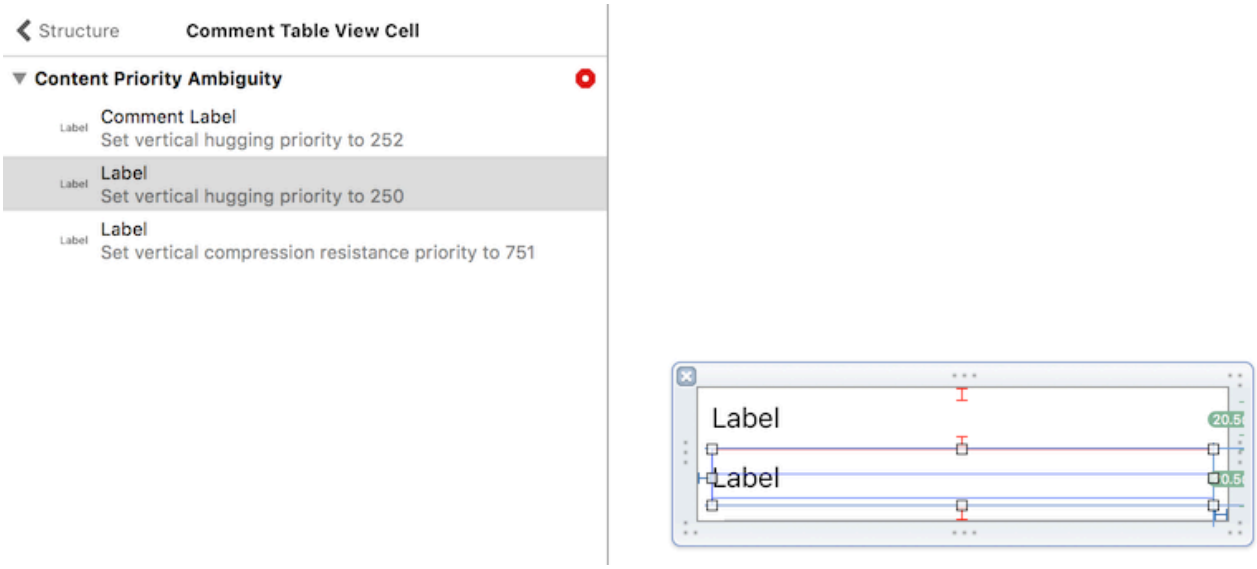
And delete the the bottom constraint of the comment label as we want to put a username label beneath it later.



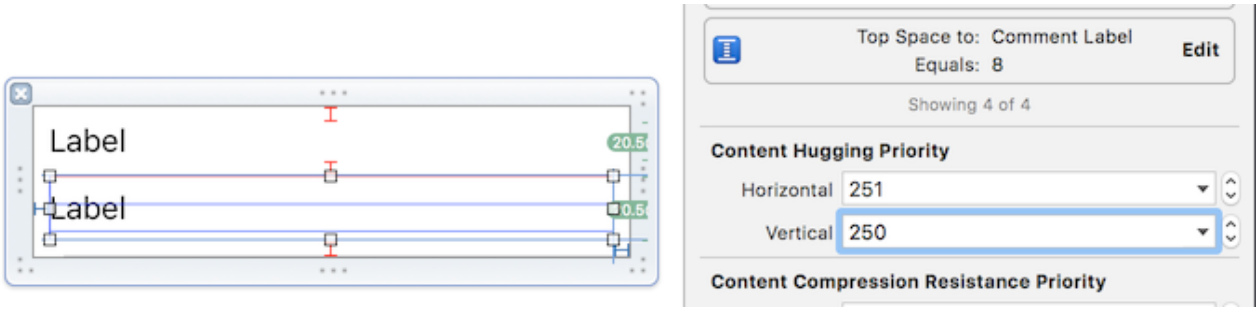
Let's drag another label below the comment label and set its constraint like this (top / leading / trailing / bottom constaint and remember uncheck "Constrain to margins"):



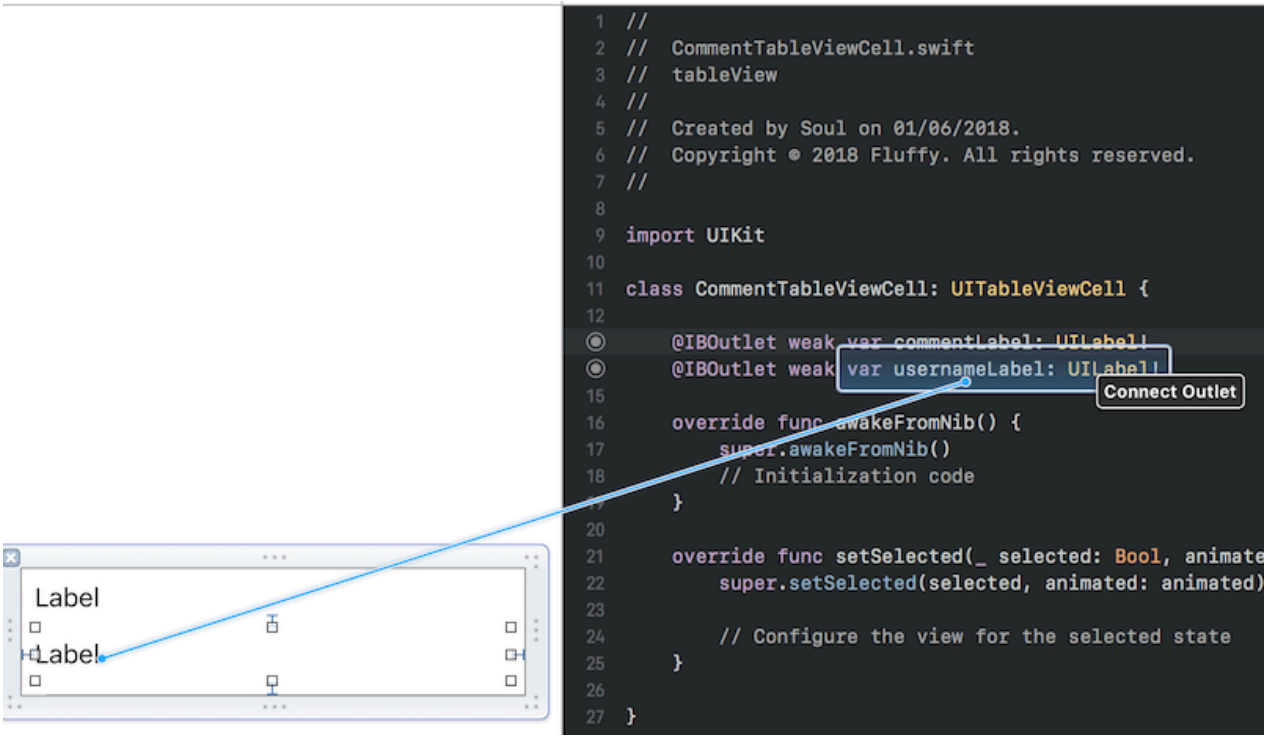
After adding the constraints, you might see an error like this:



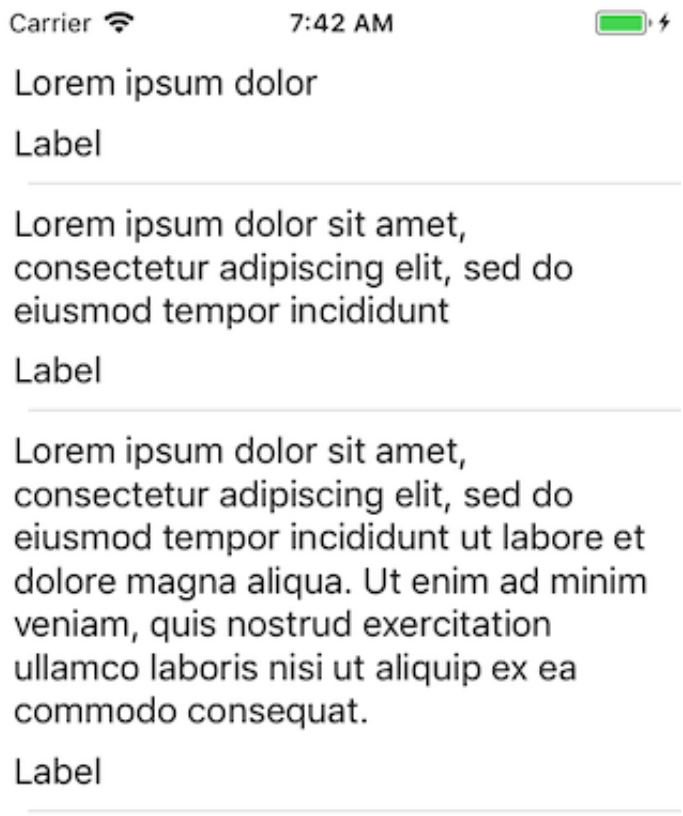
This is because Auto Layout is confused whether to expand the label on top or the label on the bottom to fill the empty spaces between them (the space between top label and bottom label). There will be another chapter explaining this concept further on, for now we will just set the vertical content hugging priority to 250 for the bottom label like this:



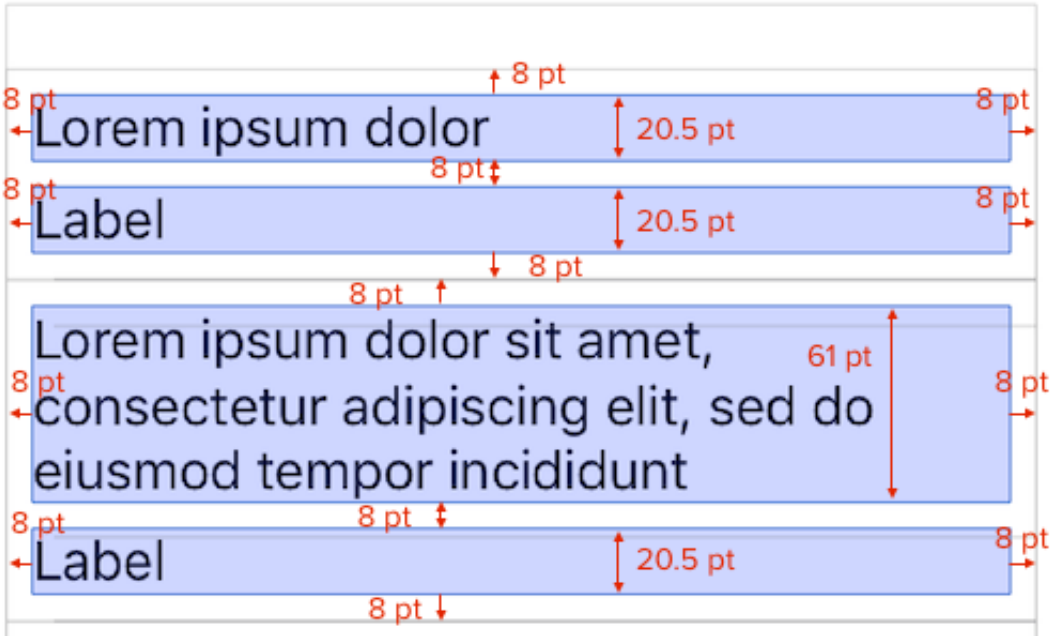
The error should be solved after changing the vertical content hugging priority. Now we will create an outlet for the bottom label in `CommentTableViewCell.swift`, let's name it as **usernameLabel**.



Now let's build and run the app again, you should see something like this:



The cell expanded nicely with the username label added to bottom. This is because Auto Layout is able to calculate the height for each cell. Below is the constraints and intrinsic content height of the labels:



The height for first row can be calculated like this:

```
firstRowHeight = 8 + commentLabelIntrinsicHeight + 8 +
usernameLabelInstrinsicHeight + 8
firstRowHeight = 8 + 20.5 + 8 + 20.5 + 8
firstRowHeight = 65
```

Similar calculation can be used on second row as well.

As long as Auto Layout is able to calculate the height of the cell using constraints and intrinsic content sizes, self-sizing table cell will be displayed nicely.

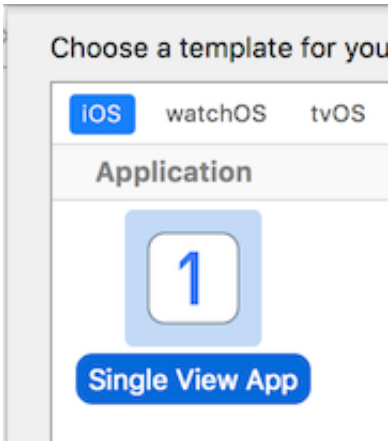
8 - Using Scrollview for content with dynamic size

This chapter will use similar height calculation as explained in chapter 7 (using tableview for content with dynamic size). I would advise to read that chapter first if you have skipped it previously 😅.

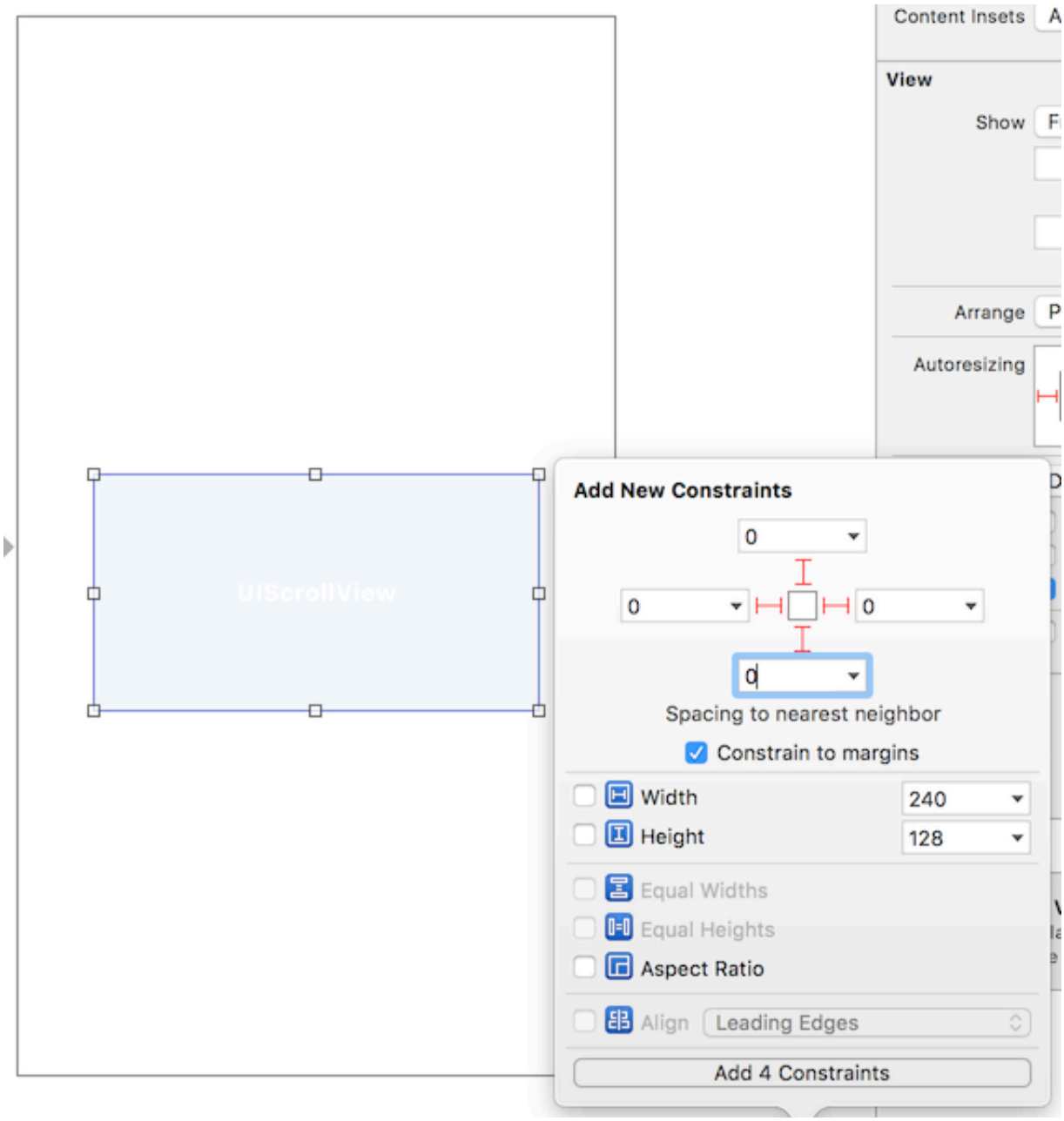
Create a scrollview with content inside

We will create a scrollview which contain dynamic label and picture in this chapter.

Create a new Xcode Project, select "**iOS**" > "**Single View App**".

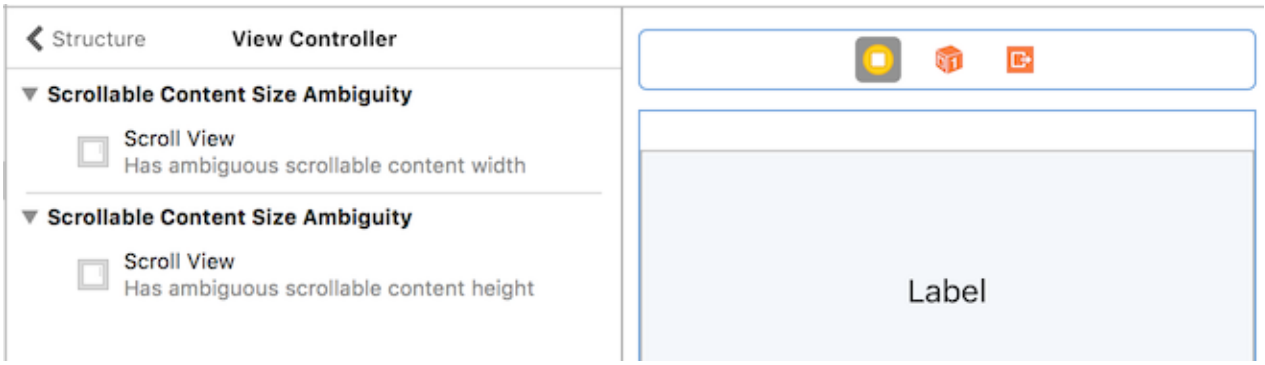


Drag a **scrollview** into the viewcontroller, and create constraints on **four sides** (leading, trailing, top and bottom set to 0).



Then drag a **label** into the scrollview (make sure the label is inside the scrollview in view hierachy). You will see an error indicator like this:

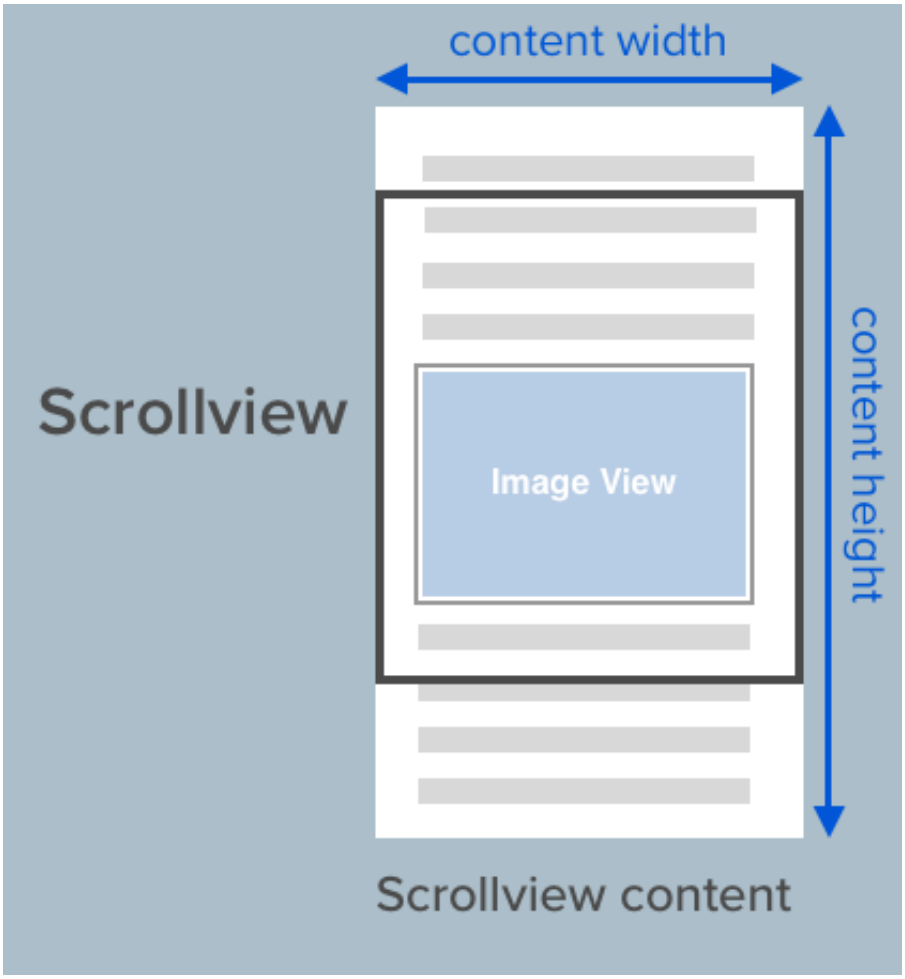
If you click into the error indicator, you will see two error messages stating ambiguity of scrollable content size like this:



What does "ambiguous scrollable content width / height" means? 🤔 We will explain it on the next section.

Structure of scrollview

Scrollview works by having a scrollable content area, like this:

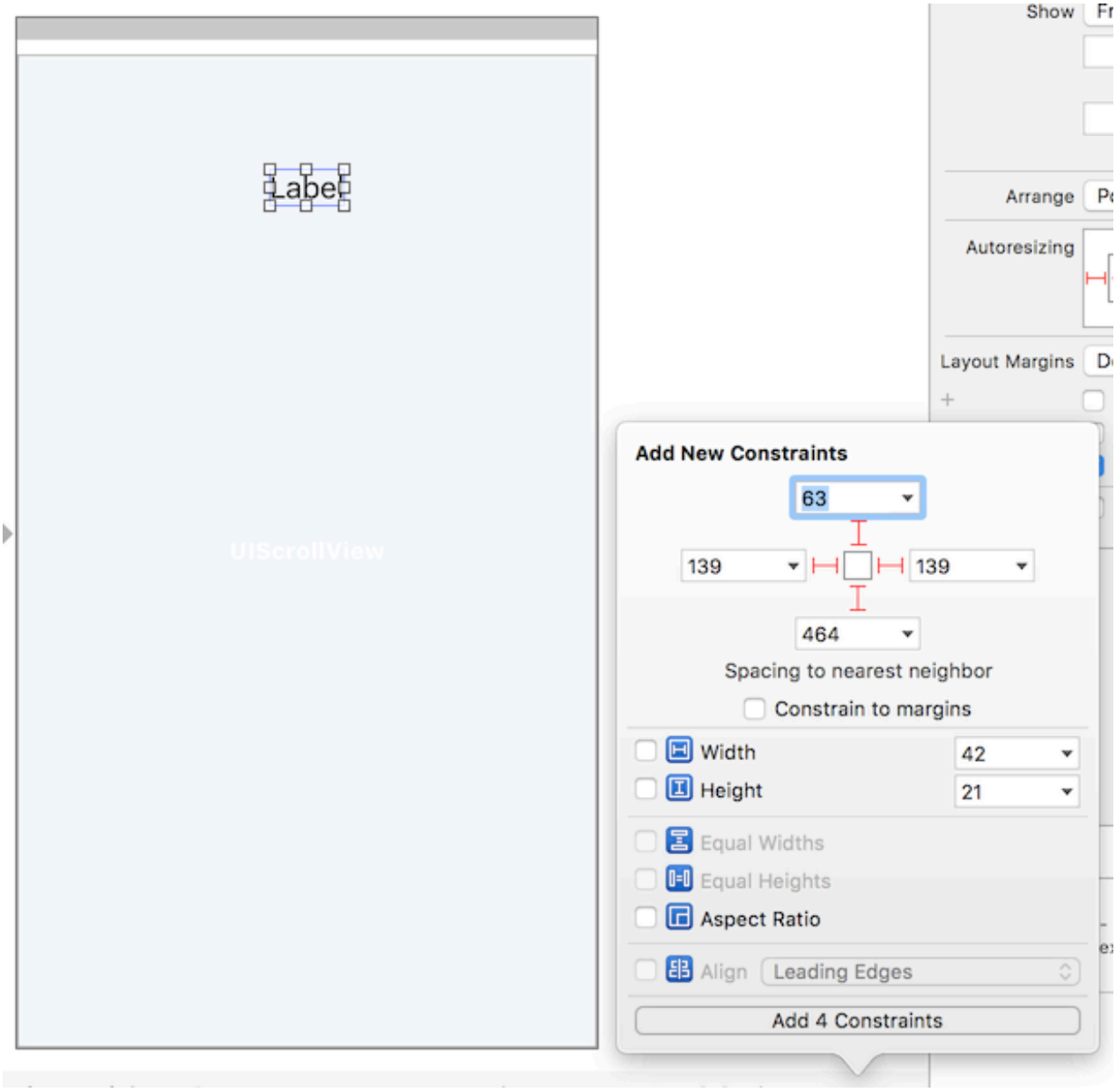


In order for scrollview to work in Auto Layout, **scrollview must know its scrollable content** (scrollview content) **width** and **height**.

Xcode shows you the error "ambiguous scrollable content width / height" because Auto Layout doesn't know what is the width and height for the scrollable content inside scrollview.

Fixing ambiguous scrollable content width and height

To fix the error "ambiguous scrollable content width / height", simply add four side constraints (leading, trailing, top and bottom, use the default value provided by Xcode) to the label.



After adding, you should see blue lines as Auto Layout is able to calculate the scrollable content width / height.

Here's the calculation for the scrollable content size:

```
// iPhone SE screen size
leadingConstraint = 139
trailingConstraint = 139

topConstraint = 63
bottomConstraint = 464

labelIntrinsicWidth = 42
labelIntrinsicHeight = 21
```

```
scrollableContentWidth = leadingConstraint + labelIntrinsicWidth +
trailingConstraint
scrollableContentWidth = 139 + 42 + 139
scrollableContentWidth = 320

scrollableContentHeight = topConstraint + labelIntrinsicHeight +
bottomConstraint
scrollableContentHeight = 63 + 21 + 464
scrollableContentHeight = 548
```

The blue lines seems like a good sign as first sight, but when the label text is changed to longer text, the scrollable content width gets stretched as the intrinsic content width of the label is increased due to longer text. Remember the calculation formula for scrollableContentWidth:

```
scrollableContentWidth = leadingConstraint + labelIntrinsicWidth +
trailingConstraint
// scrollableContentWidth increase if labelIntrinsicWidth increase due to
longer text
```

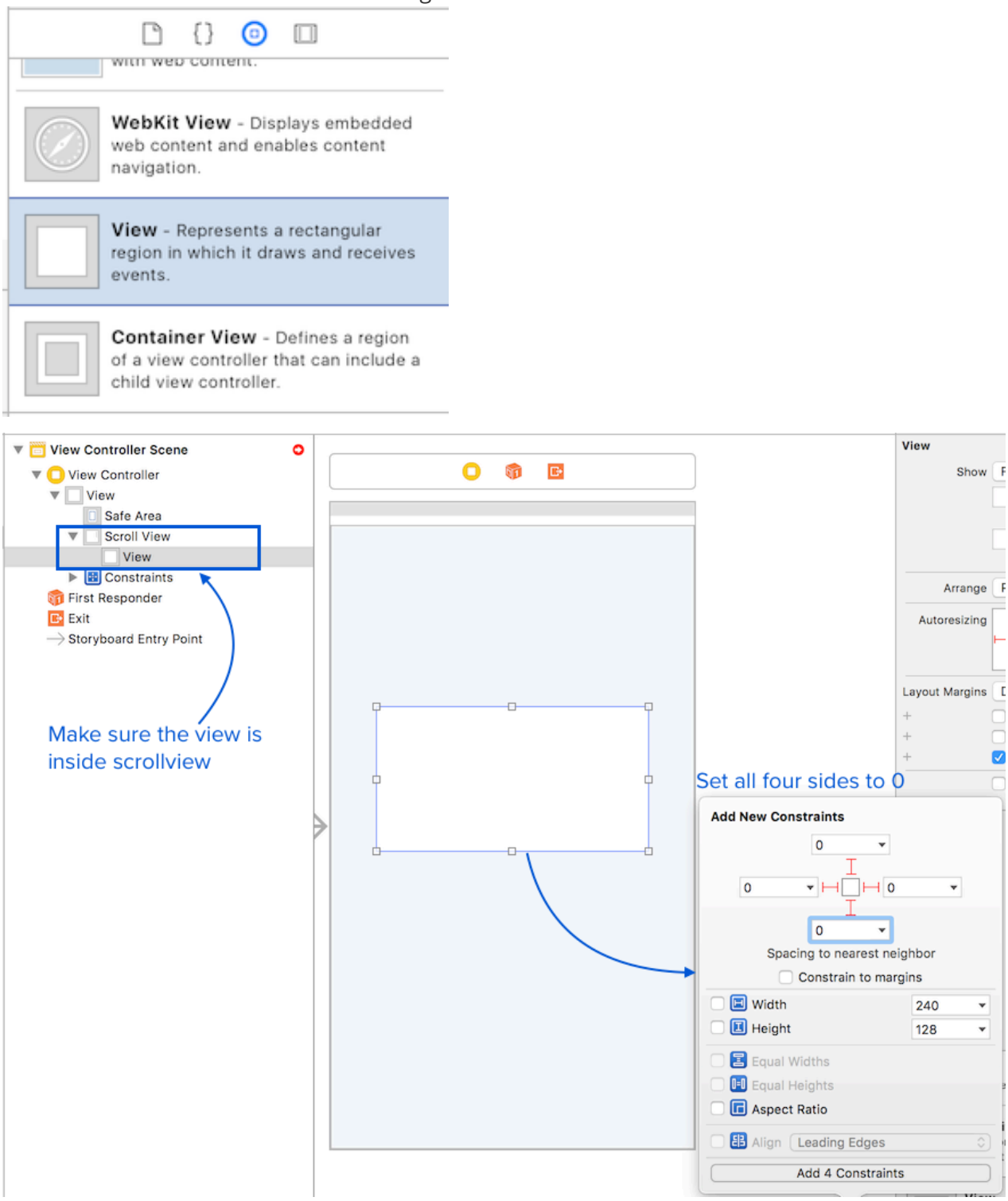
Let's change the label text "Label" to something longer then build and run the app, you will see a scrollview with horizontal scrollbar like this:

This is because the scrollable content width is longer than the scrollview's width. Using a fixed value leading / trailing constraint doesn't work as the intrinsic content width of label is dynamic based on its text. How do we create a scrollview with fixed width regardless of its inner UI elements intrinsic content width?

The way I recommend is creating a content view (UIView) for the scrollview and put UI elements inside the content view.

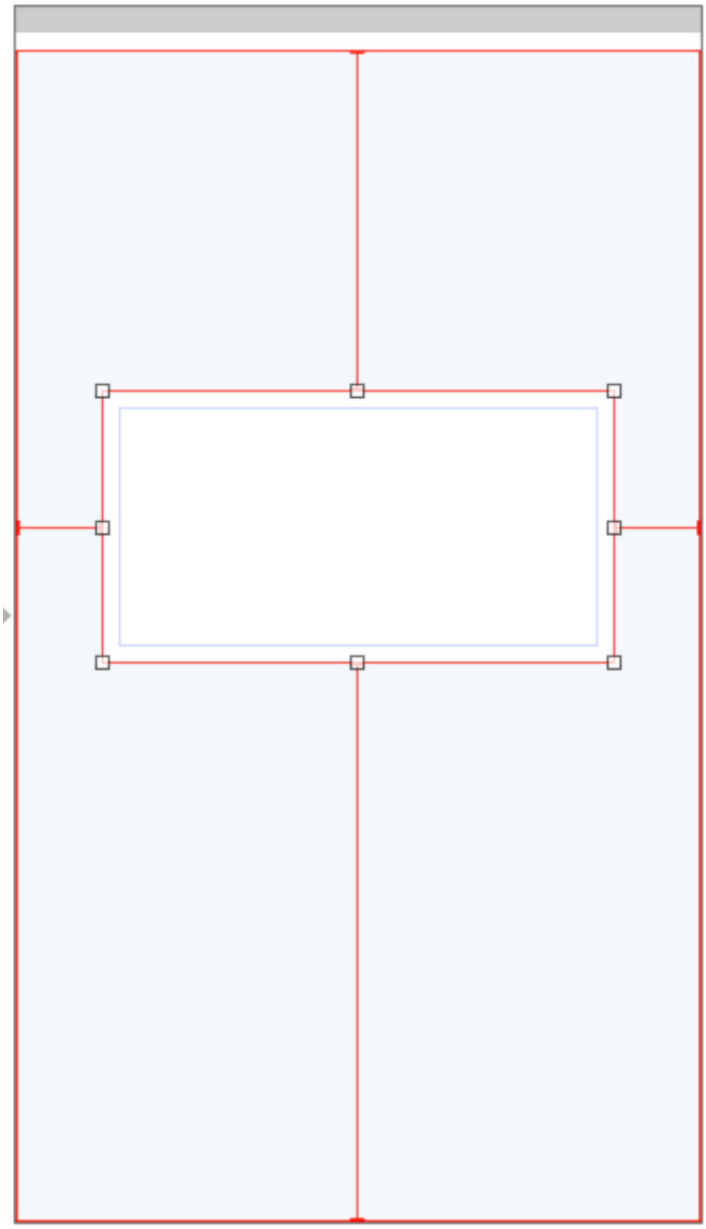
Placing content view inside scrollview

Continuing from previous step, delete the label inside scrollview. Then add a view **inside** the scrollview , and add four side constraints (leading, trailing, top and bottom, set it to 0) to the view. Remember to uncheck 'Constrain to margins' if it is checked.



We will call this view as 'Content View' as this view contains UI elements (content) for the scrollview. Auto Layout will use the **content view**'s' width and height to calculate the scrollable content width / height for the scrollview.

After adding four side constraints, Auto Layout will show you red lines like this (oh no!):

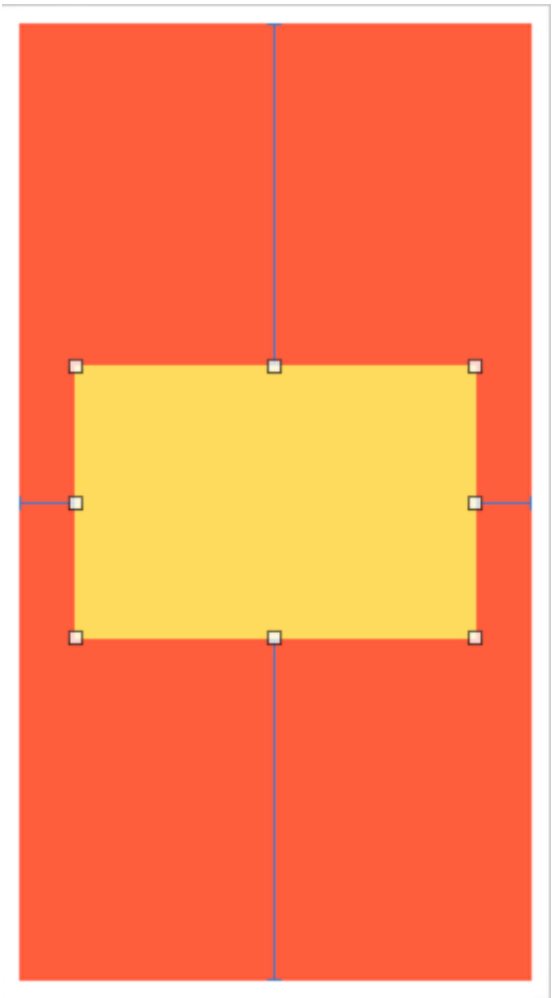


Auto Layout show you red lines because it can't calculate the scrollable content width / height as **UIView doesn't have an intrinsic content size** (refer chapter 6 - what is intrinsic content size).

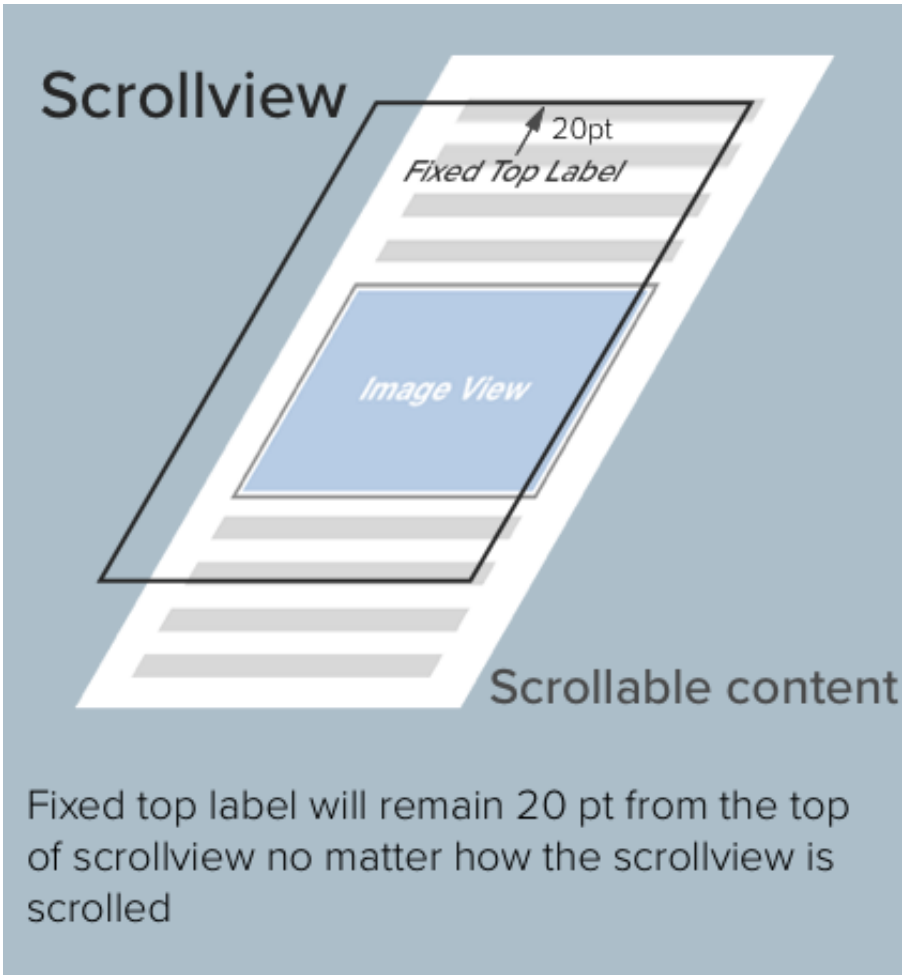
```
// iPhone SE screen size
leadingConstraint = 0
trailingConstraint = 0

topConstraint = 0
bottomConstraint = 0

// UIView doesn't have intrinsic content width!
scrollableContentWidth = leadingConstraint + viewWidth + trailingConstraint

// UIView doesn't have intrinsic content height!
scrollableContentHeight = topConstraint + viewHeight + bottomConstraint
```

You might ask, "Why won't the content view width equal the scrollview width since leading and trailing constraint is defined as 0?"

For a normal UIView, if you put a subview inside the UIView with leading / trailing / top / bottom constraints, it would work nicely like this:

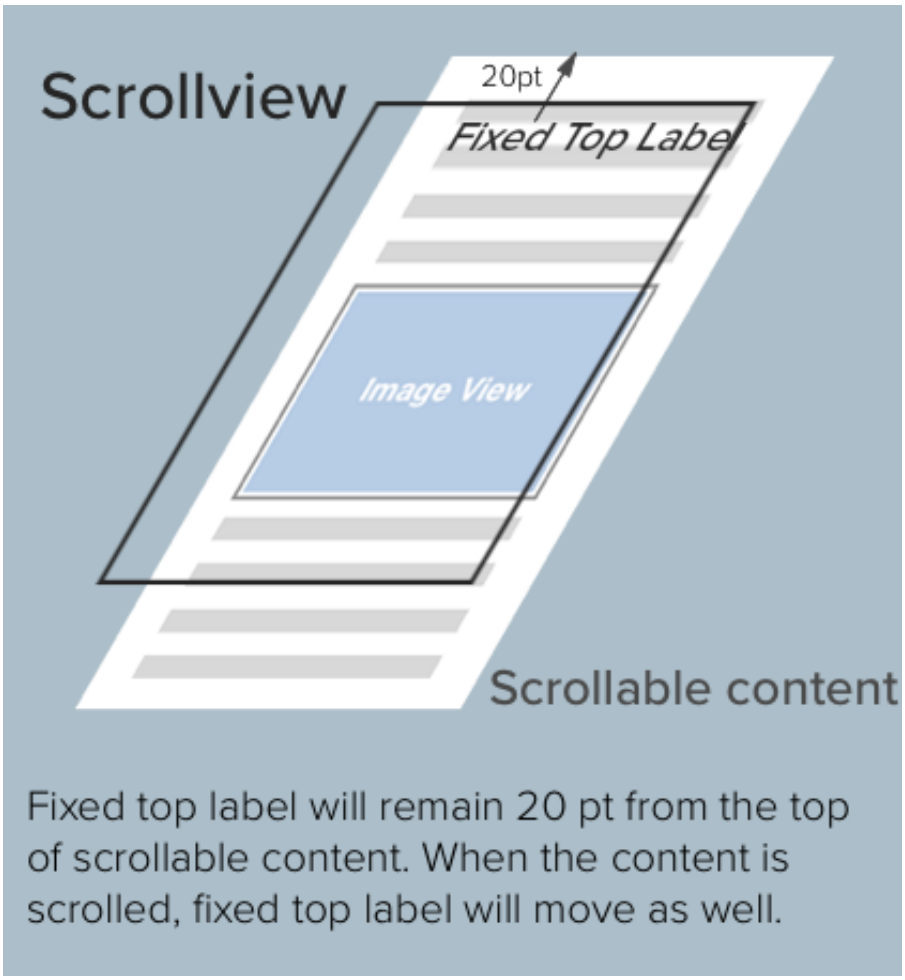


But for scrollview, Apple made the constraints work differently. If scrollview used the same principle of constraint as normal view, let's say we put a label with a top constraint to the scrollview, it would be like this:



The top label will stuck at the top (always 20 pt from the top of scrollview) no matter how you scroll the scrollview, it won't move even if you scroll! That's not what we want, aren't it?

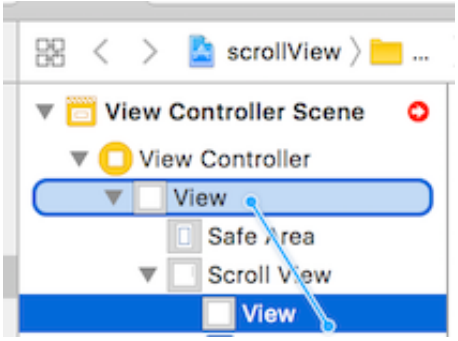
Of course Apple is smart enough to know this, so Apple made a special case for constraint inside scrollview. Apple made it so that any constraint from UI Elements inside the scrollview to the scrollview is actually linked to the scrollable content of the scrollview, not the scrollview itself.



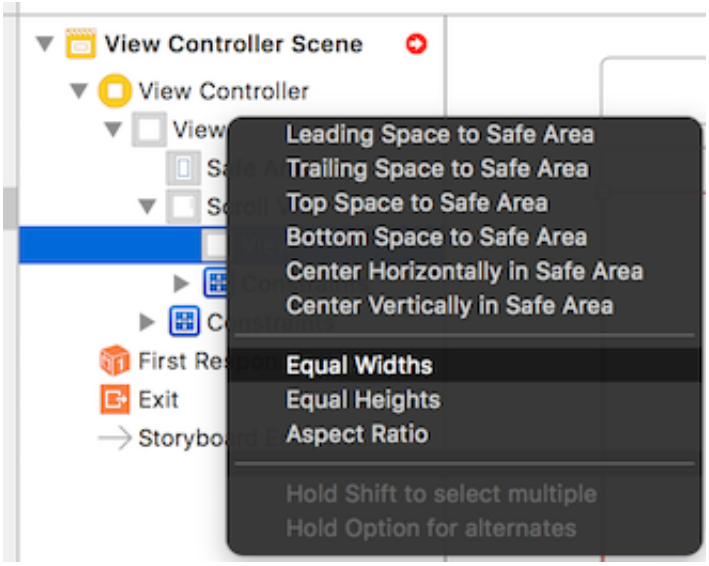
The scrollable content size is derived from the intrinsic content size (or explicitly defined size) of UI elements inside + constraint of UI elements inside to the scrollview (actually the scrollable content boundary).

Back to the content view we just placed, since it doesn't have an intrinsic content size, Auto Layout can't calculate the scrollable content size from it. To resolve this, we will first assign an width constraint to the content view. We will make the width of the content view to equal to the width of the parent of the scroll view:

Drag from the content view (the view inside scrollview) to the **parent view of scrollview**, like this:



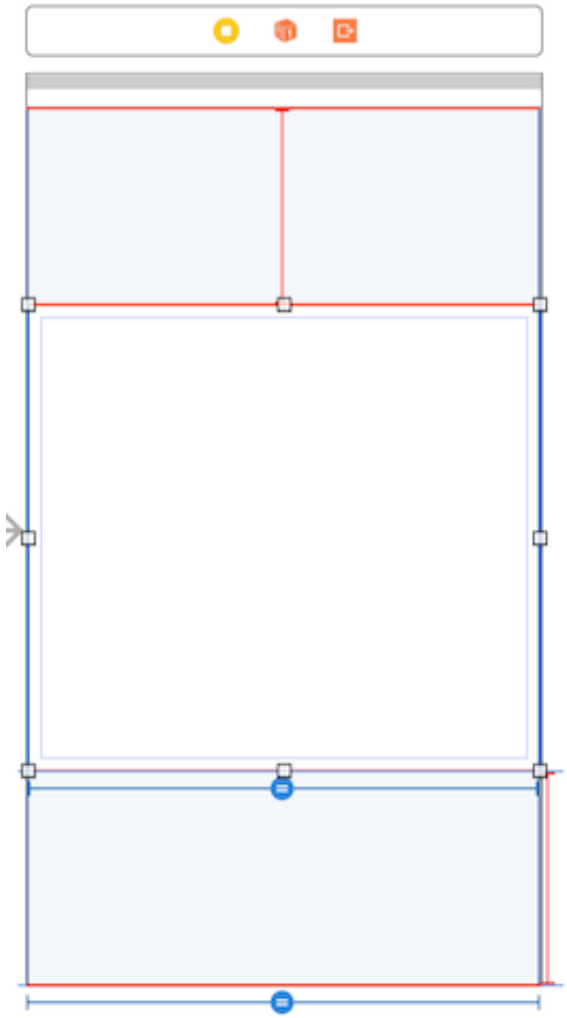
And select "Equal width", like this:



This will make the content view (the view inside scrollview) has a width of the screen width. Now that the content view has a defined width, Auto Layout can calculate the scrollable content width.

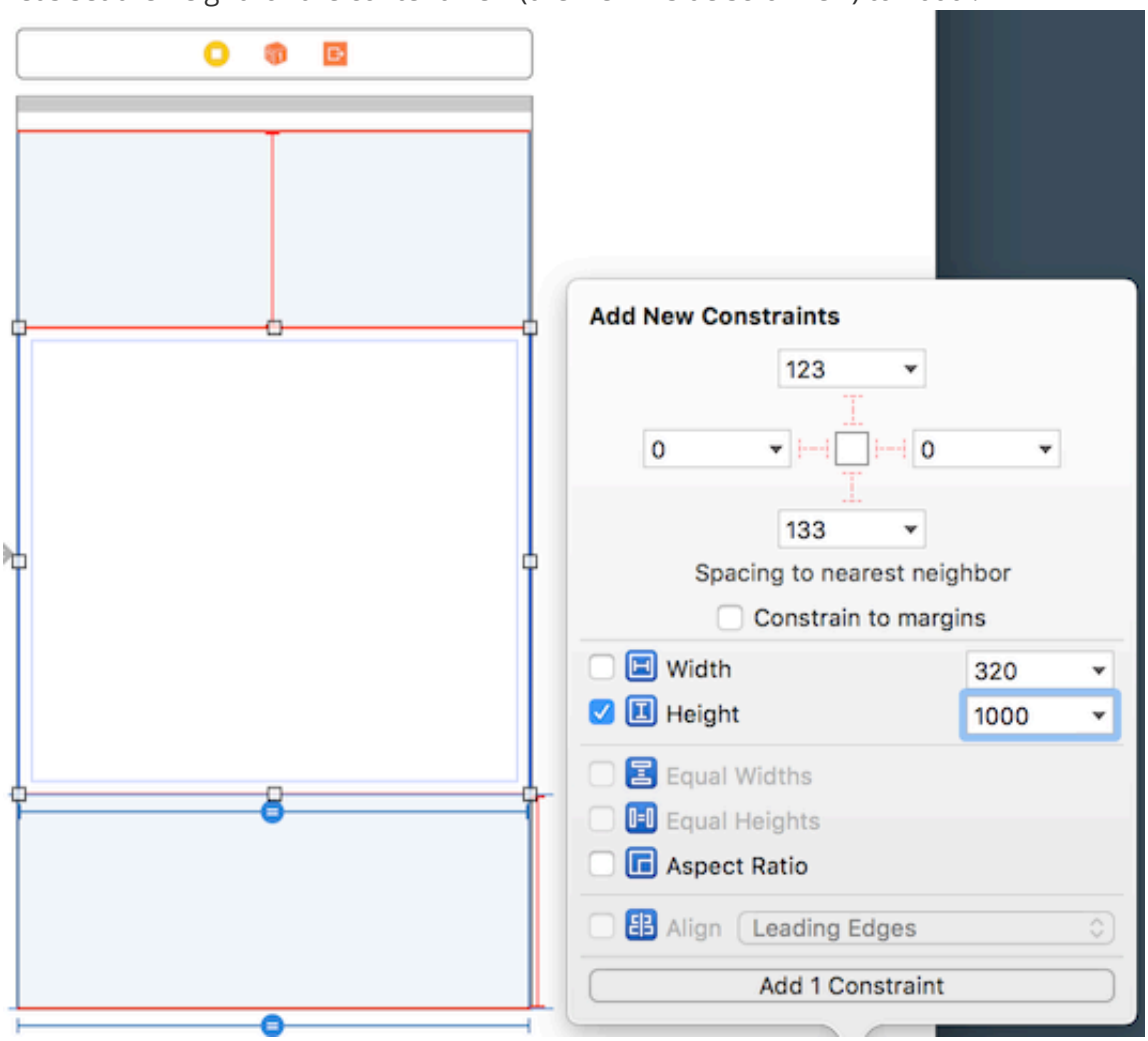
```
// iPhone SE
screenWidth = 320
contentViewWidth = screenWidth = 320
leadingConstraint = 0
trailingConstraint = 0

scrollableContentWidth = leadingConstraint + contentViewWidth +
trailingConstraint
scrollableContentWidth = 0 + 320 + 0
scrollableContentWidth = 320
```

But we still left with a red line as the height of the content view is not defined yet,

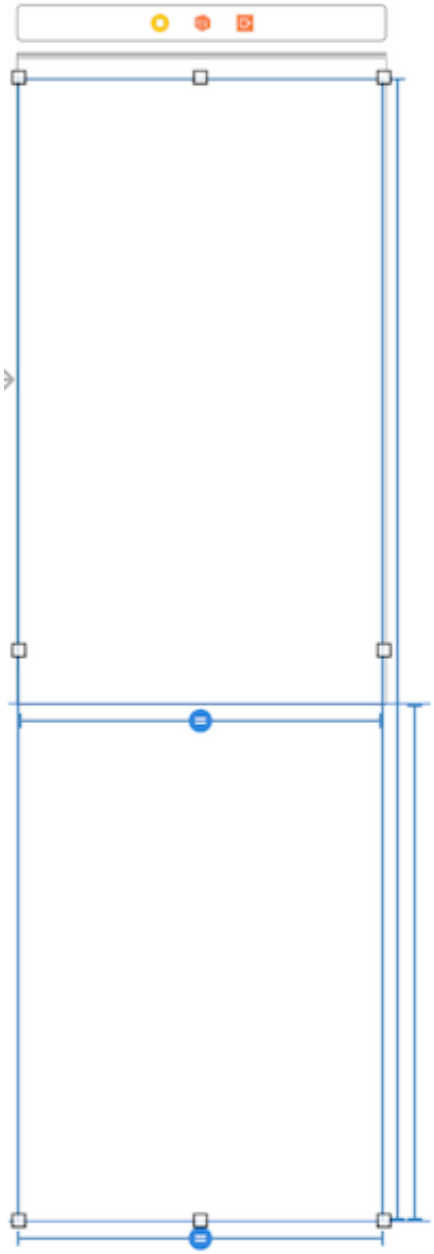


To **temporary** resolve this, we will set an explicit height value to the content view (the view inside scrollview). We will remove it later once we have finished placing elements in it as we want to make it dynamic, remember?

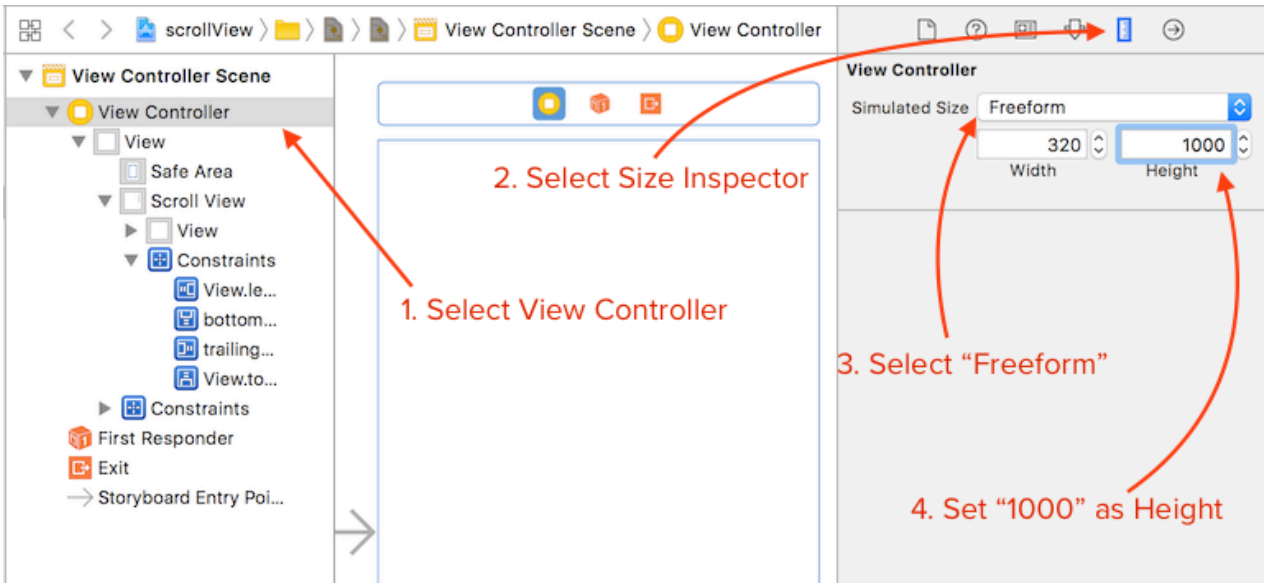
Let's set the height for the content view (the view inside scrollview) to 1000 :



Now we have all blue lines, yay! (we will remove the height constraint after finish adding UI elements). But now the content view height is larger than the view controller height, the extra height of content view is being cropped by the boundary of view controller height:



To resolve the cropped part of scrollview, we can elongate the view controller height by setting it to freefrom like this:

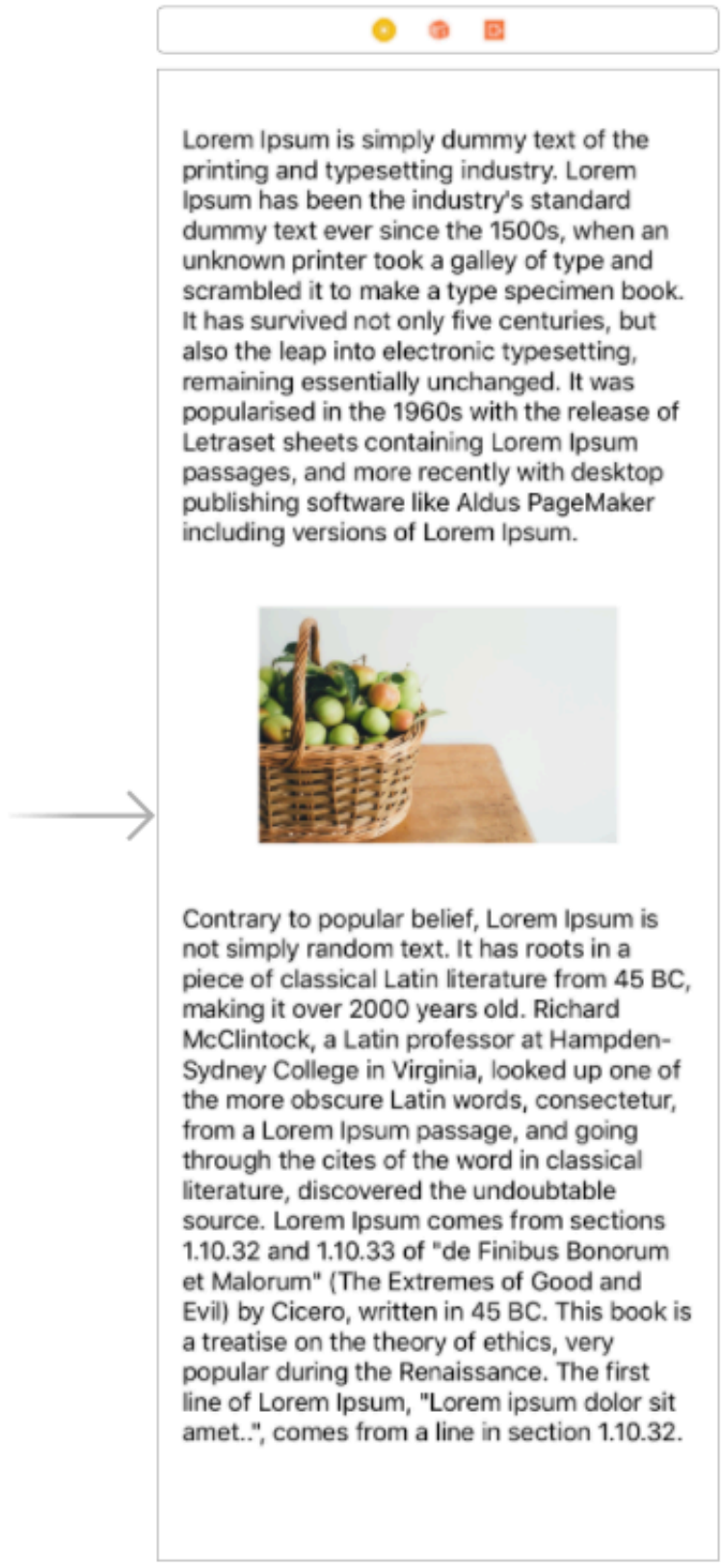


1. Select "View Controller"
2. Select "Size Inspector" Icon
3. For "Simulated Size", select "Freeform"
4. Set "1000" as its height

You should see the view controller is elongated to 1000 pt height now. Now you can start placing the label, images etc inside the content view (the view inside scrollview). After placing, be sure to set constraint from the top element to the bottom. You can increase the height of view controller / height constraint of content view as you need.

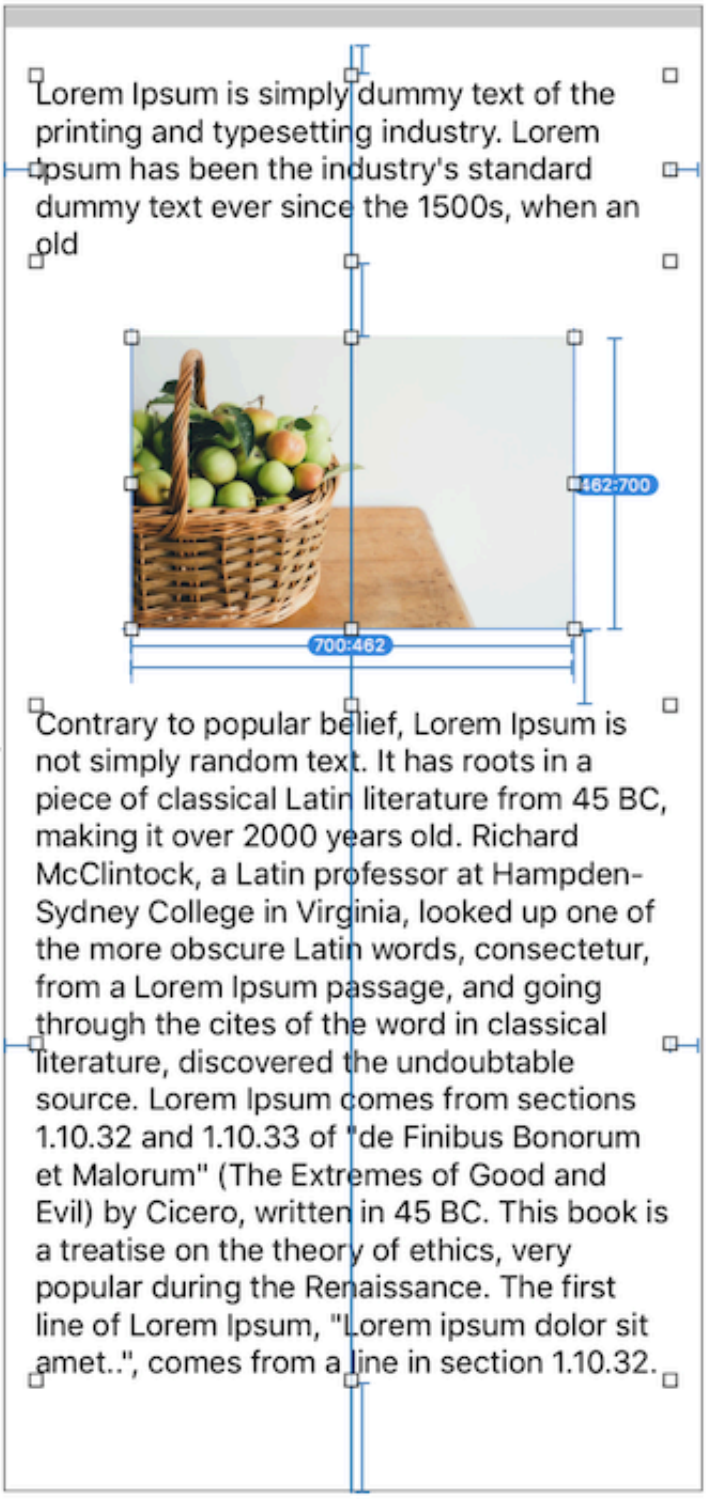
Placing UI Elements inside Content view

Here's an example after placing labels and image inside the scrollview:

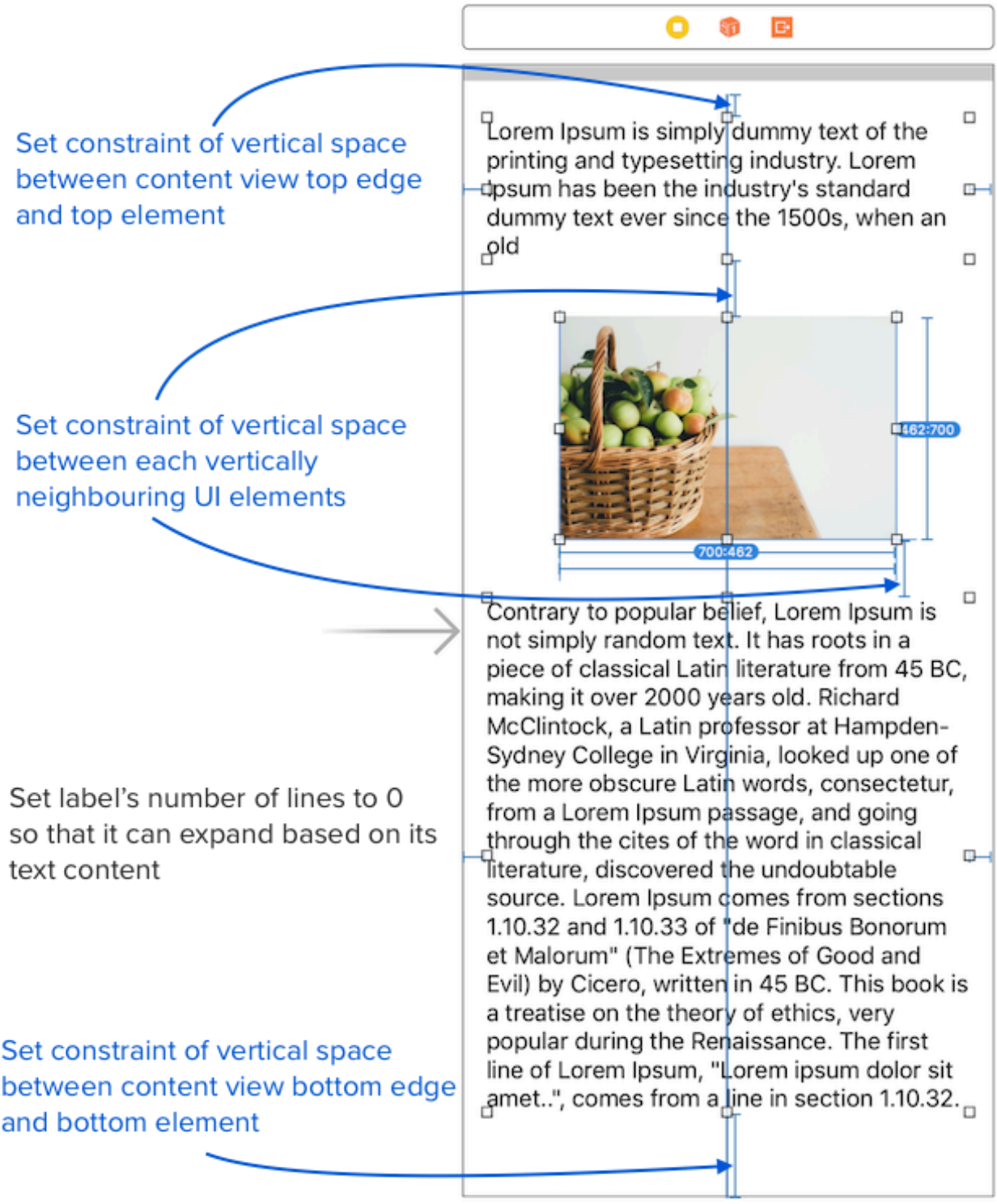


After placing the elements, add constraints for them so that Auto Layout can calculate their X position, Y position, width and height (can skip width and height if they have intrinsic content size). Remember that the constraint should be between UI Elements and the content view, not with the scrollview.

Your constraints might look like this:



Remember to set enough constraints so that Auto Layout can calculate the scrollable content height after removing the explicit height



The important note is that the vertical constraints must be connected from the top edge of content view to the bottom edge of content view. Then only Auto Layout can calculate the scrollable content height.

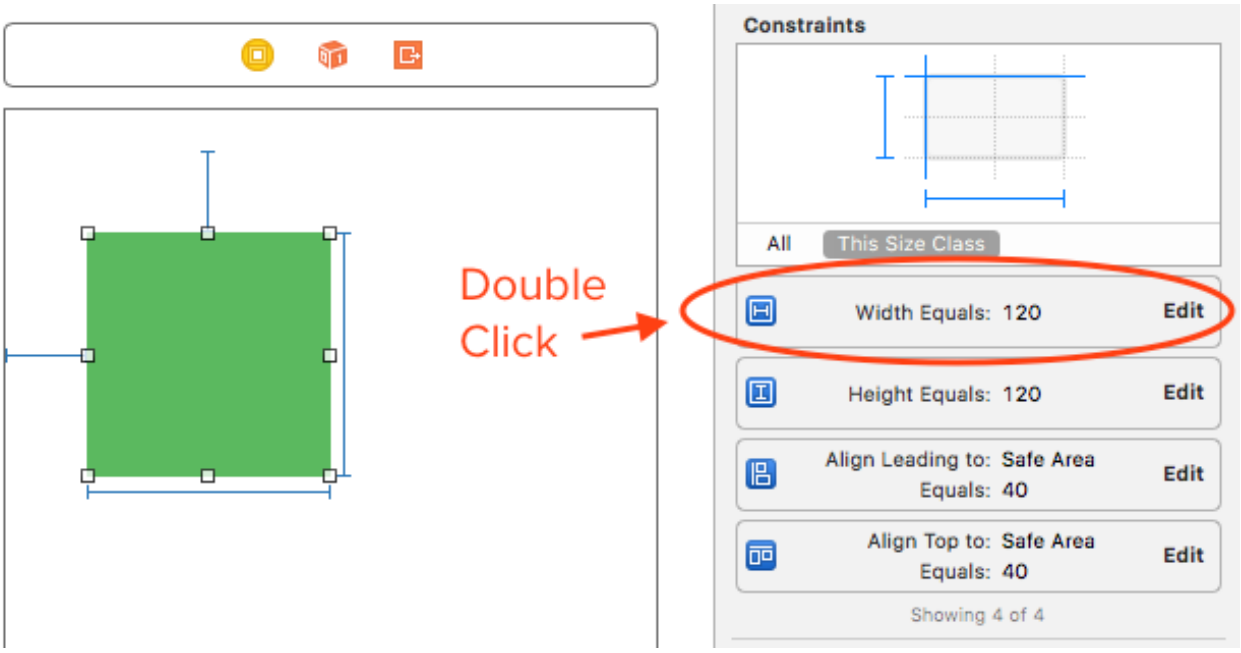
Now remove the explicit height constraint set on the content view, you should get all blue lines if you have set the constraints correctly.

After setting up constraint properly, you can create outlet for the labels and set the text value to the response value gotten from web API. Your scrollable content height will expand based on the intrinsic content size of the labels.

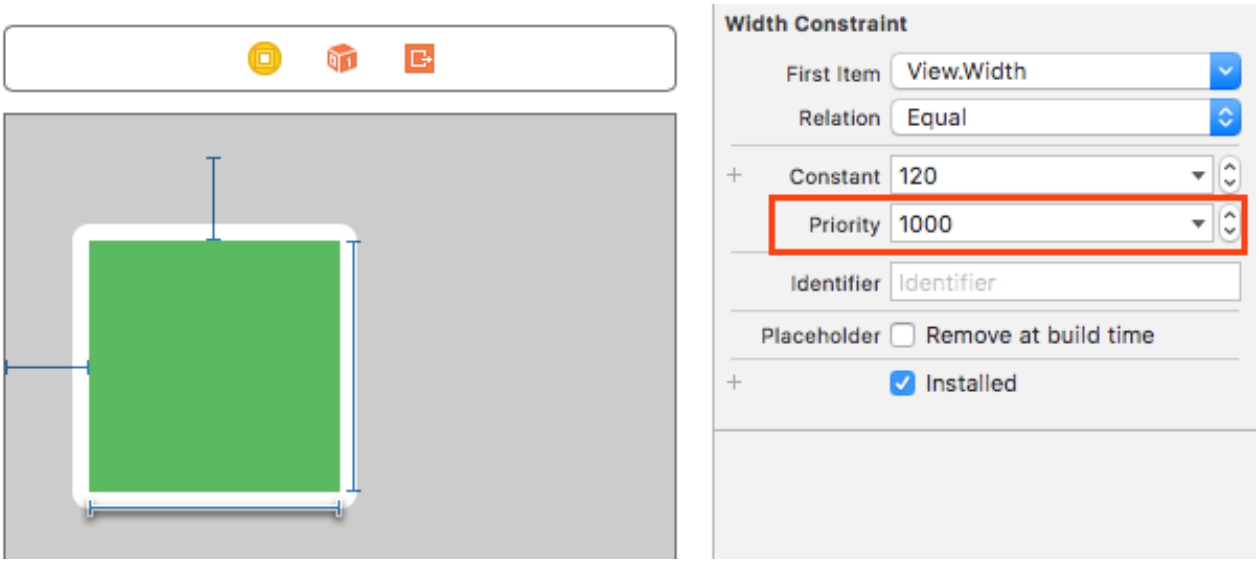
You can check out the completed sample project for this chapter here:
`exercises/FluffyScroll/FluffyScroll.xcodeproj`

9 - What is constraint priority and how to use it

To access constraint priority for an UI element, you can double click into the constraint.



You can see there's a field named "Priority", this is the priority of a constraint.



Constant priority is a number to determine how important is that constraint. The number can range from 1 to 1000, the higher the number goes, the more important a constraint is.

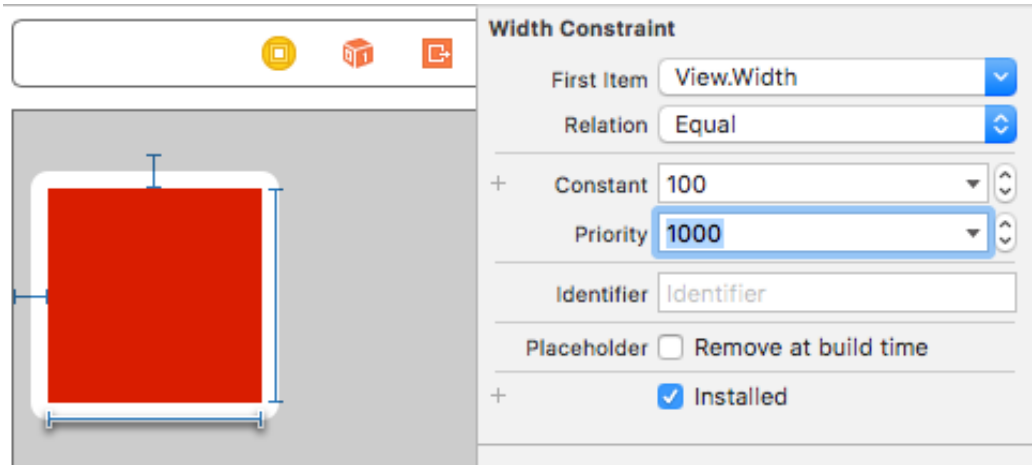
What does **important** mean? When there are conflicting constraints, Auto Layout Engine will attempt to break / ignore constraint with lower priority number first, hence making constraint with higher priority having higher importance.

If there are two conflicting constraints with the same priority number (making it a tie), Xcode will show you **red lines with numbers**.

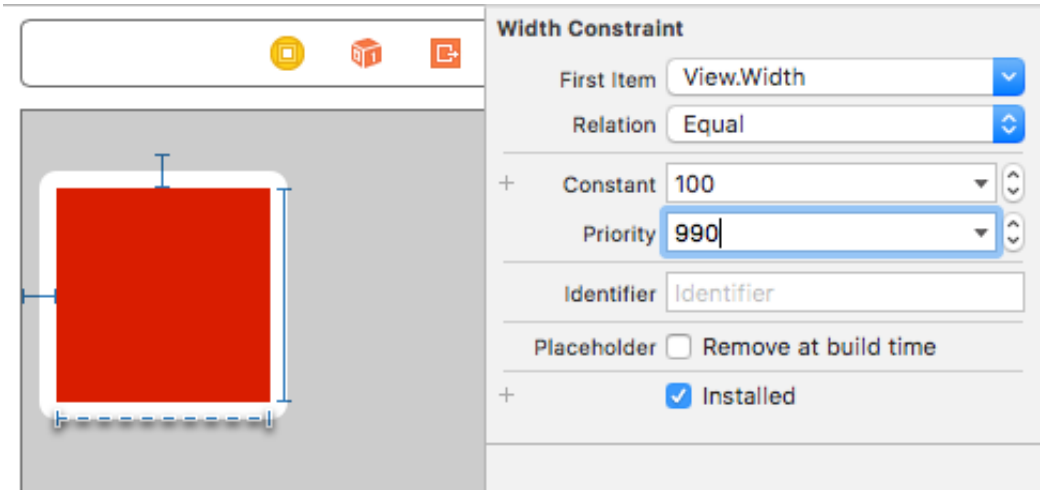
When you define a new constraint in Interface Builder / Storyboard, Xcode will automatically assign the new constraint priority as 1000. **1000** is the highest constraint priority (as defined by Apple), meaning it is **mandatory** as no other constraint can have higher priority than this and this constraint will never be ignored in favor of others.

On the other hand, constraints with priority less than 1000 (1-999) are called as **optional constraint** as it might be ignored in favor of other constraint which have higher constraint priority.

When you normally create a constraint (Xcode default set it to 1000 priority, mandatory constraint), it will show as a blue solid line like this :



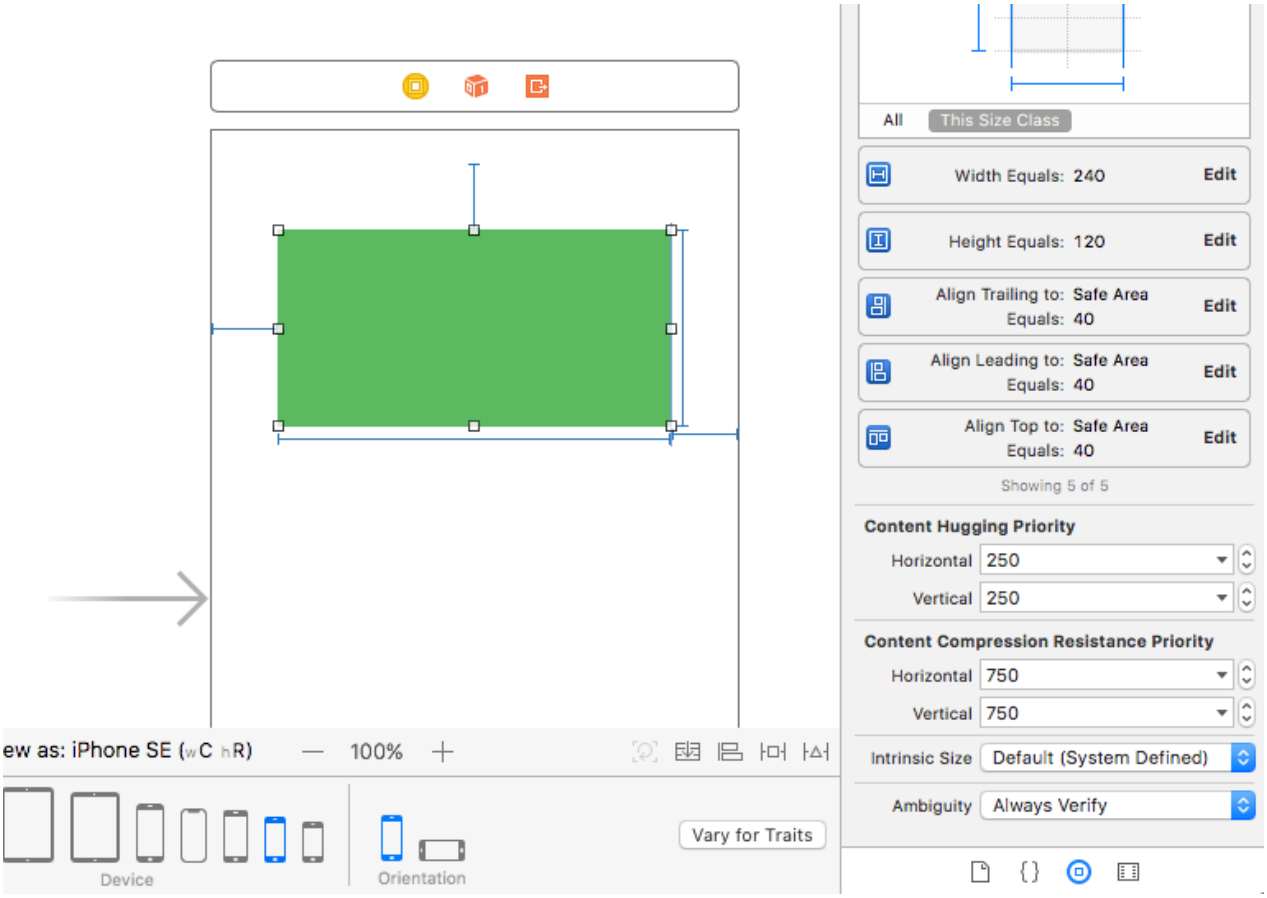
If you change the constraint priority to any number lower than 1000 (optional constraint), it will show dashed line like this :



Example of using constraint priority

Let's use an example similar to the one we used in chapter 5 (why conflicting constraint happen).

Here's a green view with some constraints defined, it works good on iPhone SE :



Its constraints are :

1. The width of green view is 240
2. The height of green view is 120
3. Leading space from screen left to left of the green view is 40

4. Trailing space from screen right to right of the green view is 40
5. Top space from screen top to top of the green view is 40

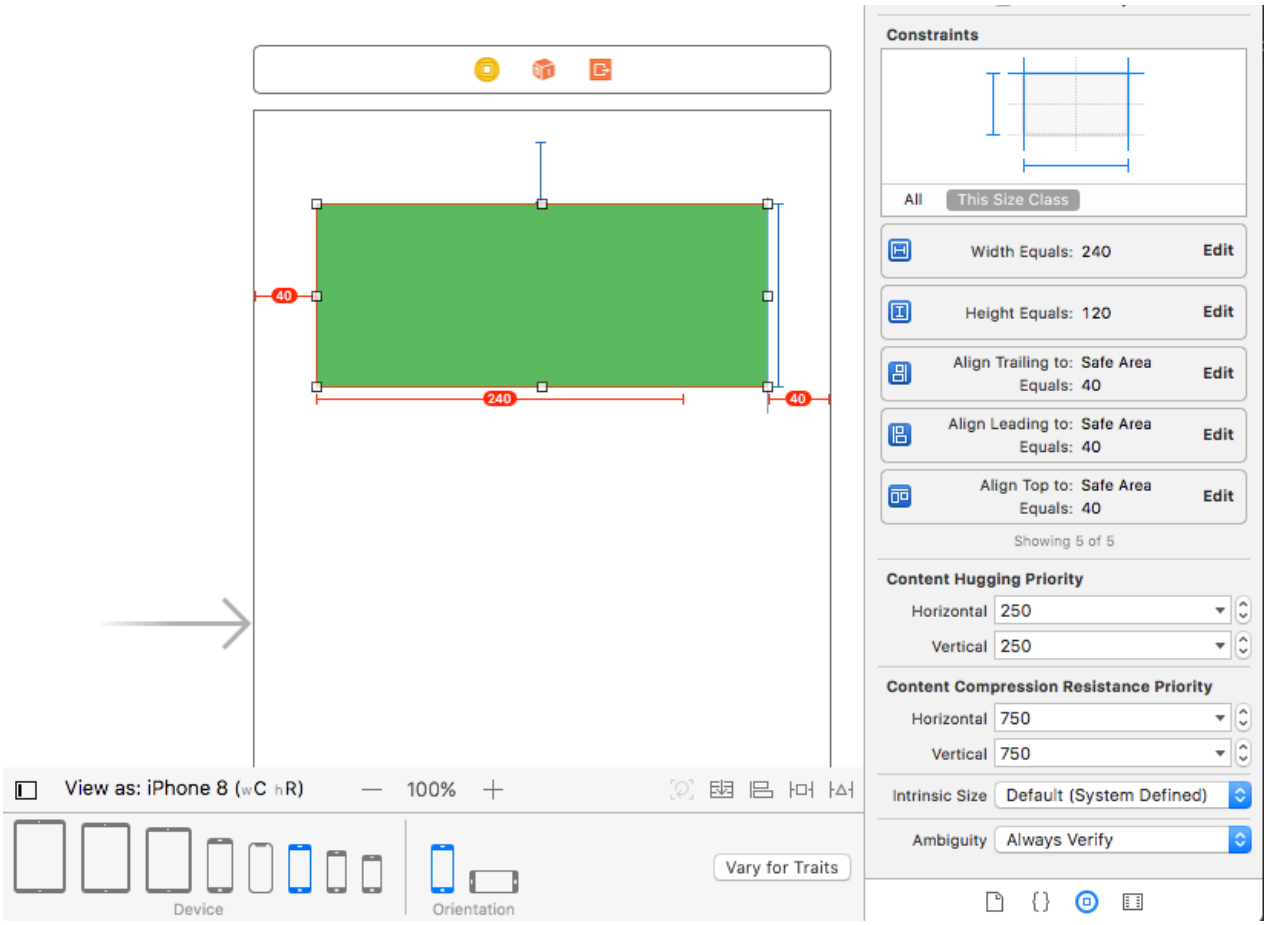
If you have read chapter 5 - "Why conflicting constraint happen" previously, you know that constraint 1 and 3, 4 will be conflicting on another screen size.

In iPhone SE, its screen width is 320 pt. Auto Layout will use the screen width minus leading space and trailing space to calculate the width of the green view :

```
// iPhone SE
screenWidth = 320
leadingSpace = 40
trailingSpace = 40

greenViewWidth = screenWidth - leadingSpace - trailingSpace
greenViewWidth = 320 - 40 - 40
greenViewWidth = 240
```

Using the calculation above, Auto Layout calculated that the width of green view should be 240. Then we also have another constraint explicitly specified the width of green view to 240, since these two values are equal, Xcode doesn't get confused and show blue lines.

But when we view this layout in iPhone 8 size, Xcode will complain that theres conflicting constraints :



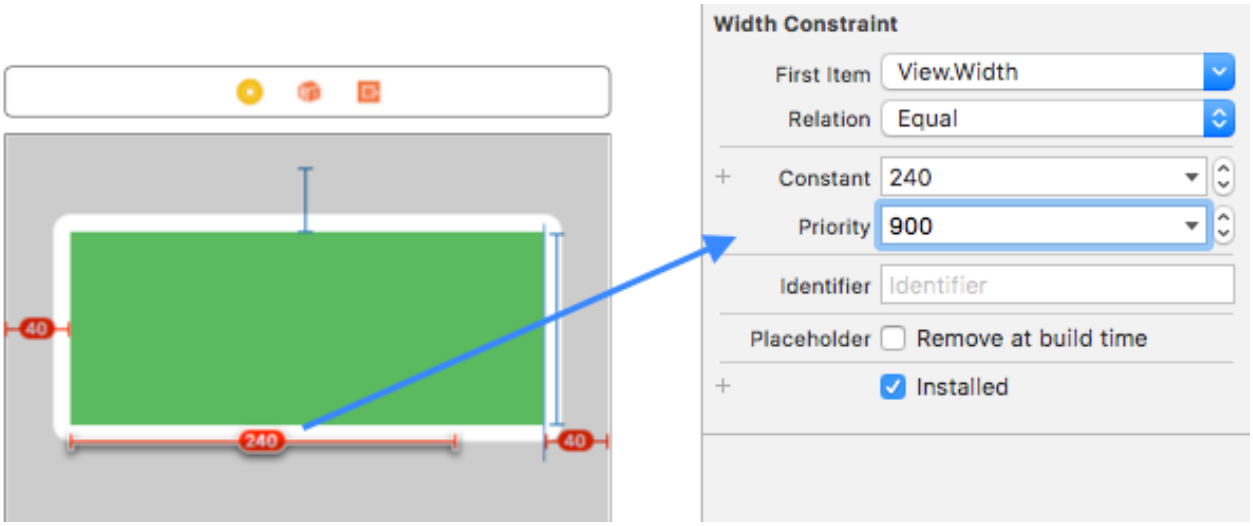
It's conflicting because iPhone 8 screen width is 375. By using the leading and trailing constrains defined, Auto Layout calculate that the width of green view :

```
// iPhone 8
screenWidth = 375
leadingSpace = 40
trailingSpace = 40

greenViewWidth = screenWidth - leadingSpace - trailingSpace
greenViewWidth = 375 - 40 - 40
greenViewWidth = 295
```

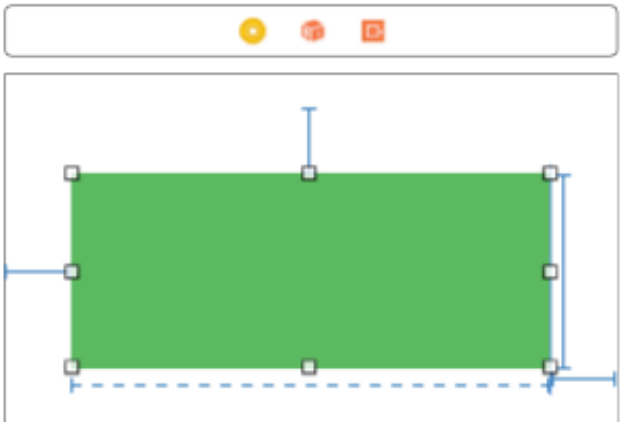
By calculation, width of green view will be 295, but at the same time there's also another constraint which explicitly define the width of green view to equal to 240. It is impossible to make the green width to be both 295 and 240 at the same time, hence Xcode complains using red lines with numbers.

Notice that all the constraints are solid line, meaning they are mandatory and have the same constraint priority (1000). Previously, we solved this by removing the explicit width constraint, now we can use constraint priority to solve this.

Let's set the priority of the explicit width constraint to a lower value, say 900.

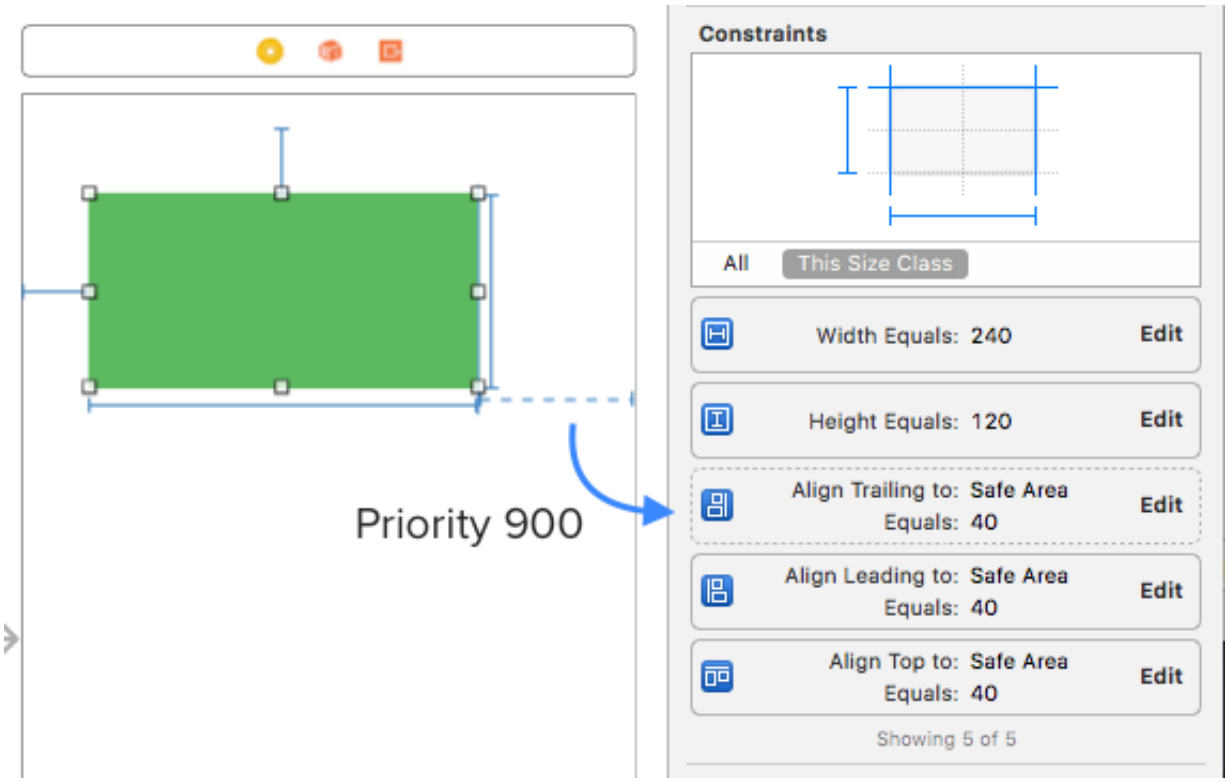


This will solve the conflicting constraint hence Xcode will now show blue lines :



Since the constraint priority of the explicit width constraint have been set to 900 now, the leading constraint and trailing constraint has higher priority (1000). Thus when there's two possible width value, Auto Layout will pick the one derived from constraints with higher priority. In this case, the leading and trailing constraint. The constraint with lower priority will be ignored by Auto Layout Engine if conflicting situation arise.

Similarly, we can keep the explicit width constraint priority to 1000 and instead lower the trailing constraint priority to 900.

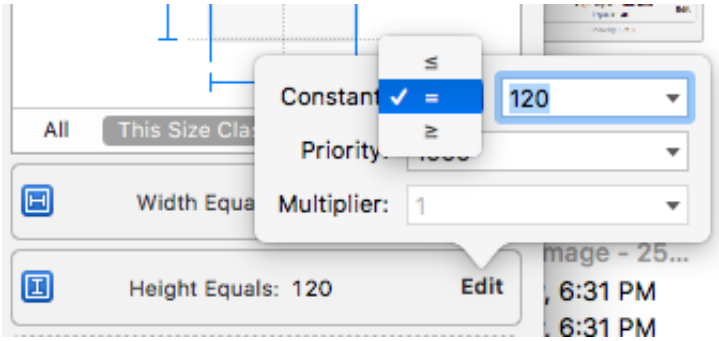


This will make the explicit width constraint priority higher than the trailing constraint, thus Auto Layout will use the explicit width constraint as the width of the green view and ignore the trailing constraint (and also the calculation of width by using leading/trailing constraint). Doing this will make the green view width become 240 for all screen sizes.

Usage of constraint priority

Constraint priority is useful when you have possible conflicting constraint and you want to tell Xcode which of them can be safely ignored when conflict arise in order to resolve conflict. Usually you will want to modify constraint priority of a constraint which uses **larger or equal to** (>=) / **smaller or equal to** (<=).

eg: You can set a constraint that specify the view height must be larger or equal to 120 pt.



Constraint priority is also used when dealing with content hugging / compression resistance constraint. We will talk more about this in the next chapter.

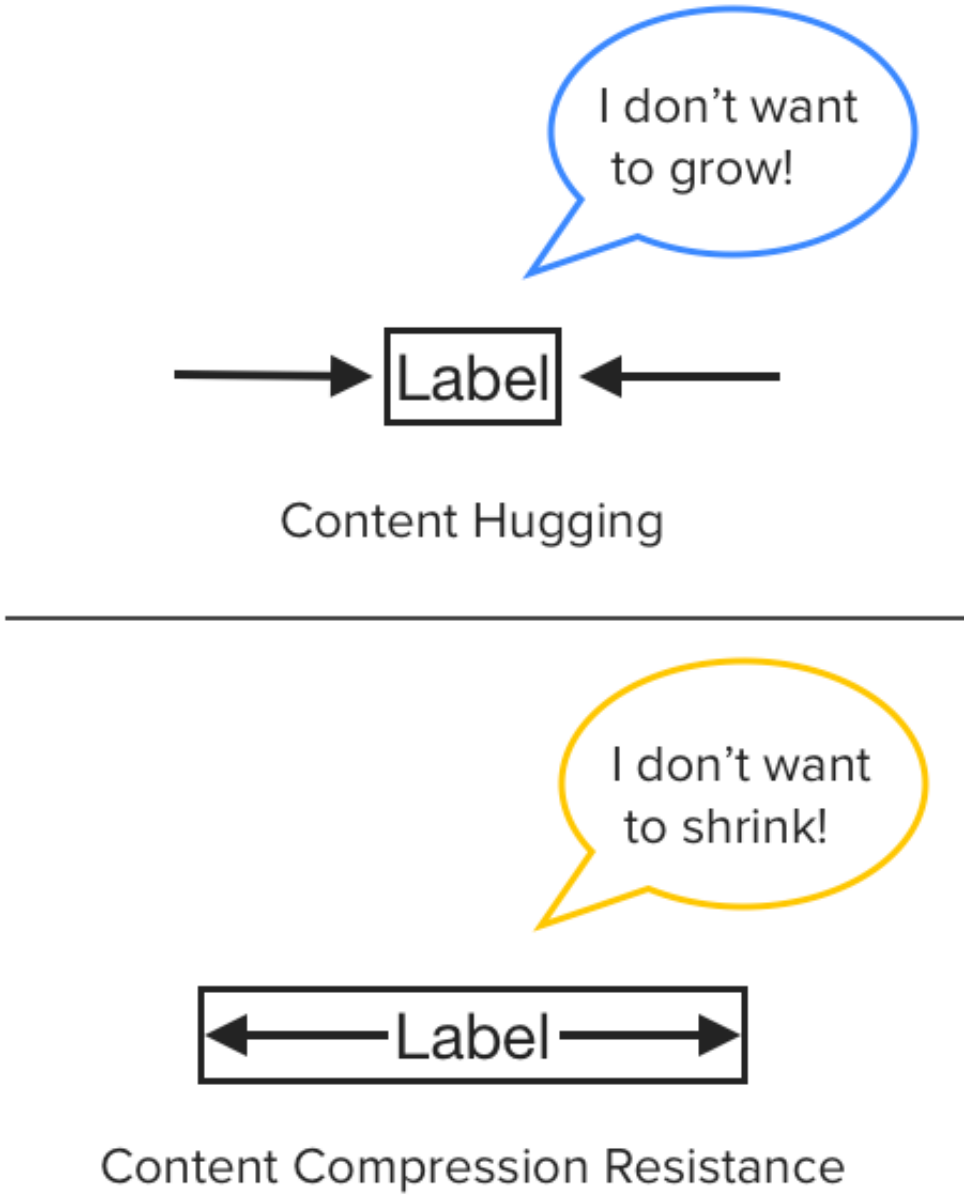
10 - What is content hugging and compression resistance

As a recap, intrinsic content size of an UI element is the width / height derived based on the content of it. (eg: The intrinsic content size of a Label is calculated using the length of its text).

Content hugging of an UI element resist it growing larger than its intrinsic content size. You can think someone is 'hugging' the UI element so tight that it prevent the UI element from expanding than its intrinsic content size.

Content compression resistance of an UI element resist it being shrinked smaller than its intrinsic content size. You can think the UI element has some force inside to resist it being compressed smaller than its intrinsic content size.

Here's a graphical overview of content hugging and content compression resistance:



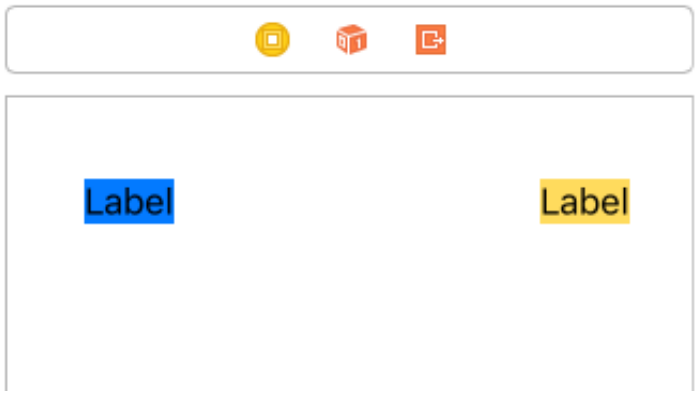
Content priority ambiguity (red lines, with numbers and **@** signs) usually happen when UI elements with intrinsic content size are arranged next to each other horizontally or vertically, and Auto Layout doesn't know whether to expand / shrink which UI element to fulfill the spacing constraint between them.

We will look into how to resolve this ambiguity in the examples below.

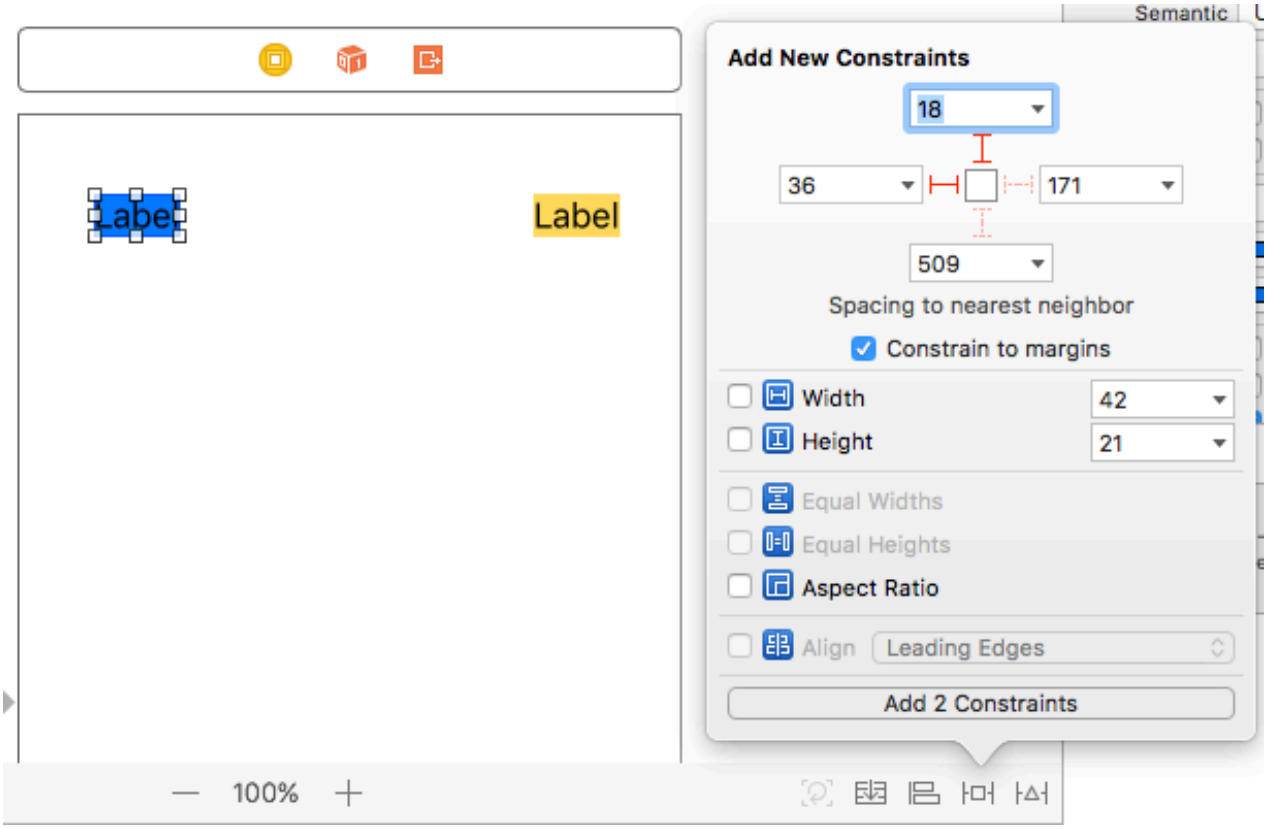
Content hugging priority example

We will use an example to explain what is content hugging and compression resistance below.

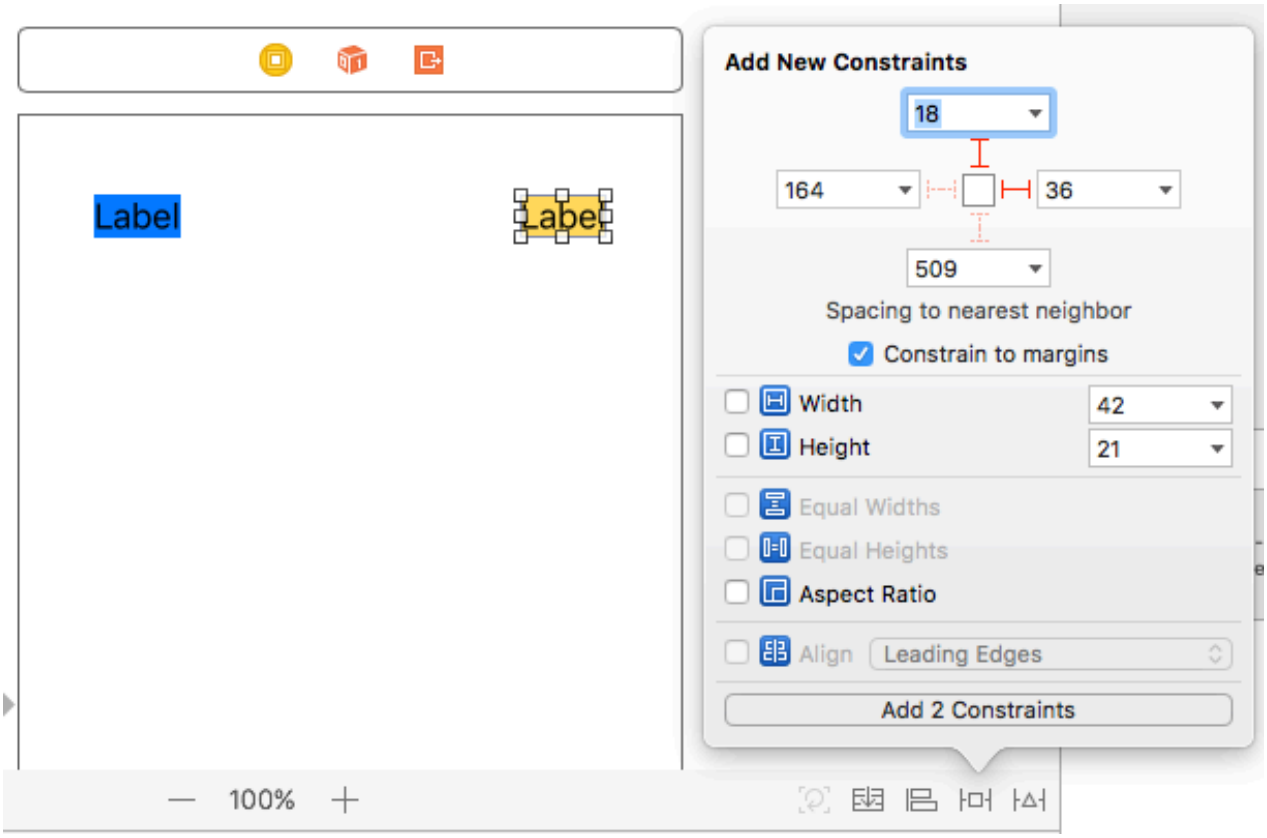
Let's place two label with blue and yellow background in a view controller like this:



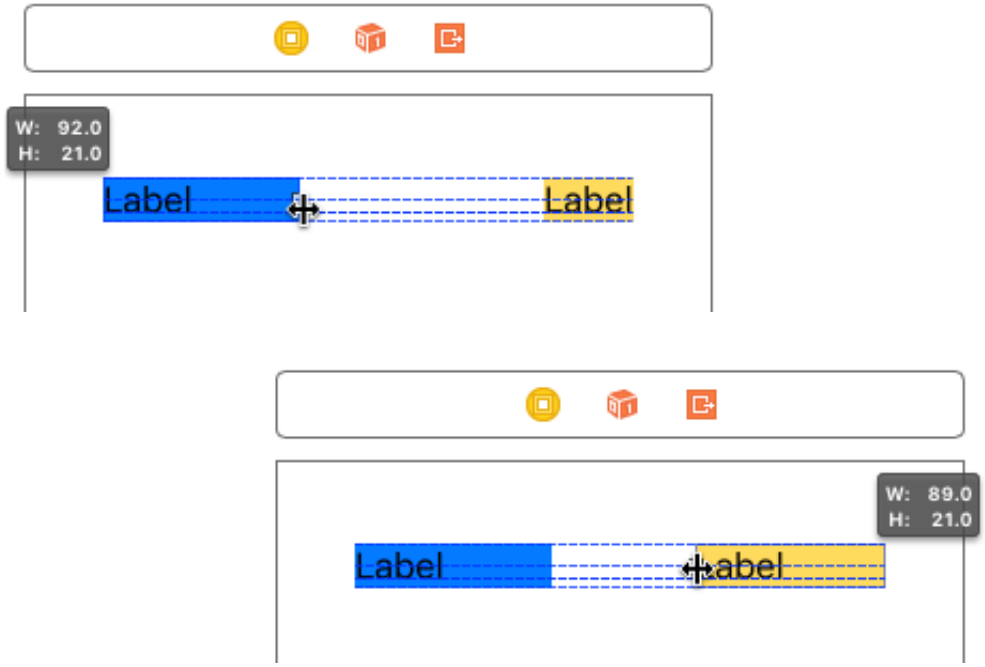
Then add top and leading constraints for the blue label,



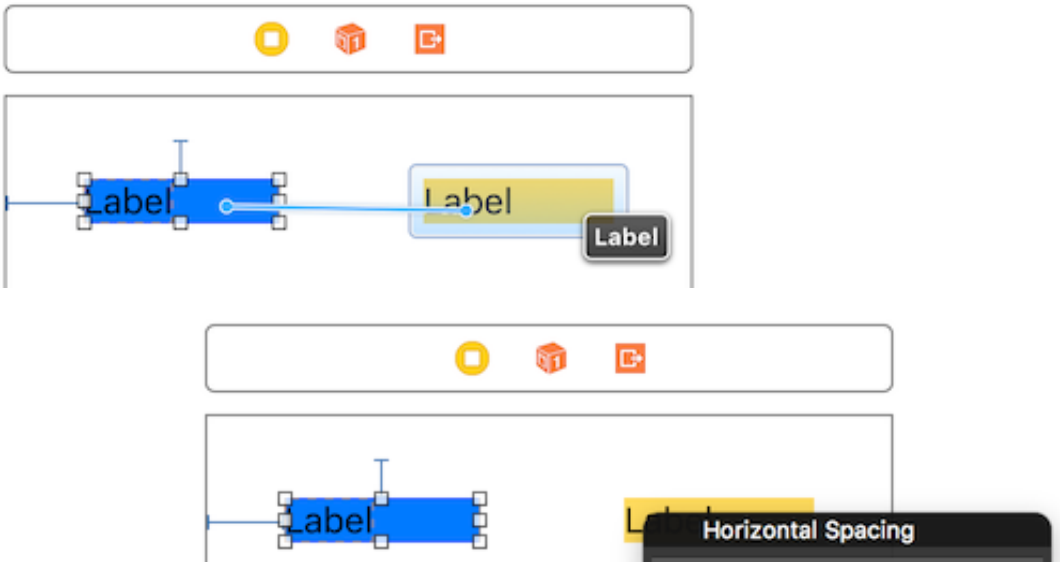
Similarly, add top and trailing constraints for the yellow label,



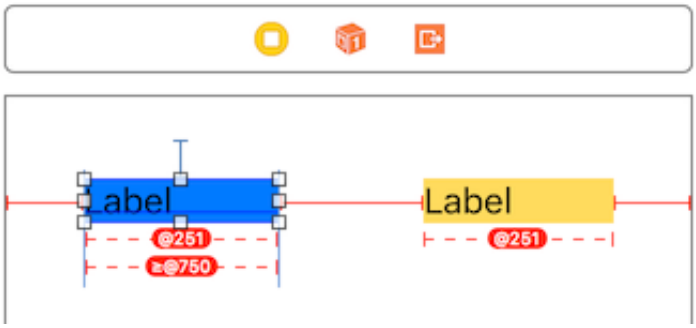
Stretch both labels so that their width is larger than their original intrinsic content width, like this:



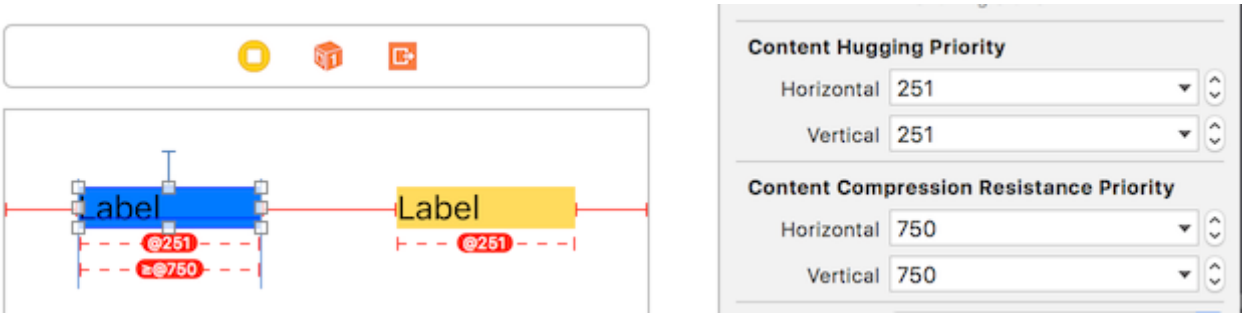
And then add a horizontal spacing between the two labels, like this:



After adding the horizontal space constraint, you will be greeted by several red lines, with @ signs and numbers 😱, like this:

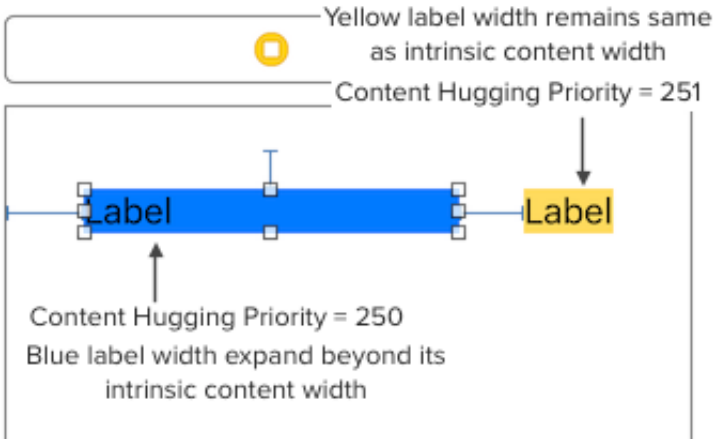


What does the `@251` and `@750` mean? It means the content hugging priority and content compression resistance priority of that label, if you check the size inspector, you can see their values:



Xcode is showing you red lines because you have just set a horizontal spacing constraint (eg: distance between blue label and yellow label must equal to 30) between these two labels, and in order to fulfill that constraint, one of the label have to grow larger than its intrinsic content width , but then Auto Layout doesn't know which label it should enlarge since both of the labels have same value of horizontal content hugging priority.

To resolve this, simply change the horizontal content hugging priority of one of the labels. For this example, we will lower the horizontal content hugging priority of **blue** label from 251 to 250. This is the result after the change:



You will see that the blue label width expands beyond its intrinsic content width while the yellow label width remains same as its intrinsic content width. This is because the yellow label has higher content hugging priority than blue label (251 > 250), higher content hugging priority means yellow label is more likely to hug itself and prevent itself to grow larger than its intrinsic content width.

Since the blue label has lower content hugging priority compared to yellow label, blue label will expand its width beyond its intrinsic content width to fulfill the horizontal spacing constraint between blue and yellow labels.

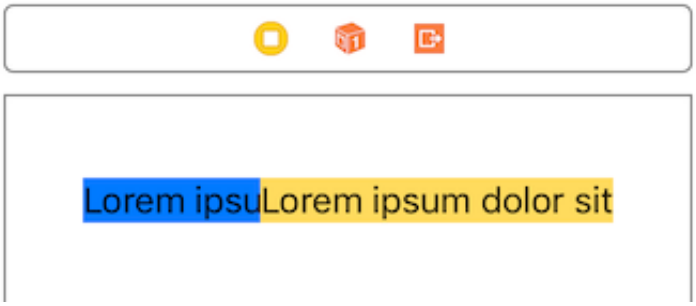
UI element with higher content hugging priority will "hug" itself to stay within the intrinsic content size and let the other UI element with lower content hugging priority to grow instead.

After lowering the content hugging priority of blue label, Auto Layout will know that it should **expand** the blue label width to fulfill the horizontal spacing constraint. Since the ambiguity is solved, Xcode will show you nice blue lines.

By default, Xcode will set the horizontal/vertical content hugging priority to 251 and horizontal/vertical content compression resistance priority to 750 when you drag a new UI element into the interface builder.

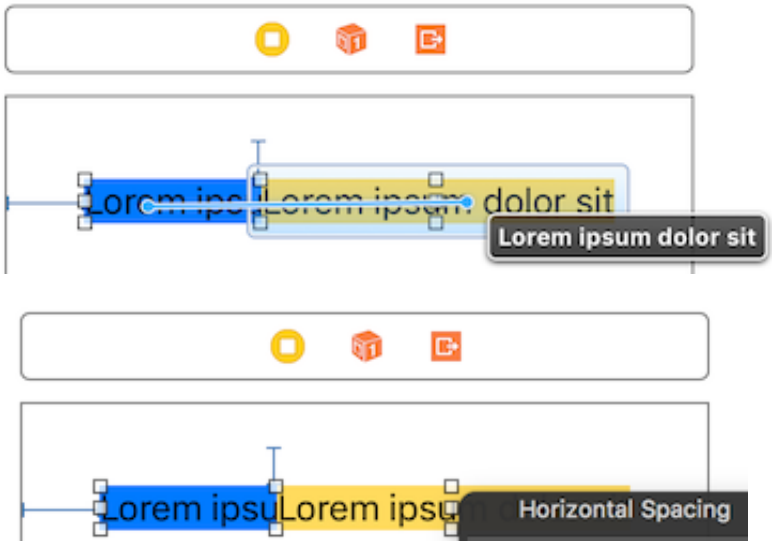
Content compression resistance example

Similar to previous example, we will place two labels next to each other again , but this time their text content is too long that they overlap each other :

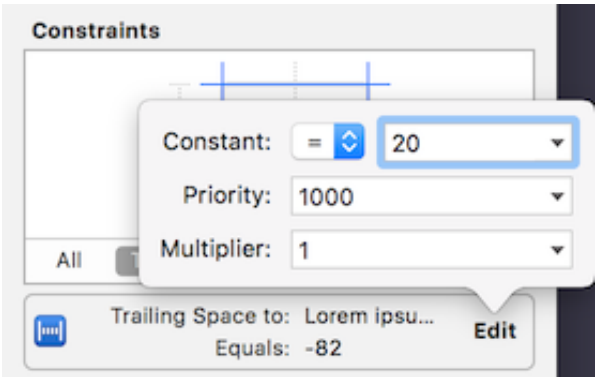


Both blue and yellow label contain the text "Lorem ipsum dolor sit". Blue label has a top and leading constraint, yellow label has a top and trailing constraint.

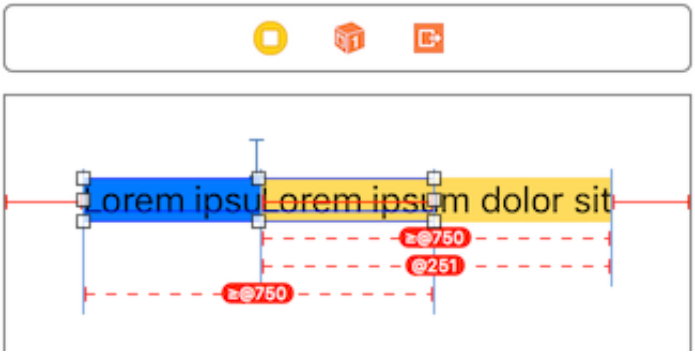
Now let's say we want to add a 20pt horizontal spacing constraint between these two labels:



After adding the horizontal spacing constraint, set its value to 20.



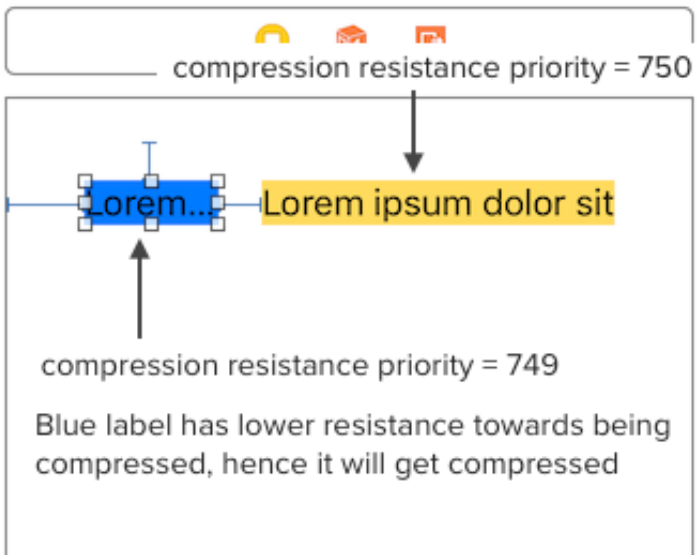
After adding the horizontal spacing constraint (20pt), you will get the farmiliar red lines:



Since both labels' text content is too long and overlap each other, setting a 20 pt horizontal spacing constraint between them would require one of the label to be shrinked smaller so there is enough space in the middle for the 20pt horizontal spacing.

Xcode is showing you red lines because you have just set a horizontal spacing constraint (eg: distance between blue label and yellow label must equal to 20) between these two labels, and in order to fulfill that constraint, one of the label have to shrink smaller than its intrinsic content width , but then Auto Layout doesn't know which label it should shrink since both of the labels have same value of horizontal content compression resistance priority.

To resolve this, simply change the horizontal content compression resistance priority of one of the labels. For this example, we will lower the horizontal content compression resistance priority of **blue** label from 750 to 749. This is the result after the change:



You will see that the blue label width shrink smaller than its intrinsic content width and its text got truncated while the yellow label width remains same as its intrinsic content width. This is because the yellow label has higher content compression resistance priority than blue label (750 > 749), higher content compression resistance priority means yellow label is more resistant towards being compressed, allowing yellow label to maintain its intrinsic content width.

Since the blue label has lower content compression resistance priority compared to yellow label, blue label will be prone to being compressed thus its width is shrinked smaller than its intrinsic content width to fulfill the horizontal spacing constraint between blue and yellow labels.

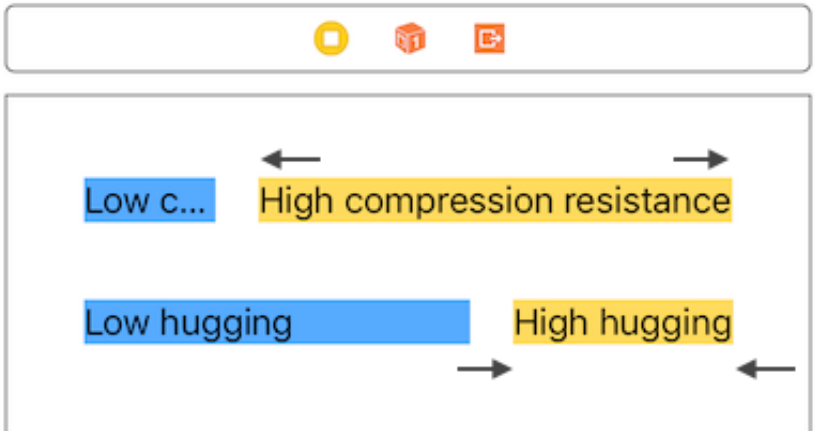
UI element with higher content compression resistance priority will be more resistant towards being compressed thus making itself to stay within its intrinsic content size and let the other UI element with lower content compression resistance priority to get compressed instead.

After lowering the content compression resistance priority of blue label, Auto Layout will know that it should **shrink** the blue label width to fulfill the horizontal spacing constraint. Since the ambiguity is solved, Xcode will show you nice blue lines.

Summary

In this chapter we have shown examples of content hugging priority and content compression resistance priority, although the examples we have shown is in horizontal, the same concept can be applied to vertical axis too.

Here is a summary graph of content hugging priority and content compression resistance priority.



11 - What is Stack View and how does it simplify constraints

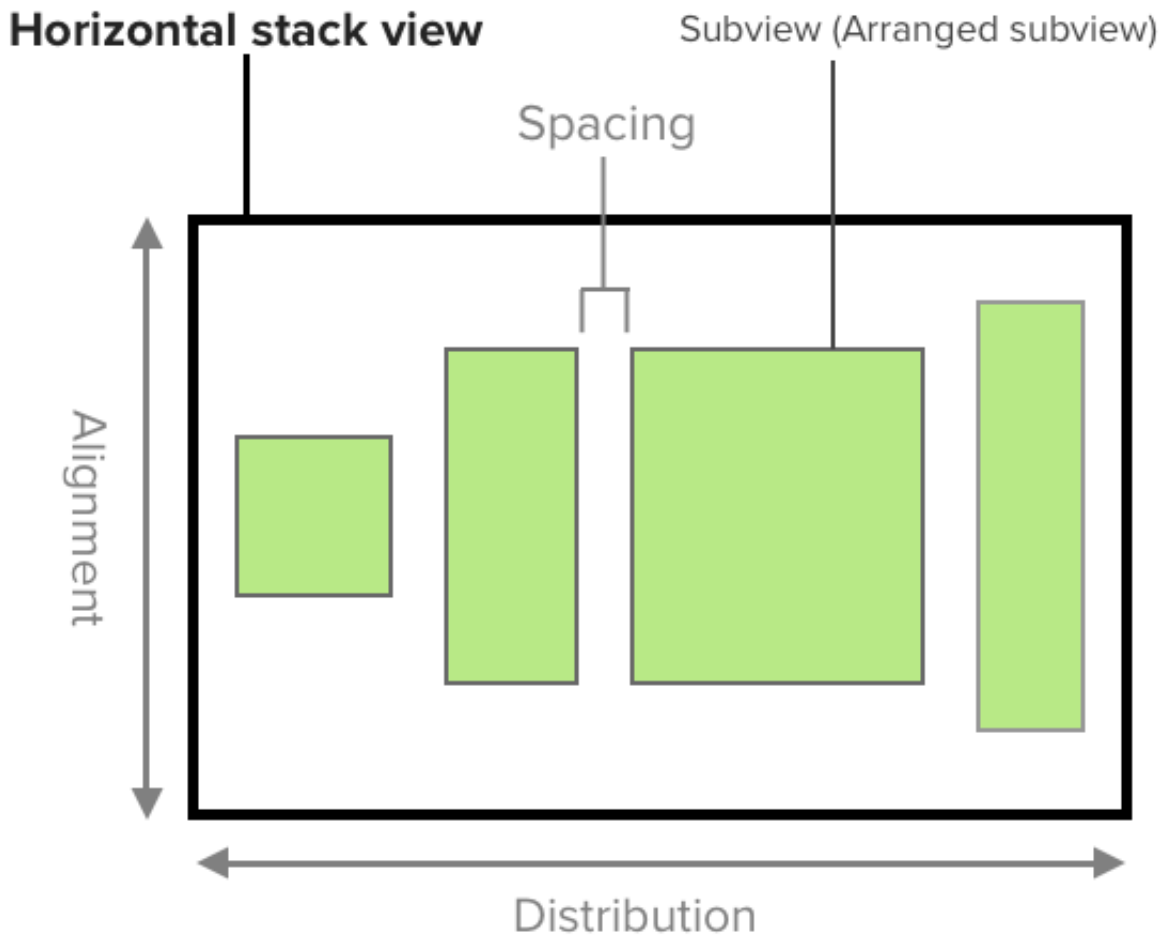
Apple introduced Stack View in iOS 9 and it simplifies the effort needed when a new constraint requirement is added / updated.

Ever experience that after you have finished laying out constraints nicely for every view, then you realized you left out one view? or your client suddenly ask to add another view beside an existing view? Do you update the affected constraint one-by-one? Did you ever end up just deleting all constraints and re-adding them from scratch because it was easier than editing each of the constraint and trying to keep up with the red lines?

Good news, Stack view allows you to add / remove child views inside it easily without needing to change any constraint!

Structure of stack view

Stack view itself is like a normal view, but subviews inside stack view will be arranged according to the stack view 's axis (horizontal / vertical), alignment, distribution and spacing property. Below is the structure of a horizontal stack view:



Stack view's properties such as "alignment", "distribution" and "spacing" will be explained later on in this chapter.

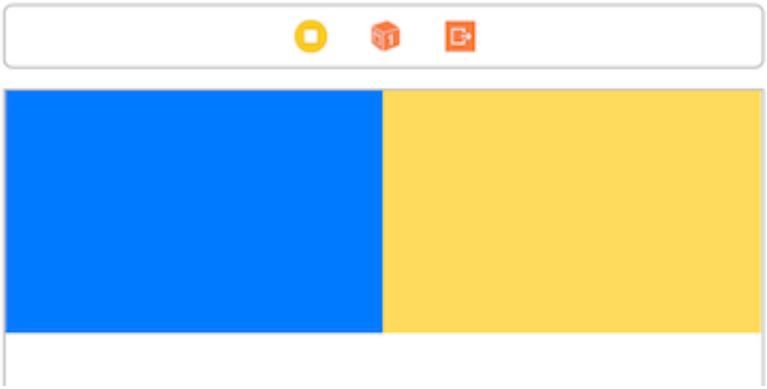
We will start by explaining how Stack View simplifies constraints below.

Example without using Stack View

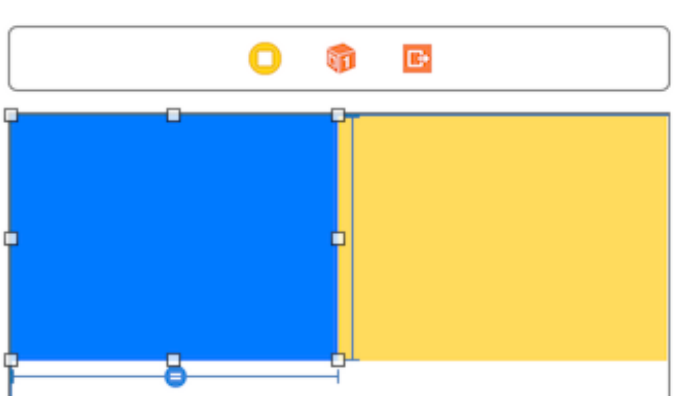
For this example, we will open the Xcode project located in `exercises/addAnotherView/addAnotherView.xcodeproj`.

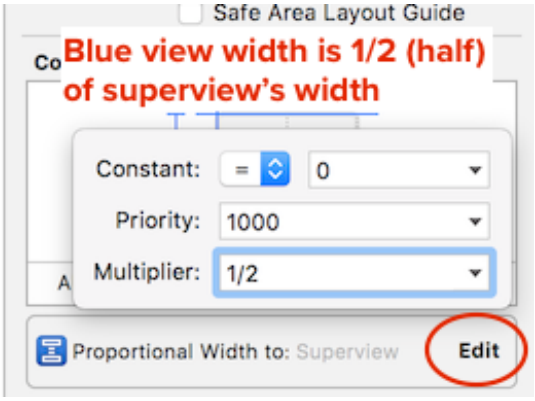
We will open **Main.storyboard** for this example. (Shh don't look into Answer.storyboard yet 👀)

In the view controller, we have two views with constraints placed like this:



The blue view and yellow view have width constraint which their width are half of their superview's (viewcontroller's root view) :

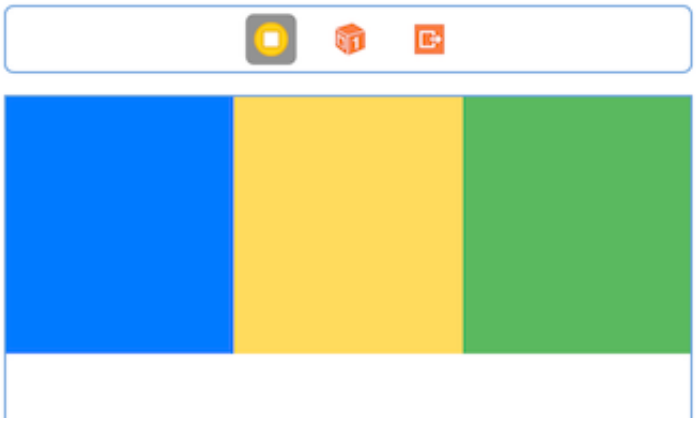




Here's the constraints which are already added:

1. Blue view has proportional width to superview, blue view width = 0.5 * superview width
2. Blue view height is 120 pt
3. Blue view top space to superview is 0
4. Blue view leading space to superview is 0
5. Yellow view has proportional width to superview, yellow view width = 0.5 * superview width
6. Yellow view height is 120 pt
7. Yellow view top space to superview is 0
8. Yellow view trailing space to superview is 0

It looks nice and good, then suddenly your boss / client wants to add a green view at the right of the yellow view and require three of the views (blue, yellow, green) to have same width / height. Your boss / client want the UI to look like this:



How do you proceed with this change? 🤔

I encourage you to do this exercise by yourself first before proceeding below to see the answer so you can appreciate stack view more 😆.

Answer discussion

The changes I made to transform to the three views above are as follows:

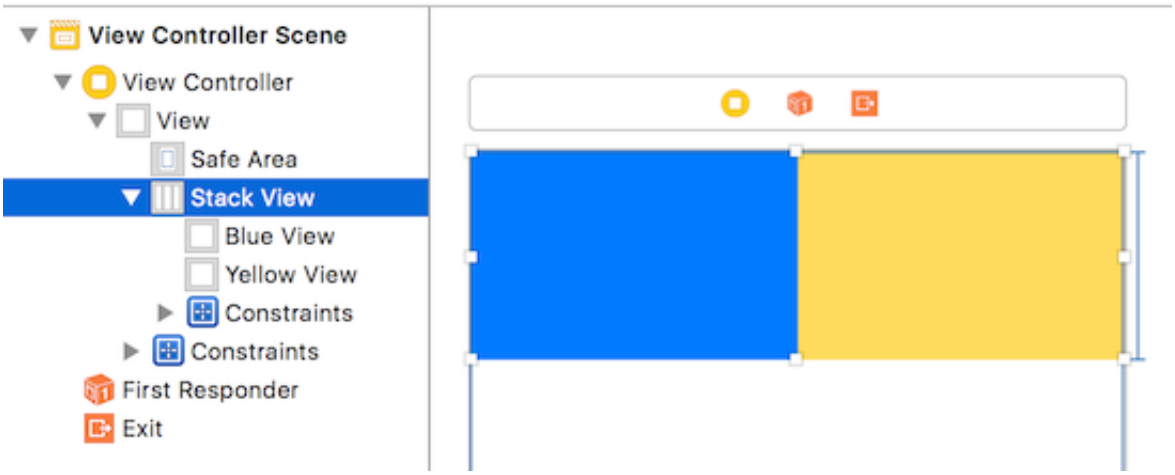
1. Set width of blue view to equal 1/3 of superview's width
2. Set width of yellow view to equal 1/3 of superview's width
3. Remove yellow view trailing constraint
4. Add horizontal spacing constraint between blue view and yellow view, and set its value to 0
5. Drag a new view to the view controller and set its background color to green
6. Set width of green view to equal 1/3 of superview's width
7. Set height of green view to equal 120 pt
8. Set top constraint of green view to superview to 0
9. Add horizontal spacing constraint between green view and yellow view, and set its value to 0

Phew, there are 9 changes I need to make just to add a single view to an existing layout, and this example has only 2 views, imagine how much adjustment it will take to add another view in the middle of a view controller that has multiple buttons, textfields and image views. 😱 (You can check the answer in `Answer.storyboard`)

We will discuss how to achieve the same goal (add green view at the end) using Stack View in the next section.

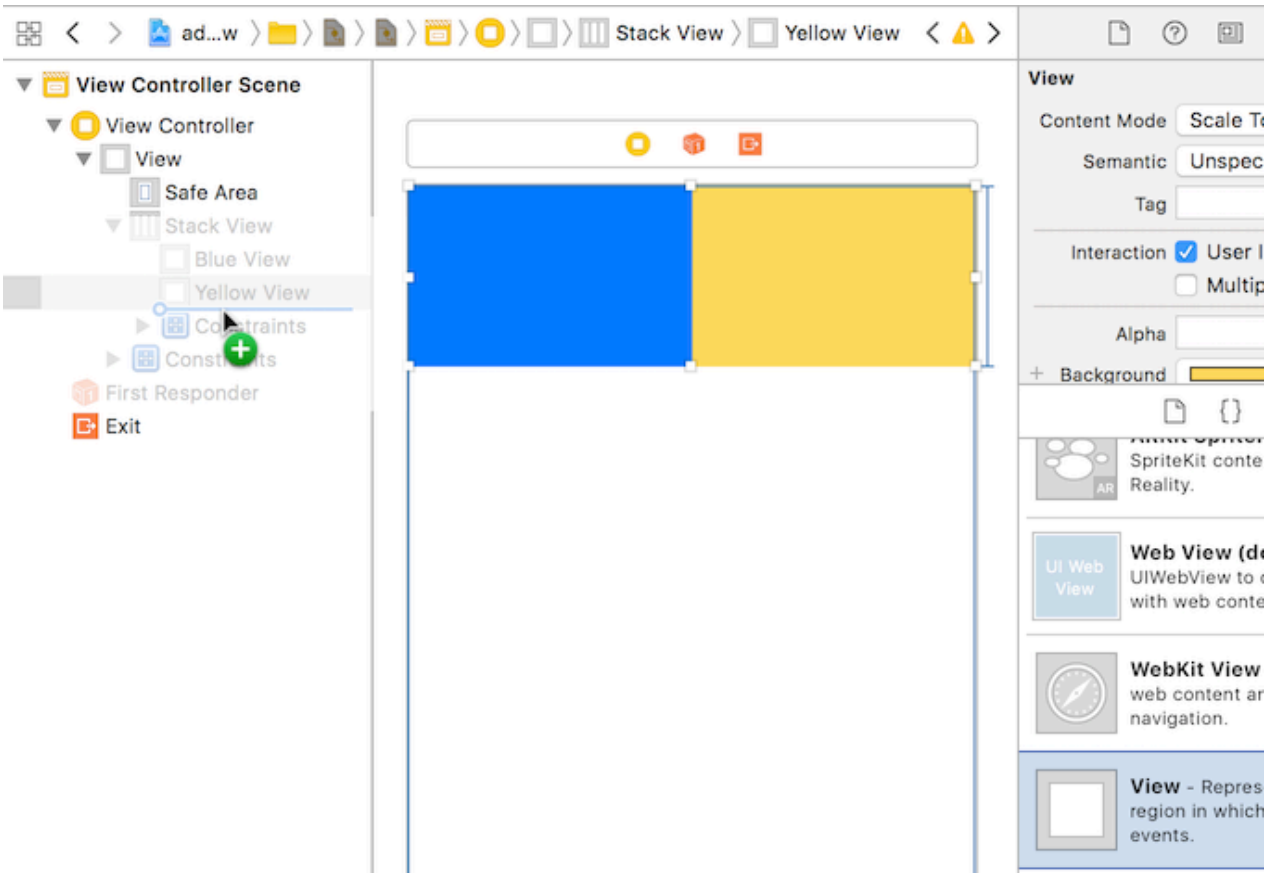
Example using Stack View

In the same Xcode project file (`exercises/addAnotherView/addAnotherView.xcodeproj`), open **Stackview.storyboard** , you will see the same blue and yellow view, but this time they are located inside a stack view.



Note: I have already preset the stack view's **Distribution** property to "Fill Equally" and **Axis** property to "Horizontal" beforehand.

Now, to add the green view that has the same width and height as blue / yellow views, all you need to do is drag a new view into the stack view and place it after the yellow view, like this:

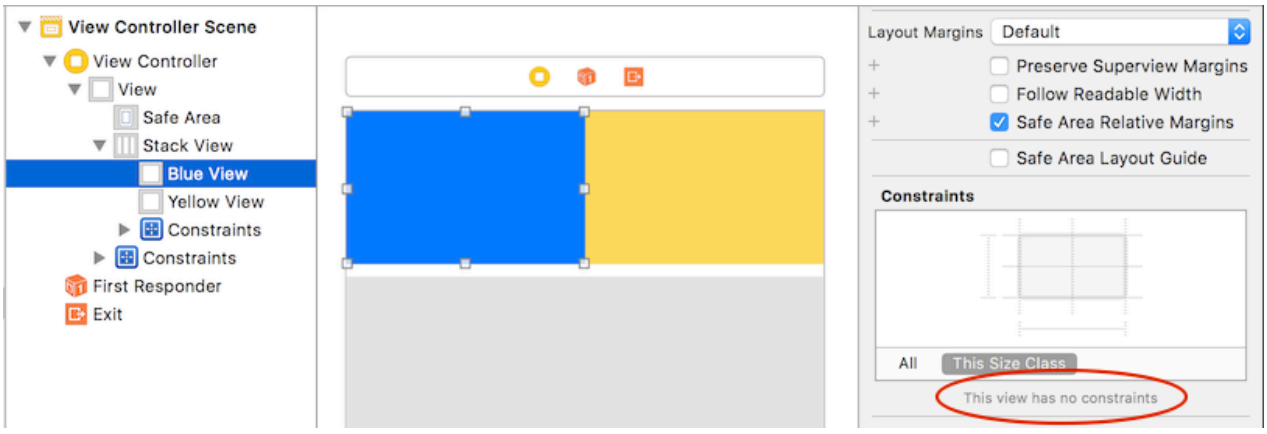


Lo and behold, stack view already took care of the sizing and positioning of the new view, it should look like this:

All that's left is to change the background color of the view to green, we managed to shorten 9 steps into 1 step! That's some awesome stack view magic right there.

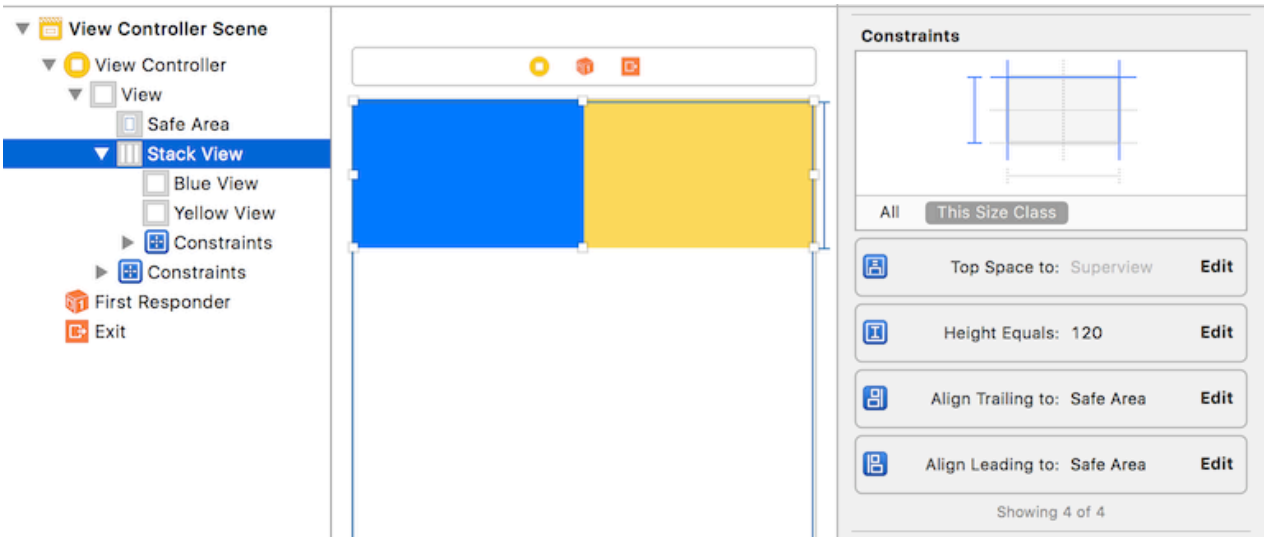
As the stack view's **Distribution** property is set to **Fill equally**, all the child views inside it will have their width divided equally. Stack view also auto aligns child views along its axis, for this example, the stack view axis is horizontal and it will align its child view from left to right. If its axis is vertical, the child views will be aligned from top to bottom.

In this example, there's no constraint set for the child views inside stack view, you can verify it by opening the size inspector of one of the child views :



Usually there's no need for constraint between the stack view and its child views as stack view will adjust position/sizing of its child view based on its child views' intrinsic content size or stack view will fill them equally if you choose "Fill equally" for "Distribution".

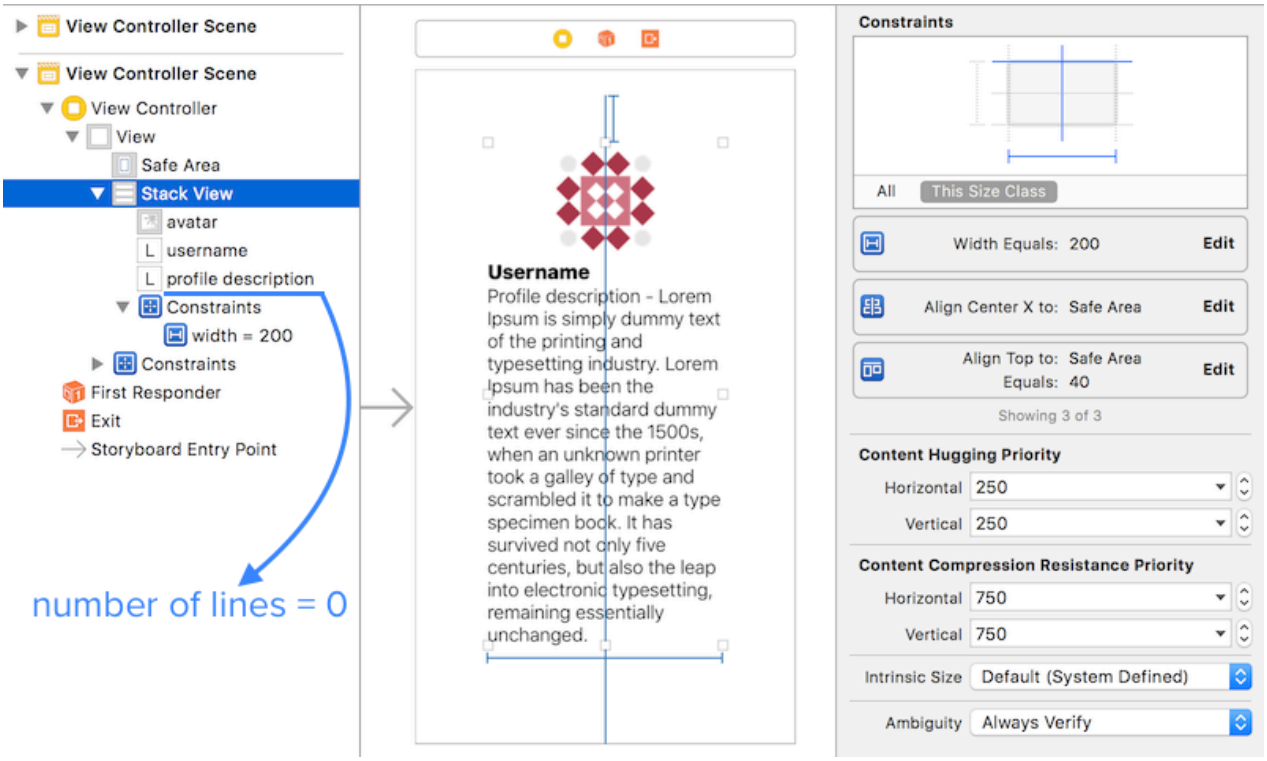
But remember stack view is also a view itself, you will still need to define constraints for the stack view itself so Auto Layout can know where to position the stack view and also the size of it :



We can see that the Stack View has top constraint to super view (root view of view controller), leading/ trailing constraint to safe area and explicit height constraint.

Intrinsic content size of child view in stack view

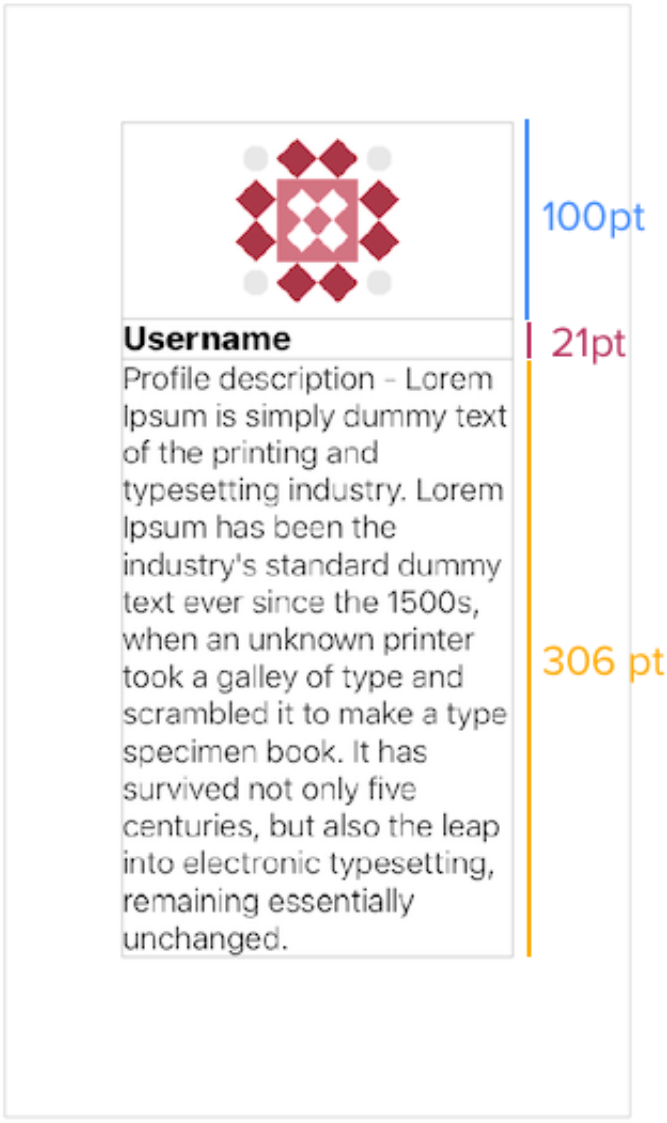
By default (when you drag a new stack view to interface builder), a stack view will use the intrinsic content size of its child views as their size. Child views are stacked next to each other (no spacing between them by default). Here's an example of a vertical stack view with two labels and an image:



Here's the calculation of the position and size of the stack view:

```
// iPhone SE screen size
screenWidth = 320
stackViewWidth = 200

stackViewX = (screenWidth / 2) - (stackViewWidth / 2)
stackViewX = (320 / 2) - (200 / 2)
stackViewX = 60

stackViewY = 40

stackViewHeight = ?
```

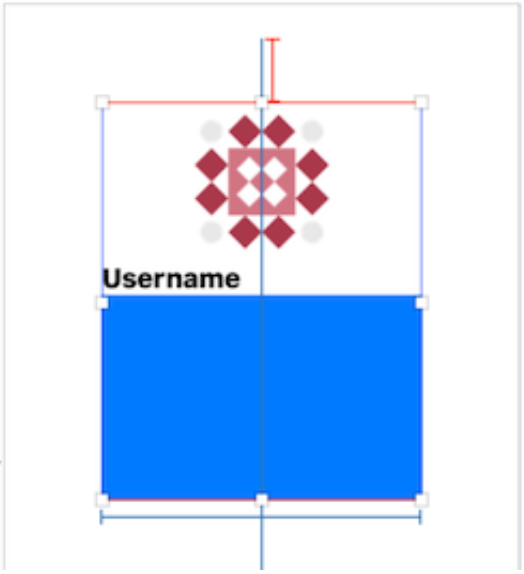
As we didn't define explicit height constraint for the stack view, Auto Layout will use the intrinsic content height of its child views to derive its height:



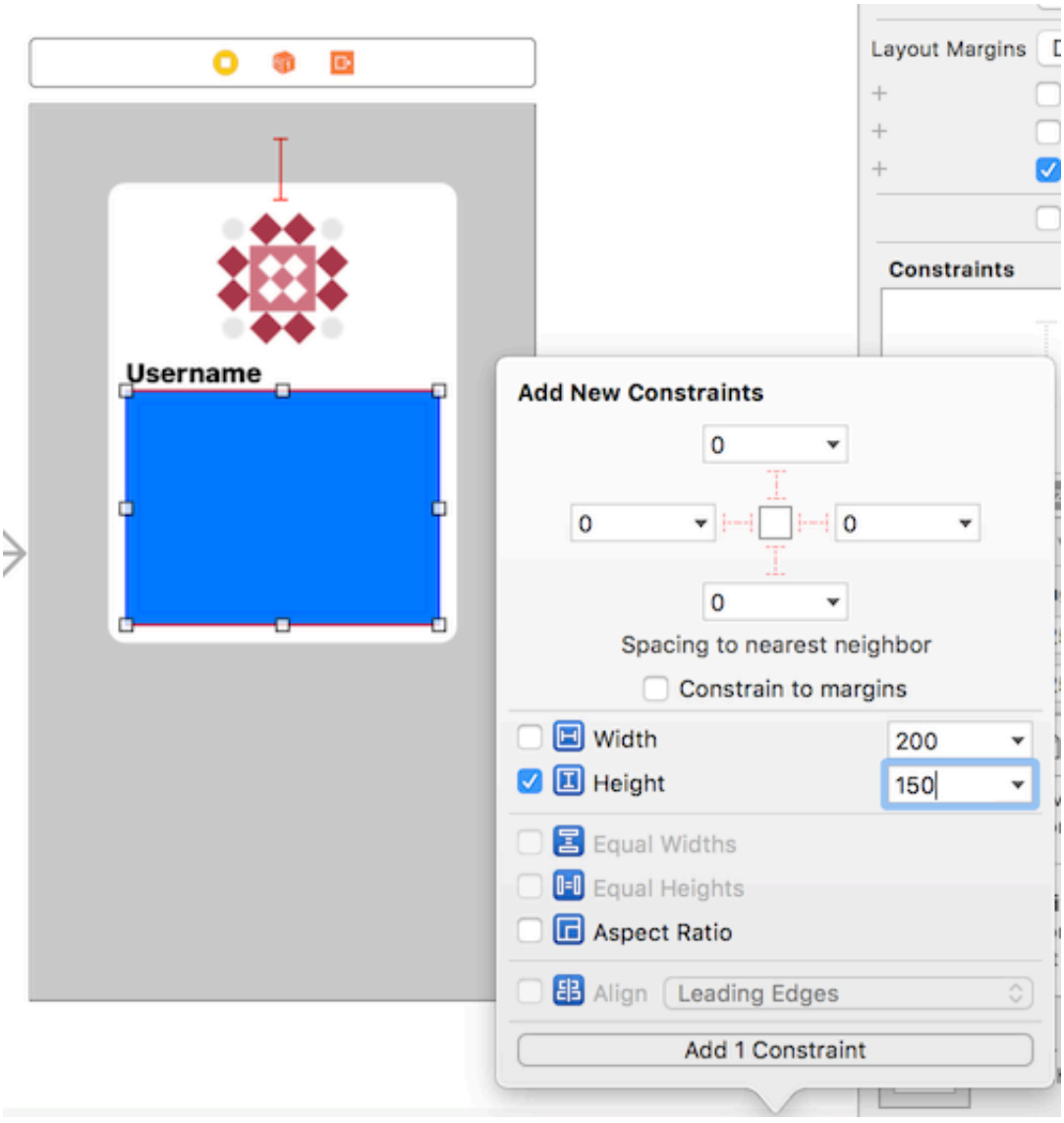
By adding up child views intrinsic height, Auto Layout can calculate the stack view height:

```
stackViewHeight = 100 + 21 + 306
stackViewHeight = 427
```

If we replace the profile description label with an UIView, Xcode would show us red lines as Auto Layout can't calculate the height of stack view because UIView doesn't have an intrinsic content size:



To fix this, we can set an explicit height (eg: 150) for the blue view like this:



After setting the height of blue view, Auto Layout now know how to calculate the height of stack view:

```
blueViewHeight = 150

stackViewHeight = avatarIntrinsicHeight + usernameIntrinsicHeight +
blueViewHeight
stackViewHeight = 100 + 21 + 150
stackViewHeight = 271
```

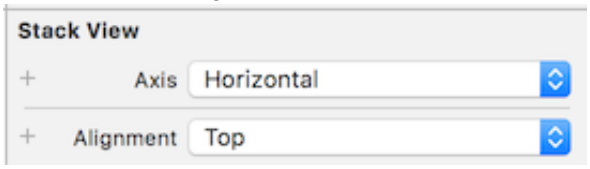
The takeaway of this section is that stack view will use intrinsic content size of its child elements by default, if you are planning to put UI element with no intrinsic content size, you will need to define the width/ height explicitly for that element. (unless you set the distribution to 'Fill Equally', which we discuss in the next section)

Stack view's property

There are few properties in stack view you can tweak, this section will cover `Alignment` , `Distribution` and `Spacing` .

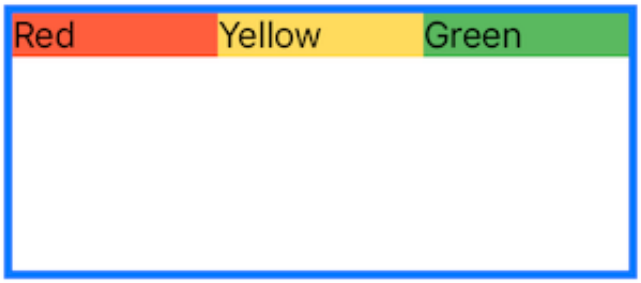
Alignment

For a horizontal stack view, you can choose between '**Fill**', '**Top**', '**Center**' and '**Bottom**' for alignment.



For a vertical stack view, you can choose between '**Fill**', '**Leading**', '**Center**' and '**Trailing**' for alignment.

Top Alignment

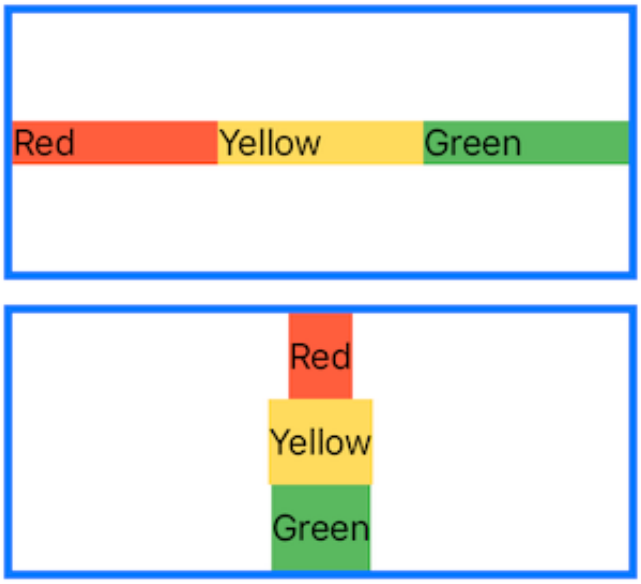


Elements inside will align towards top of the stack view.

Leading Alignment

Elements inside will align towards left of the stack view.

Center Alignment



Elements inside will align on the center of the stack view.

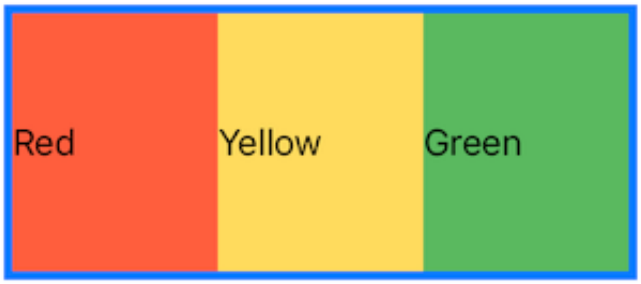
Bottom Alignment

Elements inside will align towards bottom of the stack view.

Trailing Alignment

Elements inside will align towards right of the stack view.

Fill Alignment



For horizontal stack view, elements inside will expand their height to match the stack view height.

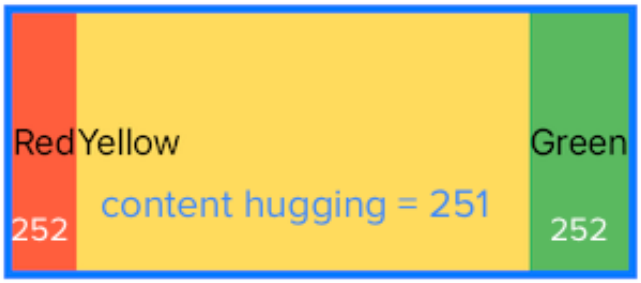
For vertical stack view, elements inside will expand their width to match the stack view width.

Distribution

There's five options for distribution, "**Fill**", "**Fill Equally**", "**Fill Proportionally**", "**Equal Spacing**", "**Equal Centering**".

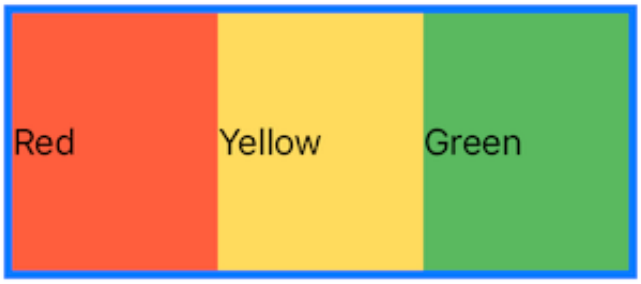
Fill Distribution

Fill distribution will let one of the child view to expand to take up the remaining space while the other views remain at their intrinsic content size. Stack view will select the child view with the lowest content hugging priority to expand, while other views remain at their intrinsic content size.



Fill Equally

All child views will take up equal amount of space in the stack view. Intrinsic content size of child views are ignored.



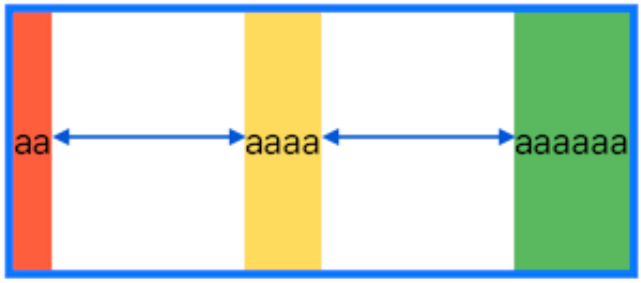
Fill Proportionally

Child views will take up space proportional to their original intrinsic content size to fill up the space of stack view. In other words, label with longer text will take more space while label with shorter text will take less space.

For the examples below, red label text has 2 characters, yellow label text has 4 characters and green label text has 6 characters, making the total characters in the stack view 2 + 4 + 6 = 12. Since red label text has 2 characters, it will occupy 2 / 12 = 16.67% of the width of the stack view, similar calculation can be used on the yellow / green label as well.

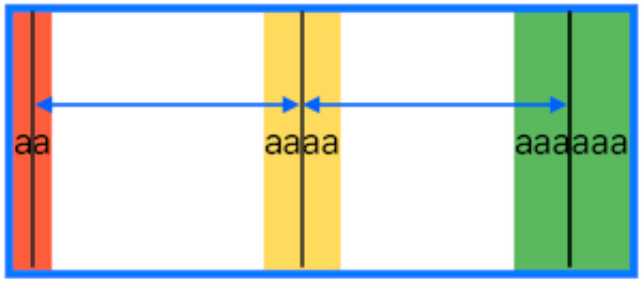
Equal Spacing

Child views will remain their intrinsic content size / explicit defined size, and the spacing between each elements has the same distance.



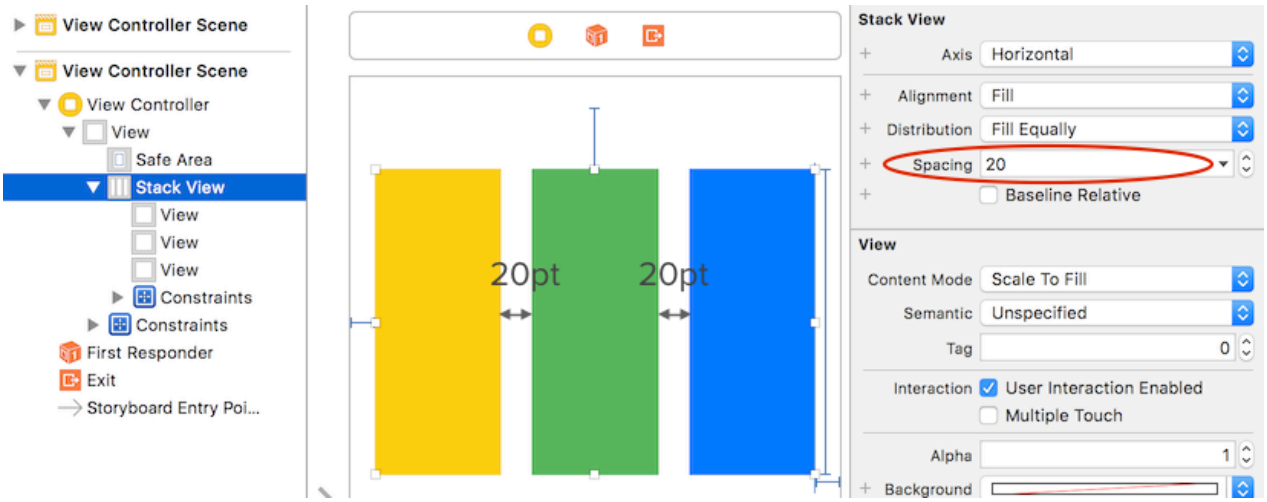
Equal Centering

Similar to Equal Spacing, but this time the spacing between each elements' center point has the same distance instead.



Manual spacing

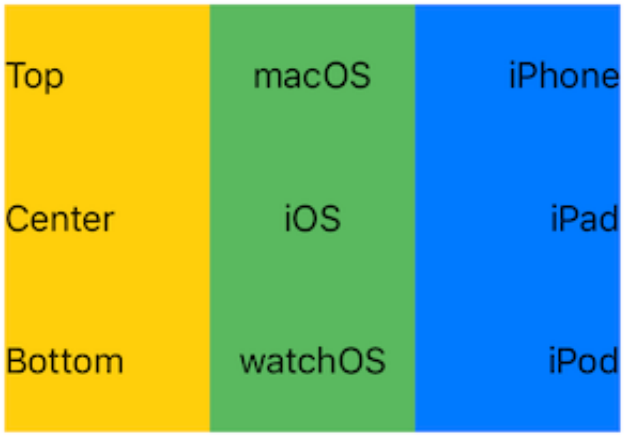
You can manually add some spacing between child elements in a stack view like this :



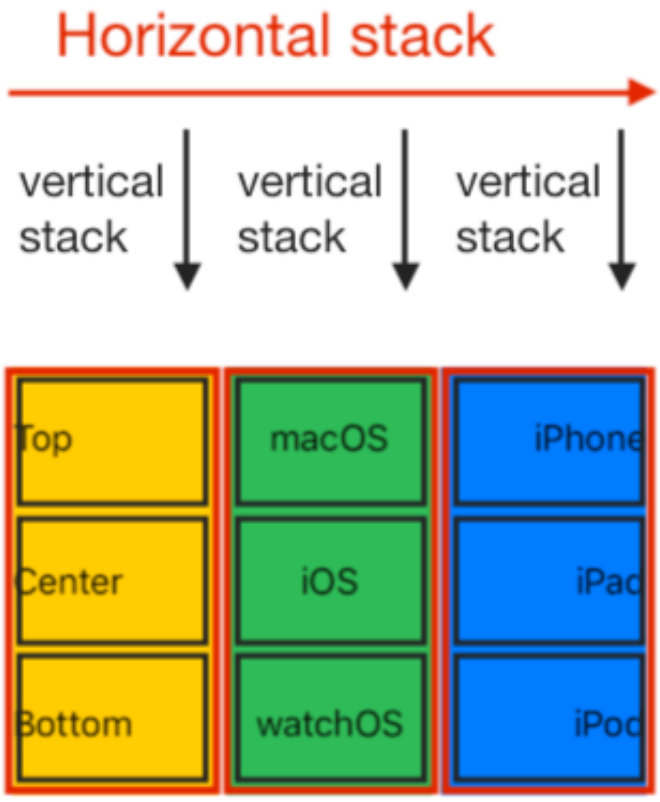
In the above example, the three child views (yellow, green and blue) will share the same width and has 20pt spacing between them.

Nesting stack views

You can put a stack view inside a stack view for better arrangement. Example:



In the above example, there's an outer horizontal stack view, the horizontal stack view distribution is set to "Fill Equally" and has three views (yellow, green and blue) inside it. Inside of each view, there's a vertical stack view with distribution set to "Fill Equally" and contain three labels.



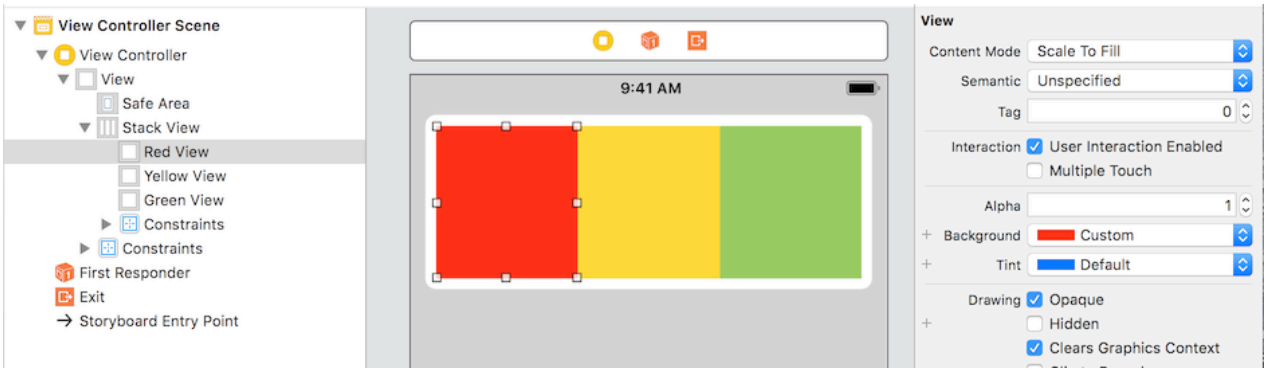
Nesting stackviews are very useful when dealing with complex layout.

You can open `exercises/nestedStackView/nestedStackView.xcodeproj` to see this example and how its constraints are set.

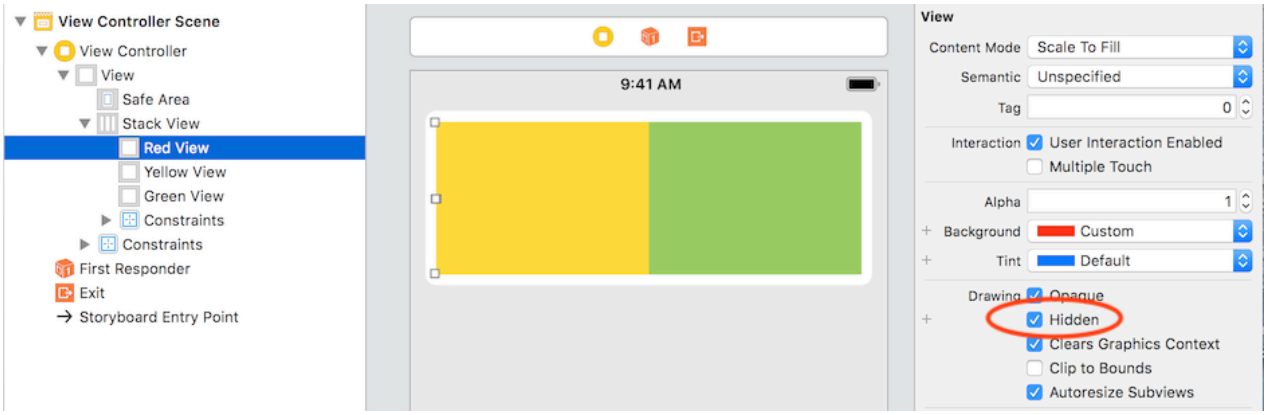
Stack view's automatic arrangement

One powerful feature of stack view is that it will auto-arrange the layout of its childviews when one of the childview visibility is set to hidden or visible. (eg: `childview.isHidden = true`)

As an example, let's create a stack view with horizontal axis and fill equally, then we place three subviews inside like this :



If we set the red view to hidden, then the stack view will rearrange the remaining visible child views (yellow view and green view) like this :



You can do this programmatically and the stackview will auto arrange its childviews as well. This is particularly useful when you want to keep the child view in the view hierachy but want to change its visibility depending on situation. Instead of adding and removing the view repeatedly, we can simply just toggle its visibility.

One thing to note is that you can't set a background color to stack view (even if you set it in interface builder, it won't show). This is because Stack View is a non-drawing view, meaning that `drawRect()` function will not be called (stack view remains invisible on screen) thus its background color is ignored. You have to place the stack view inside a view and set the outer view background color if you really want a background color.

Summary

In short, stack view is a really great tool that helps us tackle complex layout as it allows us to add / edit / delete elements easily without having to usher through a cluster of constraints.

12 - Animating views with constraint

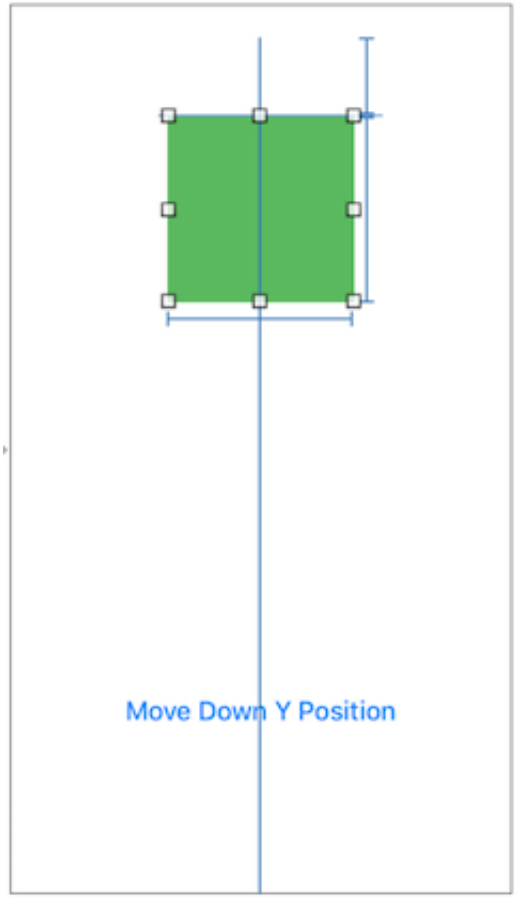
Why it is not a good idea to modify frame directly

Prior to Auto Layout / Constraint being introduced, animation is often done by changing the frame / size of a view like this:

```
// move down green view by 50 pt from its current position
UIView.animate(withDuration: 0.5, animations: {

  self.greenView.frame =  CGRect(x: self.greenView.frame.origin.x,
                                 y: self.greenView.frame.origin.y + 50,
                                 width: self.greenView.frame.size.width,
                                 height: self.greenView.frame.size.height)
})
```

Code above will move the green view 50 pt below its current position. It works well if the view is initialized using frame / coordinate system.

However in Auto Layout system, it is not a good idea to modify the frame of a view that has constraint defined. Well, you could directly alter the frame (position / size) of a view that has constraint placed using code, like this:




```
// move down green view by 50 pt from its current position
@IBAction func moveDownYButtonTapped(_ sender: Any) {
  self.greenView.frame =  CGRect(x: self.greenView.frame.origin.x,
                                 y: self.greenView.frame.origin.y + 50,
                             width: self.greenView.frame.size.width,
                            height: self.greenView.frame.size.height)
}
```

The green view's position will change if you press the button to move down the green view. But when there's event that trigger Auto Layout refresh such as rotating phone orientation, views which are using constraint will calculate their position / size again using the defined constraints, thus losing any direct change made to their frame.

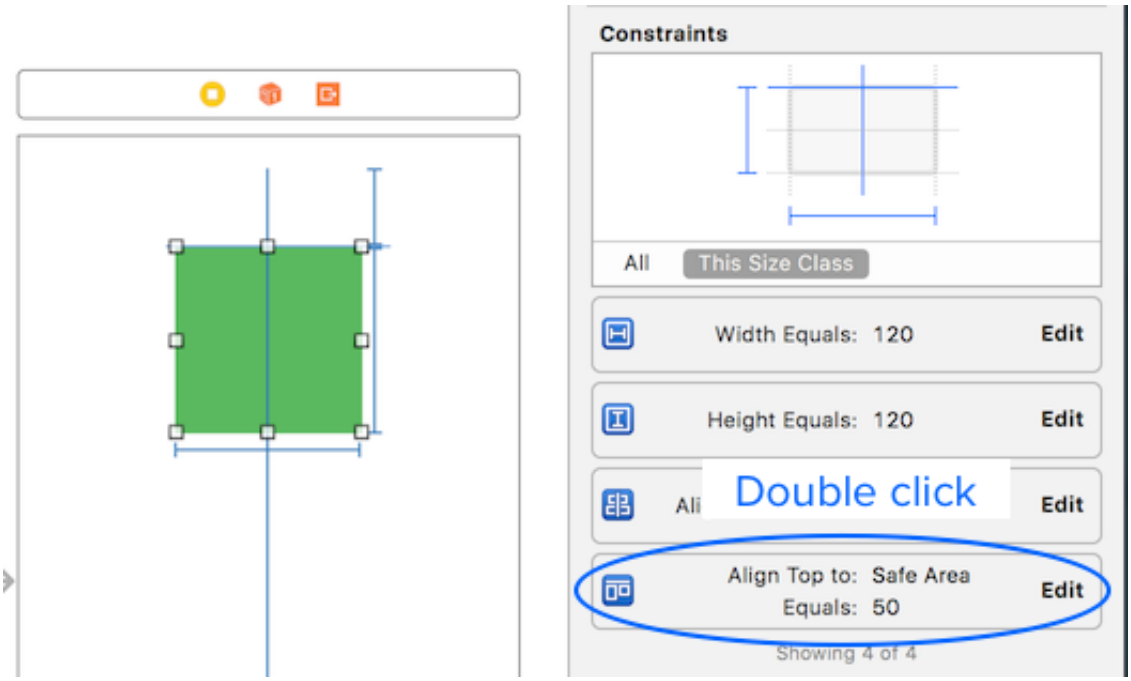
You can watch the video located in `videos/12/problemWithModifyingFrame.mp4` to see the demonstration of this problem, notice that the green view position is reset back to top after phone orientation rotates.

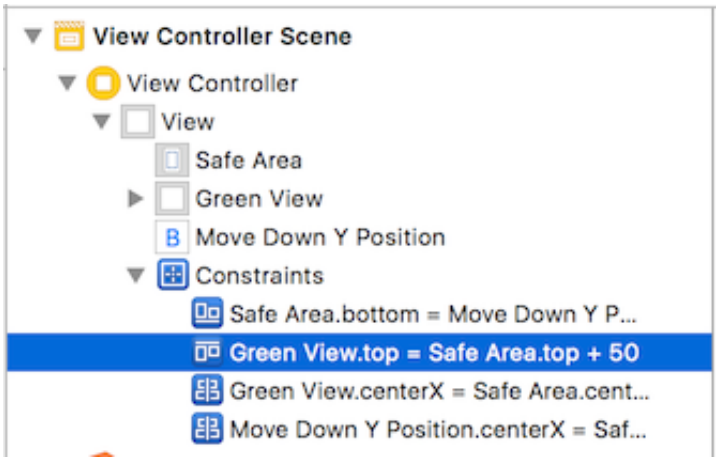
If you want to change the position / size of a view that has constraint placed, **it is advised to change the constraint value** instead of directly altering its frame. This way, even if Auto Layout refresh due to phone rotation, Auto Layout will still use the latest constraint value to calculate the frame.

We will explain how to do it in the next section.

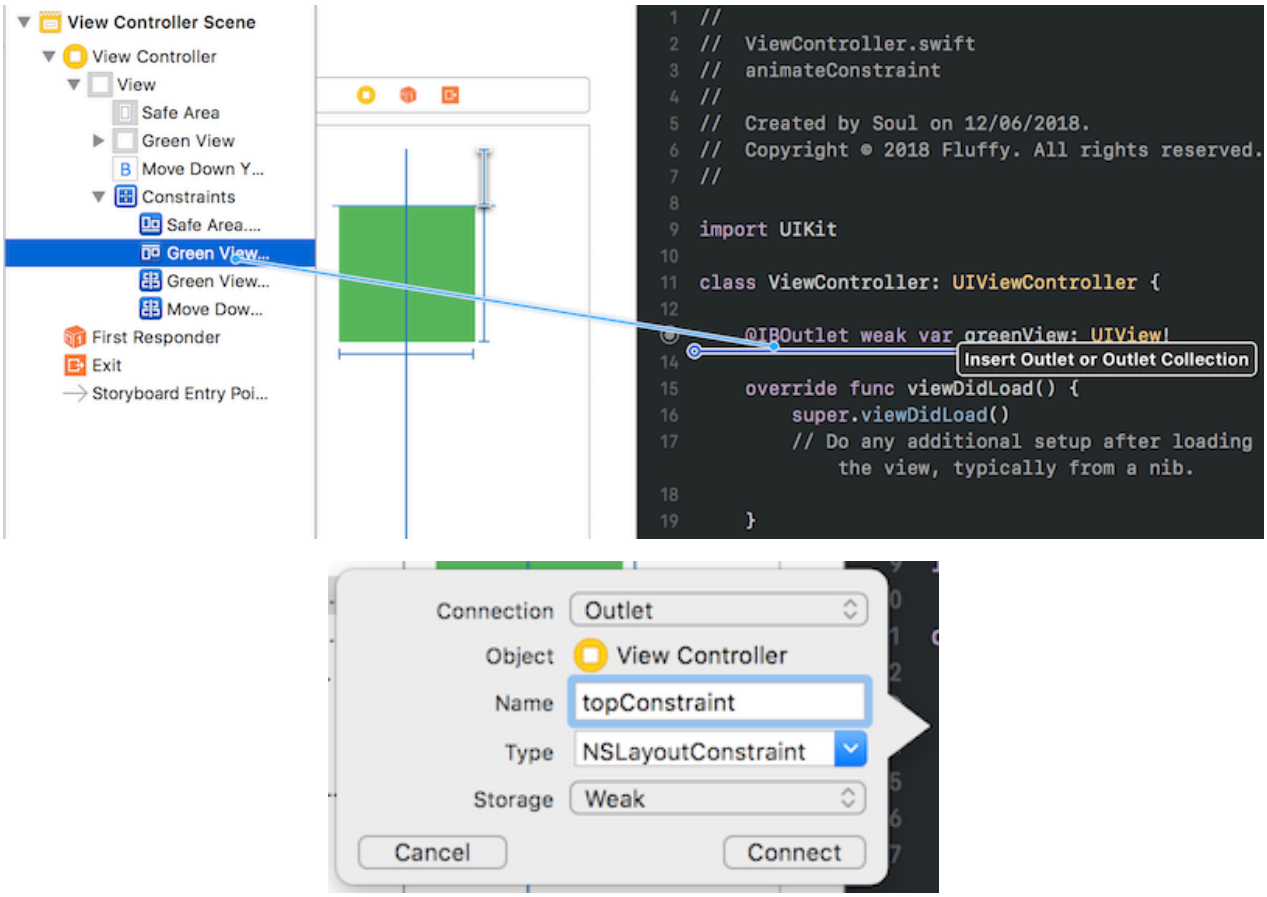
Modifying constraint value in code

To modify the value of constraint, we will first have to create an **IBOutlet** for the constraint. First, select the view and open the **Size inspector**, then double click the constraint you want, you should see the constraint is highlighted in the **Document outline** section.





Then press and hold "**Control**" + **Drag** it to the code in Assistant View to create an outlet like you would do with normal UI elements:



Then to modify the value of the constraint, we change its **constant** property like this :

```
// move down green view by 50 pt from its current position
@IBAction func moveDownYButtonTapped(_ sender: Any) {
    self.topConstraint.constant += 50
}
```

Now when the button is pressed, the green view top constraint constant value will increase by 50 pt, causing green view to move downwards. Directly changing the constraint value will make the view jumps to new position abruptly, we will look into how to animate the position change smoothly in the next section.

Animating constraint value

To animate the change of constraint value, we can wrap the change inside a `UIView.animate` block like this:

```
// call 'layoutIfNeeded()' on the parent view of the view you want to animate
// This is to ensure all the pending constraint/layout change for the
childviews inside the parent view is completed before animation starts
self.view.layoutIfNeeded()

// starts animation
UIView.animate(withDuration: 0.5, animations: {
    // make changes on constraints here
    self.topConstraint.constant += 50

    // call 'layoutIfNeeded()' on the parent view of the view you want to
animate
    // calling 'layoutIfNeeded()' on a view will adjust all of its childviews
frame to satisfy the (updated) constraints
    self.view.layoutIfNeeded()
}
```

Apple recommends calling `layoutIfNeeded()` on the parent view (of the view that you want to animate) **before** the animation block to ensure that all pending layout changes have been completed.

Inside the animation block, you will need to call `layoutIfNeeded()` on the parent view again after changing constraint value, else the view will just change its constraint abruptly without animation.

This snippet is taken from the old Apple Auto Layout guide (Apple has removed it from their website, you can read it on the wayback machine link here, https://web.archive.org/web/20150315203355/https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/AutolayoutPG/AutoLayoutbyExample/AutoLayoutbyExample.html#//apple_ref/doc/uid/TP40010853-CH5-SW15).

Yep, that's all you need to do to animate the constraint, `UIView.animate` does make animation simpler. You can see the finished animation in `videos/12/animation.mp4`.

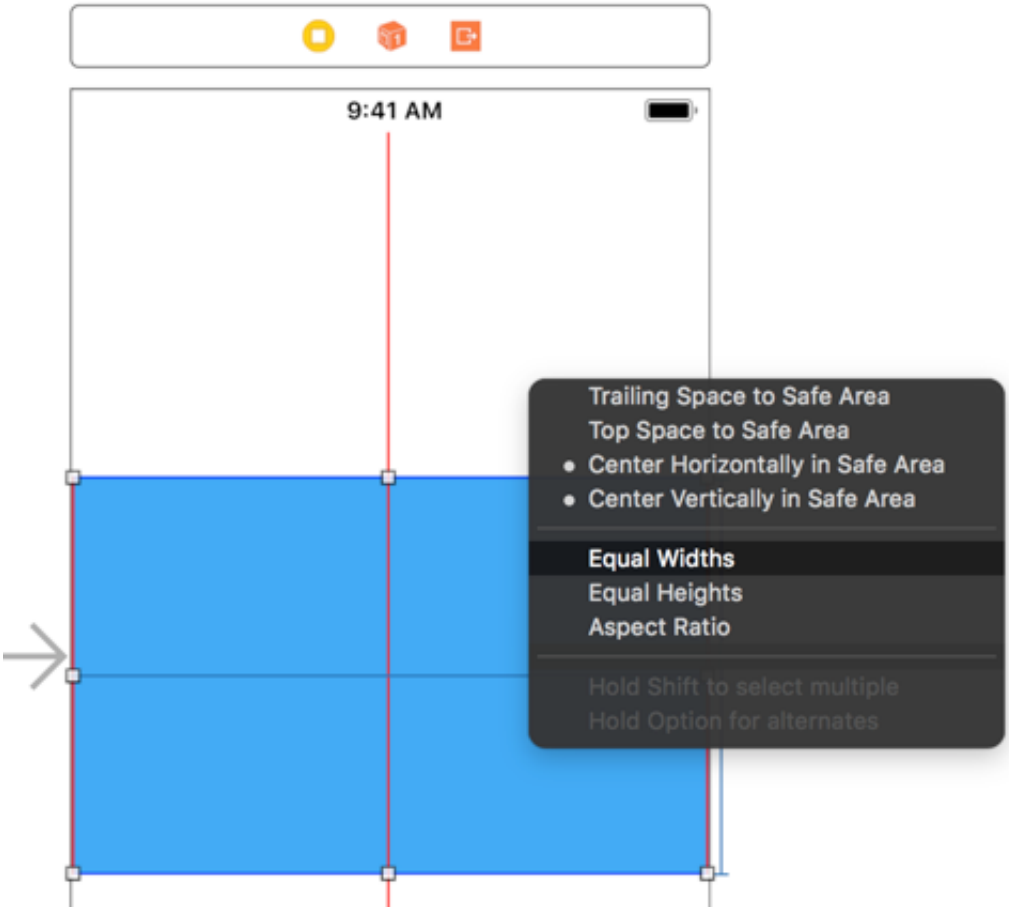
13 - Constraint tips & tricks

Proportional constraint

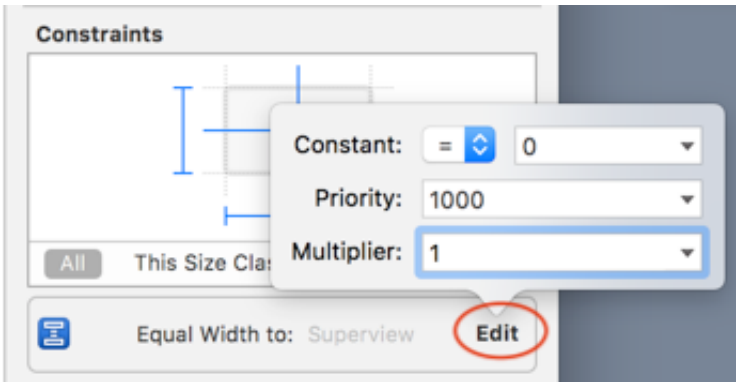
Say if you want to make a view to be 1/3 of width of its superview, how would you proceed?

When editing constraint, there is a "**multiplier**" field.

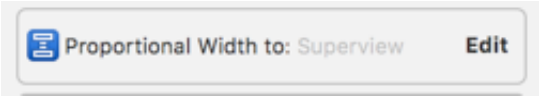
Hold `Control` + Drag the view to its superview, and select 'Equal Width'.

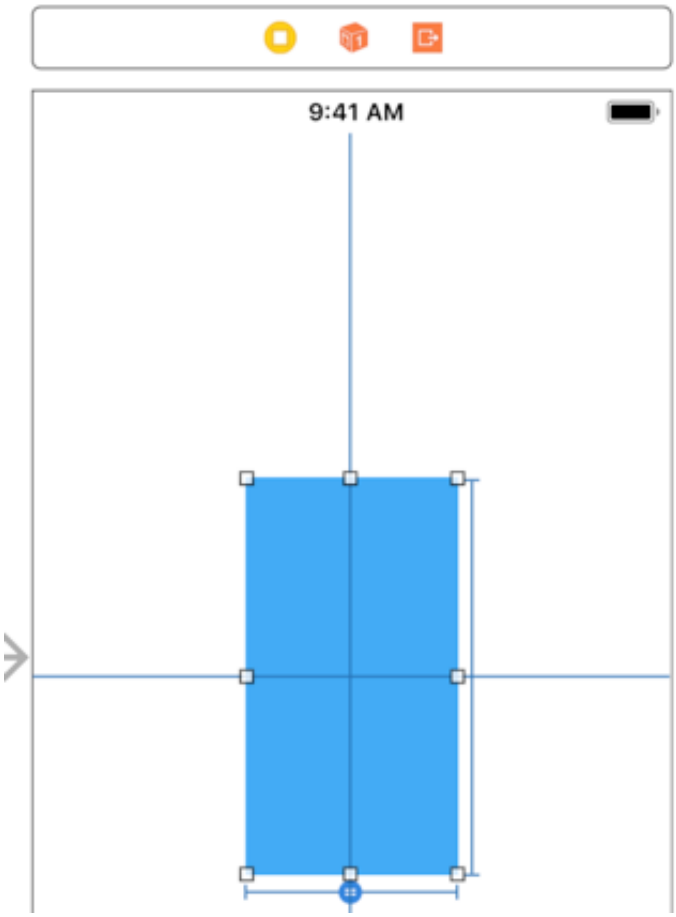


A new 'Equal Width' constraint will be created. Click '**Edit**' on the constraint, and change the **Multiplier** value to '**1/3**' and press enter. This will make the blue view to have 1/3 width of its superview.



You will see the constraint name changed from 'Equal Width' to 'Proportional Width'.





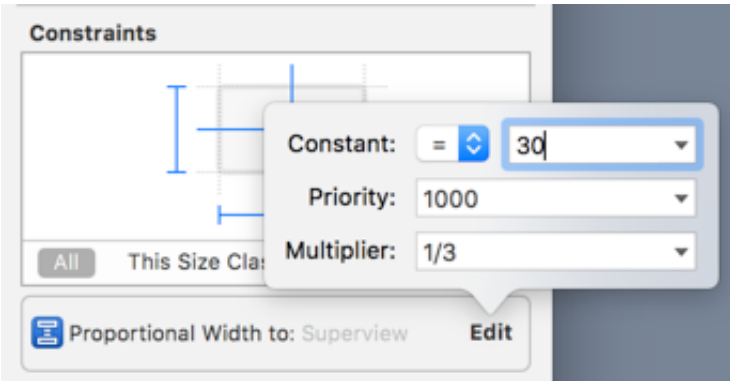
You can use 1/4 for one-fourth width, 2 for double the width, etc.

Constant value on top of proportional constraint

Continuing from previous proportional constraint, we can edit the constant value to make something like

```
Blue view's width = (1/3 * super view's width) + 30pt
```

This would add another 30pt to the 1/3 width.

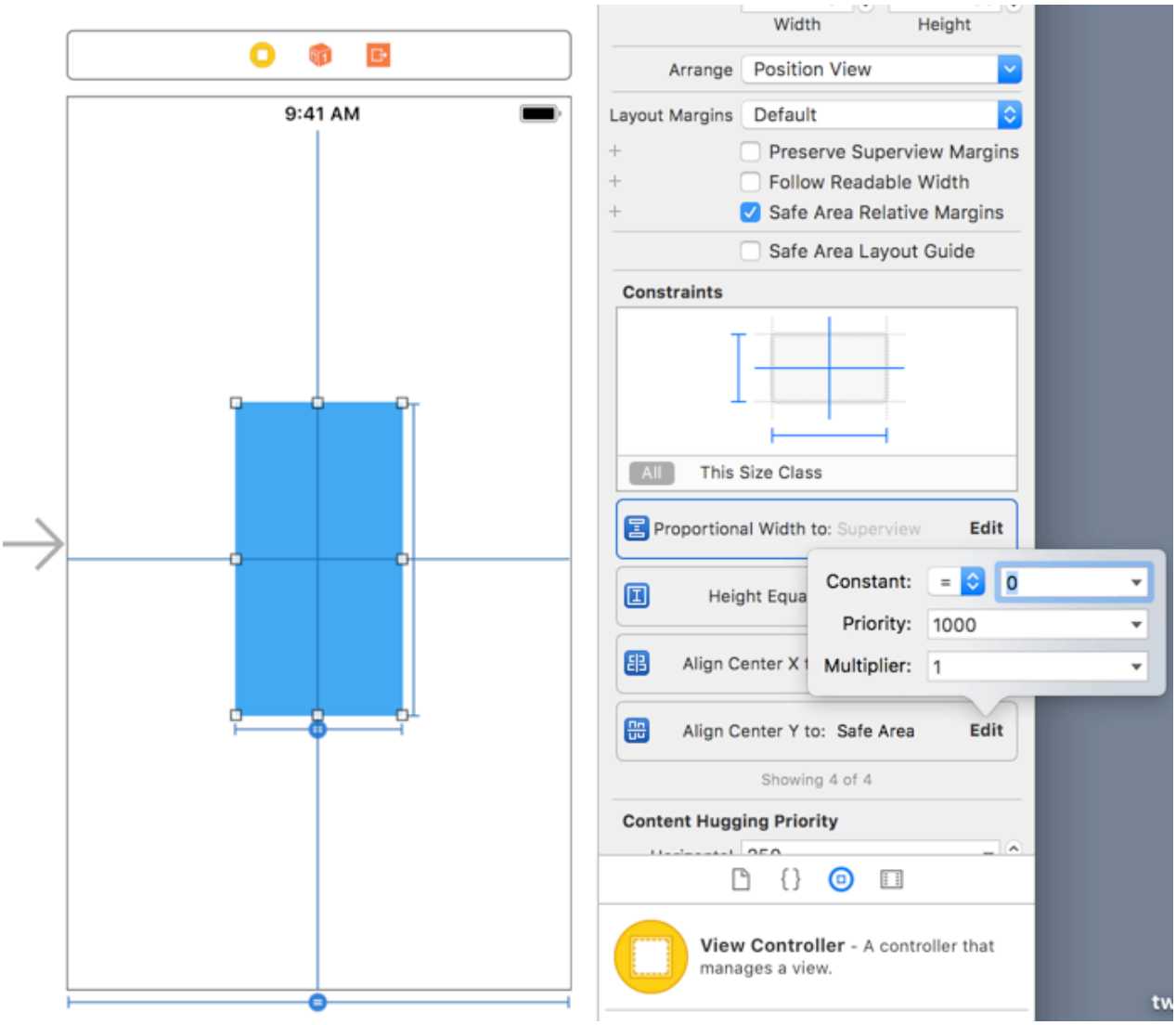


Constant value on top of centering constraint

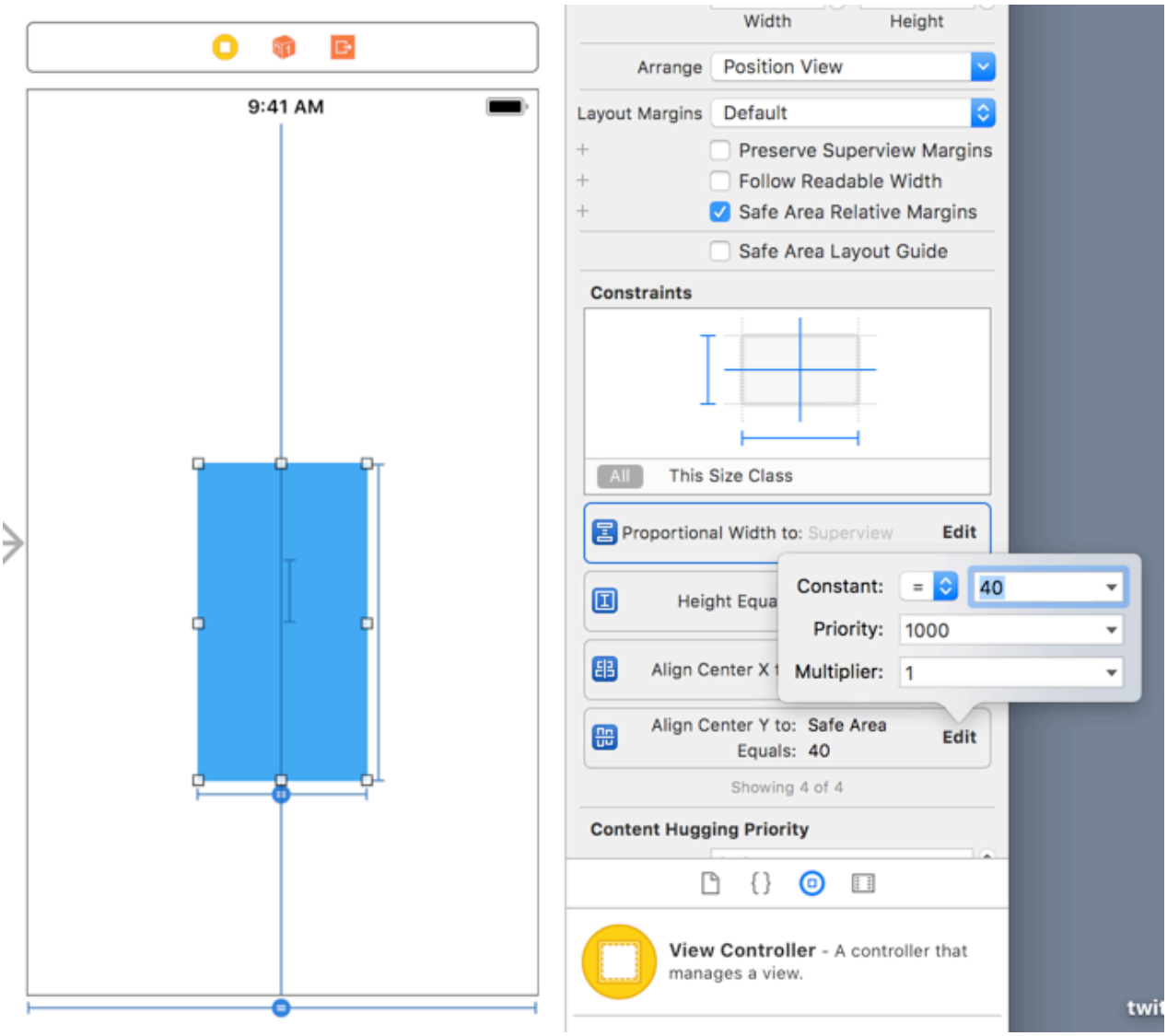
Similar to proportional width/height constraint introduced earlier, we can add constant value to centering constraint.

This allows us to achieve something like : centering blue view's vertical center 40 pt below superview's vertical center.

Vertical centering :



Vertical Centering with 40pt constant :



If you want to make the blue view's vertical center to 40pt **above** the superview's vertical center, you can put **-40** as the constant.

14 - What is Size Classes and how to use it to create adaptive layout

As the variation of iOS devices has increased over the years (iPhone SE, iPhone 8, iPhone 8 Plus, iPad Mini, iPad Pro 10, 11, 12 inch!), with 2 possible orientation (portrait, landscape) and the introduction of slide over and split screen feature in iPad, there's 300+ possible different dimension for an app to run on.

Using Auto Layout alone would be hard to cater for these different dimension, as the available screen space varies greatly for the same constraint. Of course you could programmatically check for different screen size and adjust the constraint's constant, but then there would be a lot of screen sizes to check!

To reduce the number of screen variations, Apple has grouped these possible screen sizes using two traits, namely **width** (horizontal) and **height** (vertical). There's two values for each of these traits, **compact** and **regular** . Theres 4 type of variations :

1. Compact Width + Compact Height
2. Compact Width + Regular Height
3. Regular Width + Compact Height
4. Regular Width + Regular Height

These 4 combinations are called as **Size Classes**.

Here's the list of size classes categorization.

- In the tables below, "**iPad**" consists of iPad mini and iPad. "**iPad Pro**" consists of all iPad Pro models (9.7", 10.5") except the 12.9" model.

Portrait mode

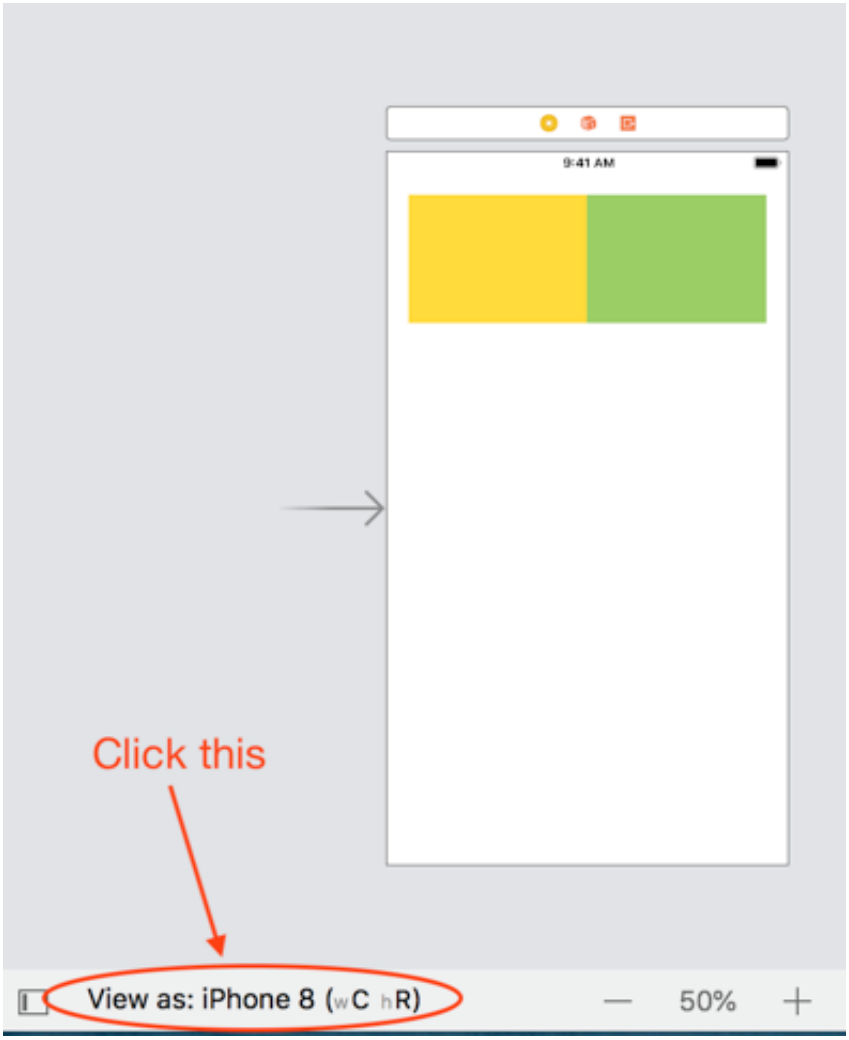
Device	Orientation	Split View	Width Size Class (w)	Height Size Class (h)
iPhone SE	Portrait	-	Compact (C)	Regular (R)
iPhone 8	Portrait	-	Compact (C)	Regular (R)
iPhone 8 plus	Portrait	-	Compact (C)	Regular (R)
iPhone Xs	Portrait	-	Compact (C)	Regular (R)
iPhone Xs Max / XR	Portrait	-	Compact (C)	Regular (R)
iPad	Portrait	-	Regular (R)	Regular (R)
iPad Pro	Portrait	-	Regular (R)	Regular (R)
iPad Pro 12.9"	Portrait	-	Regular (R)	Regular (R)
iPad	Portrait	1/3	Compact (C)	Regular (R)
iPad Pro	Portrait	1/3	Compact (C)	Regular (R)
iPad Pro 12.9"	Portrait	1/3	Compact (C)	Regular (R)
iPad	Portrait	2/3	Compact (C)	Regular (R)
iPad Pro	Portrait	2/3	Compact (C)	Regular (R)
iPad Pro 12.9"	Portrait	2/3	**Regular (R)**	Regular (R)

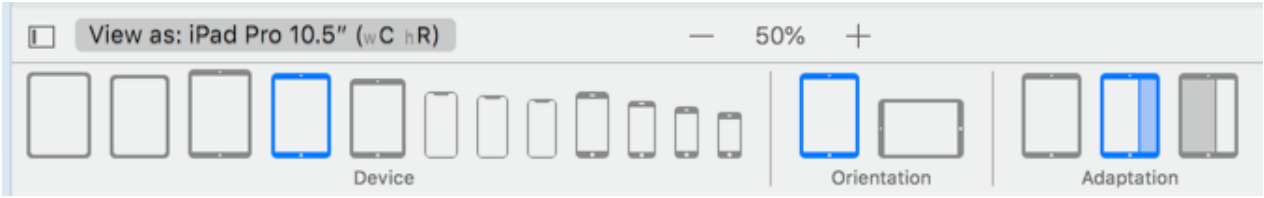
Landscape mode

Device	Orientation	Split View	Width Size Class (w)	Height Size Class (h)
iPhone SE	Landscape	-	Compact (C)	Compact (C)
iPhone 8	Landscape	-	Compact (C)	Compact (C)
iPhone 8 plus	Landscape	-	**Regular (R)**	Compact (C)
iPhone Xs	Landscape	-	Compact (C)	Compact (C)
iPhone Xs Max / XR	Landscape	-	**Regular (R)**	Compact (C)
iPad	Landscape	-	Regular (R)	Regular (R)
iPad Pro	Landscape	-	Regular (R)	Regular (R)
iPad Pro 12.9"	Landscape	-	Regular (R)	Regular (R)
iPad	Landscape	1/3	Compact (C)	Regular (R)
iPad Pro	Landscape	1/3	Compact (C)	Regular (R)
iPad Pro 12.9"	Landscape	1/3	Compact (C)	Regular (R)
iPad	Landscape	1/2	Compact (C)	Regular (R)
iPad Pro	Landscape	1/2	Compact (C)	Regular (R)
iPad Pro 12.9"	Landscape	1/2	**Regular (R)**	Regular (R)
iPad	Landscape	2/3	Regular (R)	Regular (R)
iPad Pro	Landscape	2/3	Regular (R)	Regular (R)
iPad Pro 12.9"	Landscape	2/3	Regular (R)	Regular (R)

Phew, that's a huge list. No worries, you don't need to memorize the whole list, you just need to design around the 4 possible size classes as Apple suggested. When the app opens in different device/orientation/split view combination, the corresponding size classes of your design will be used.

On the bottom Interface Builder / Storyboard, there's a bar showing the current device shown, notice the **wC hR** in the bracket? It means that the current device's size classes is **C**ompact **w**idth and **R**egular **h**eight. You can switch around different devices and the size classes will change.

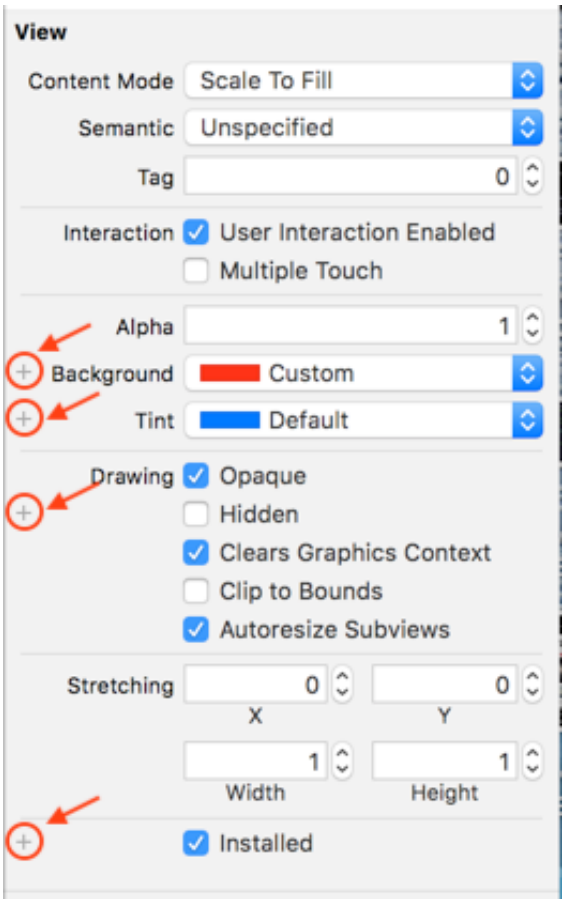




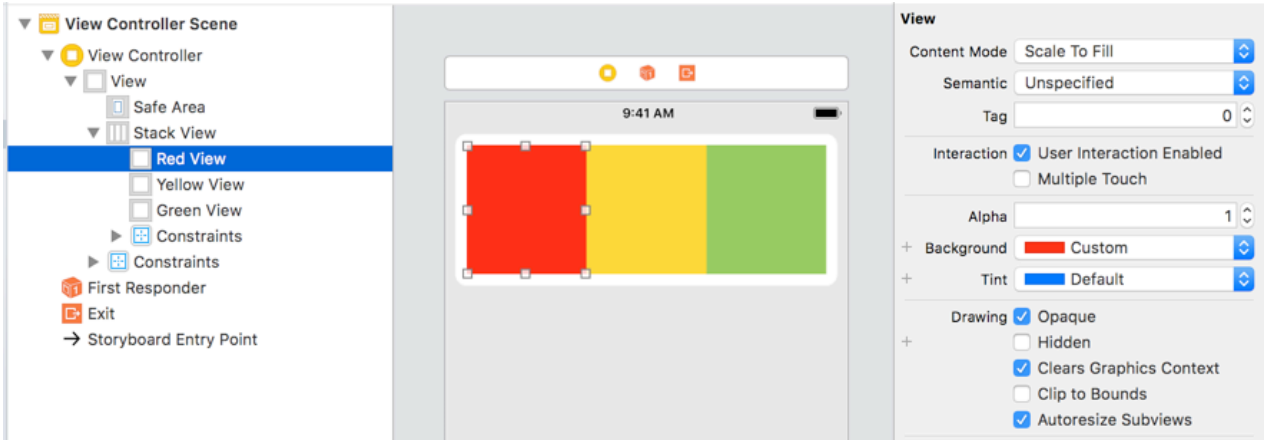
Now we know what size classes are, in the next section we are going to learn how to apply different design to different size classes.

Applying different design to different size classes

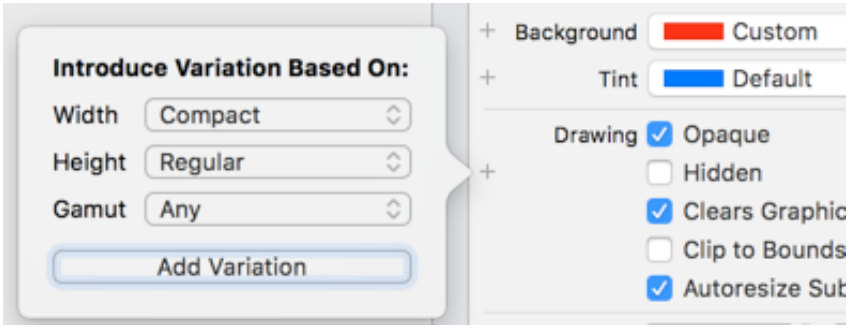
When you open the attribute inspector of a view, notice that some attribute has a small '**+**' icon beside it like this :



By clicking the '+' icon, we can add variation for the attribute on different size classes. Let's try it with the **Hidden** property. Say we have a Stack View (Distribution: Fill Equally) which contains three child view like this :

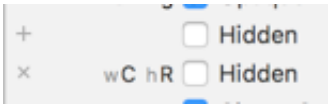


We want to hide the red view when the app is shown on a portrait iPhone (size class **w**idth = **C**ompact, **h**eight = **R**egular). Click on the '+' button beside the **Hidden** property.



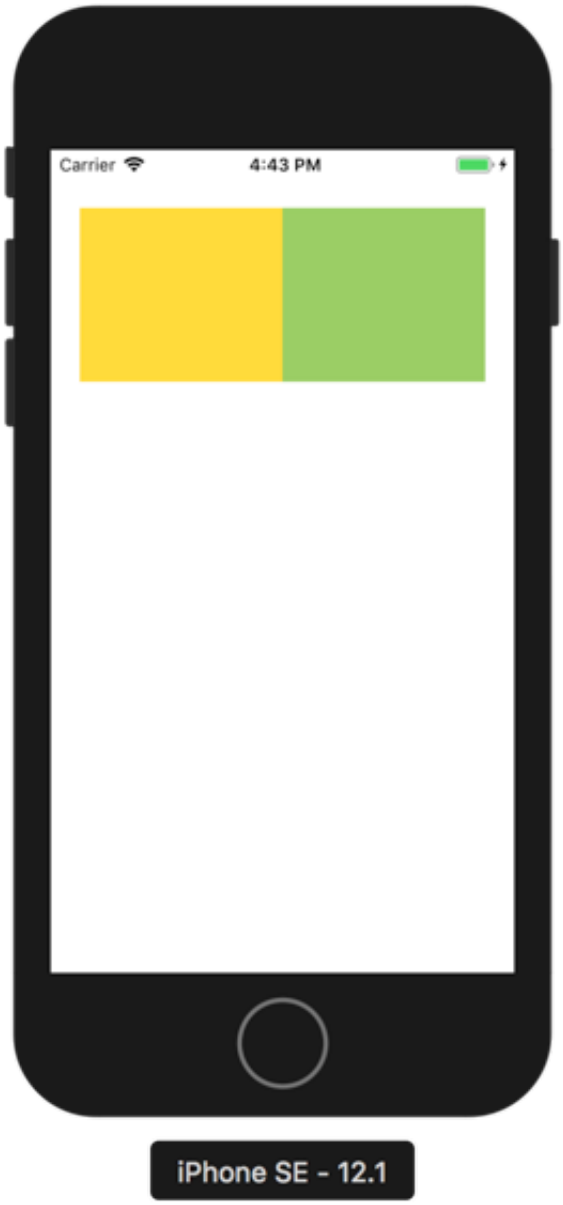
Select 'Compact' for Width, 'Regular' for Height, and click 'Add Variation'. The Gamut means the color profile of the device, which affects the color displayed on screen, we will ignore this trait as we only want to variate on screen resolution (choose 'Any').

Then you will see there's two **Hidden** property variation.



Check the Hidden checkbox for w**C** h**R**, then the red view will be hidden on screen size that fits to the compact width and regular height criteria. (eg: iPhone portrait, iPad 1/3 split view mode).

Run the app on an iPhone simulator, press Command + Left / Right Arrow Key to rotate the orientation. You would see layout like this :

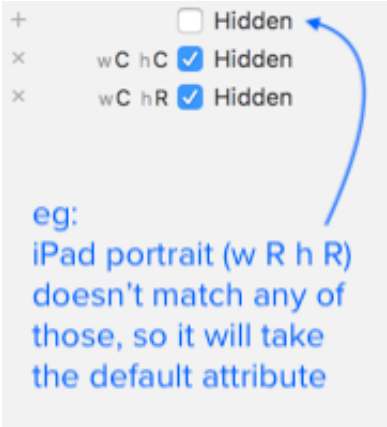


The red view is hidden on iPhone portrait orientation (Compact Width, Regular Height).

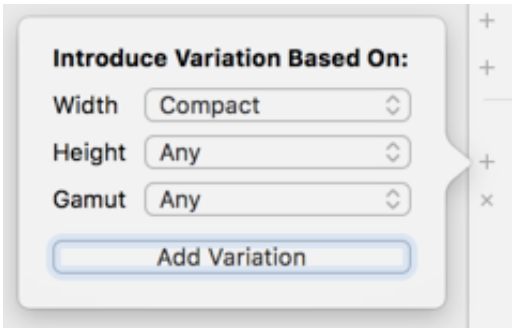
The red view is shown on iPhone landscape orientation (Compact Width, Compact Height).

We can add more variation by clicking the '+' button if we want to.

If there's no matching size classes variation, the default attribute will be used. Say we have defined two size classes (wC hC and wC hR) for an attribute, if we open the app on iPad portrait mode (wR hR), it will use the default attribute (without size classes mentioned in front).



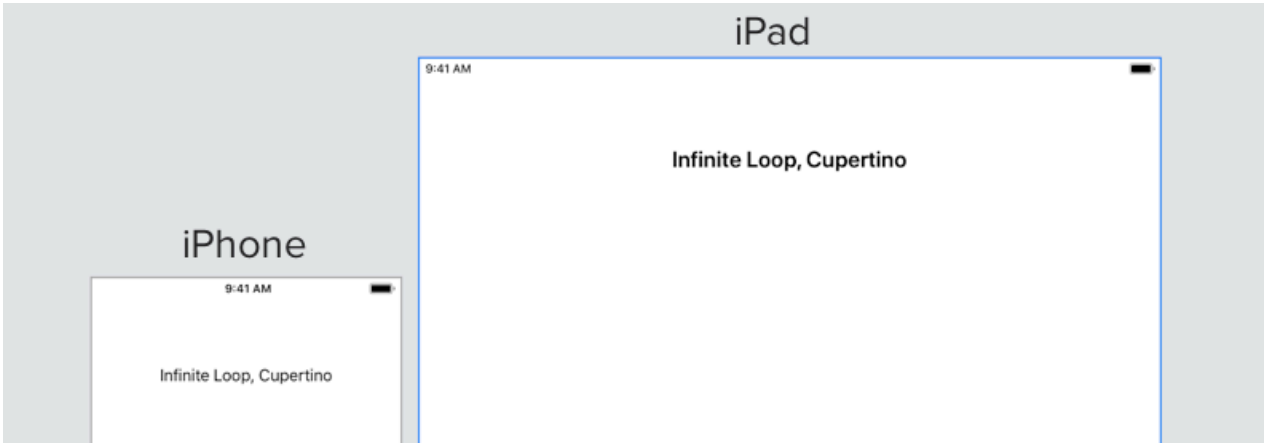
Sometimes you might want to cater to only one trait such as width, ignoring the height (regardless it is compact or regular), we can choose the height to be 'Any' in the variation :



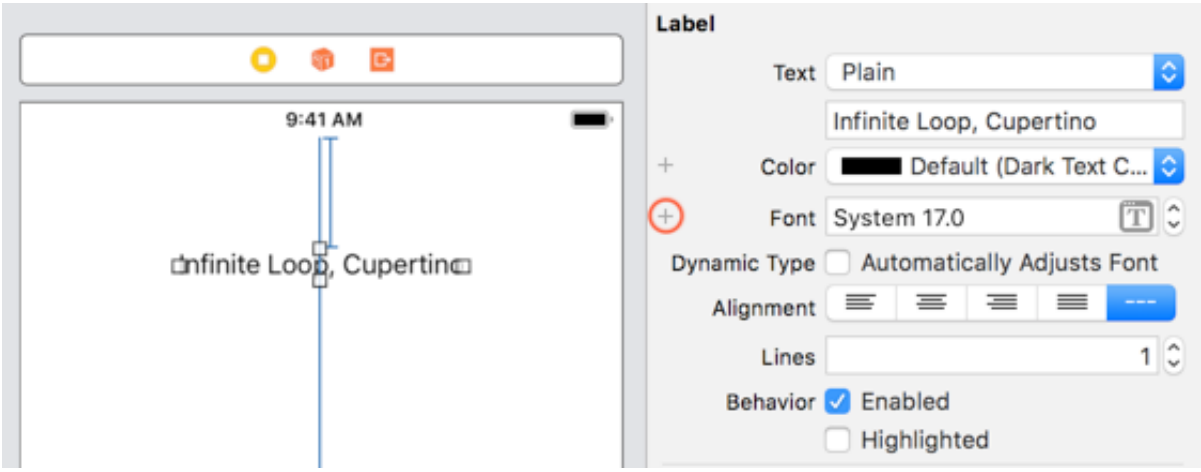
As long as the width of the app screen size belongs to compact, this variation will be used.

Different font size for different size classes

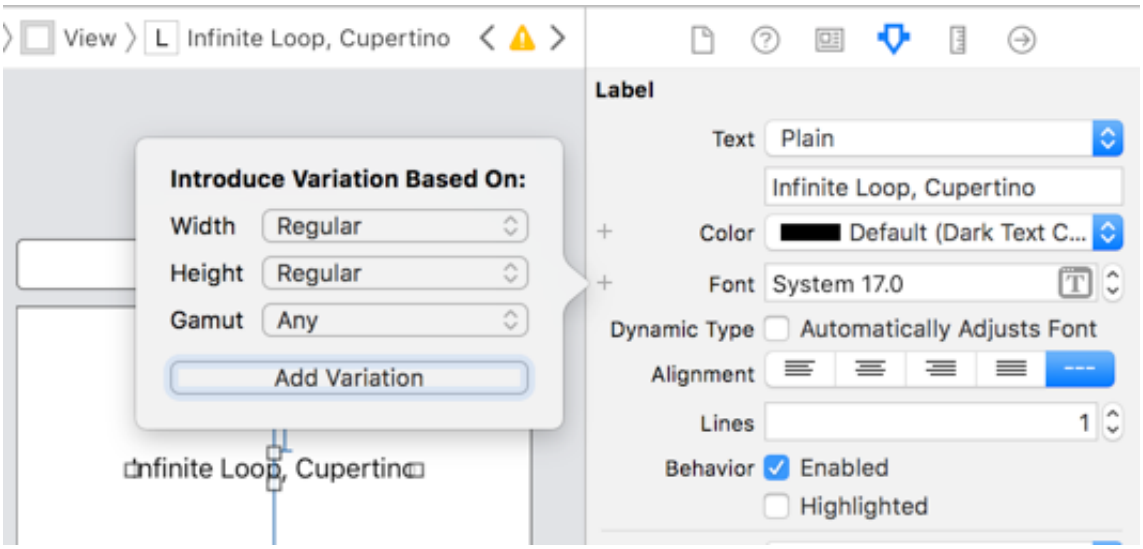
Using the '+' button, we can configure different font for different size classes. Say we want to make a label font bigger and bold when viewed on iPad (Width Regular, Height Regular).



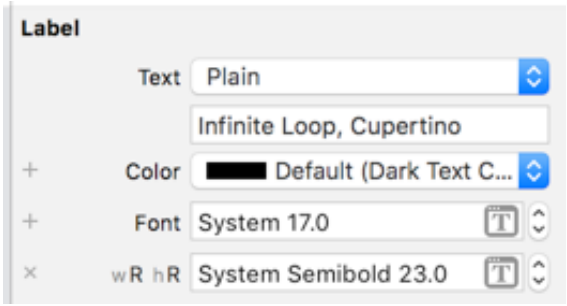
Select the label, click the '+' beside the 'Font' attribute.



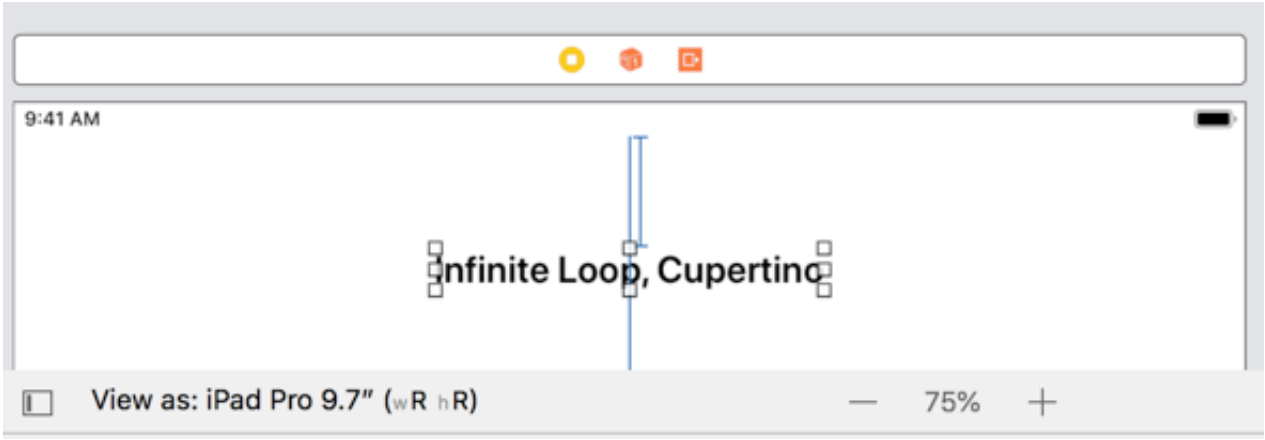
Then select Width Regular, Height Regular (iPad portrait mode is wR hR), and click 'Add Variation'.



Then on the wR hR Font variation, make it bigger and bolder (I have selected System 23.0 and Semibold for this, you can use any font you like).



Now switch the device to an iPad, and you will see a larger font displayed!

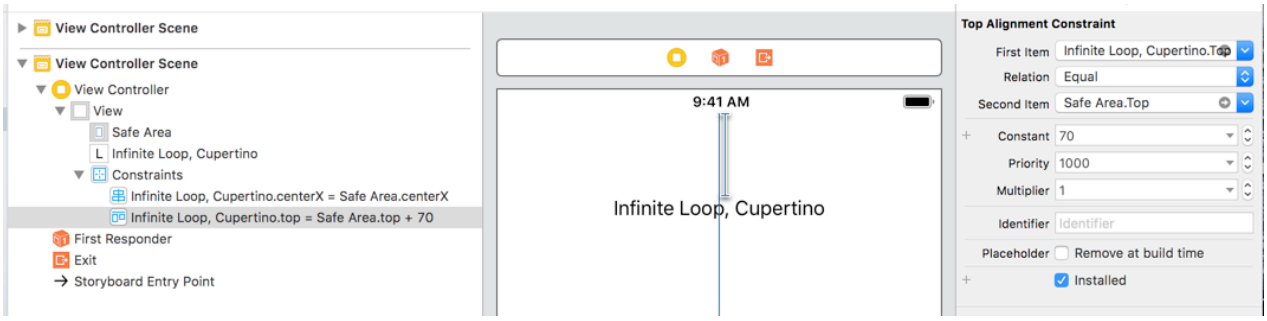


With a larger font on a larger device, user won't feel there's too much space lingering around and this make it easier for user to read the text from a further distance.

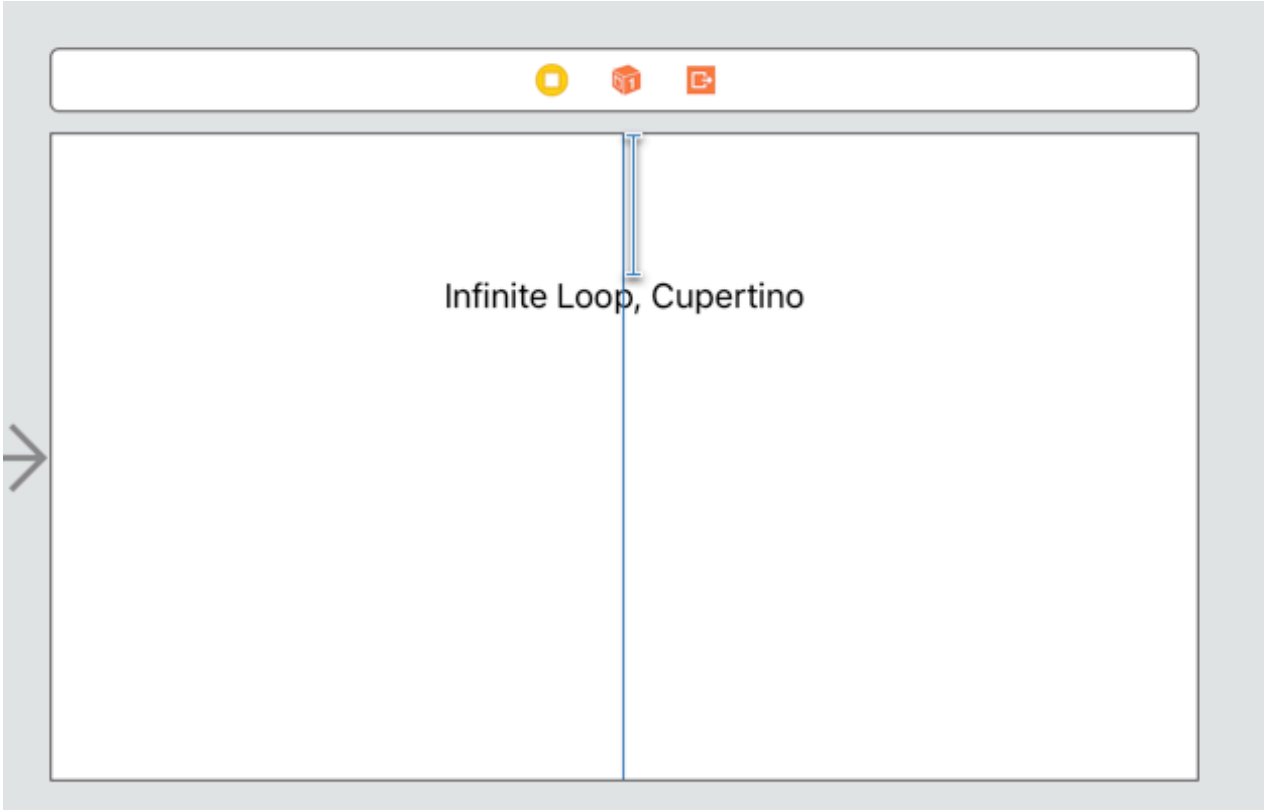
In the next section, we will discuss on how to use different constraint values for different size classes.

Different constraint values for different size classes

Using the previous font example, let's say we have a top constraint for the label, which have constant 70 from the top of the label to the top safe area.

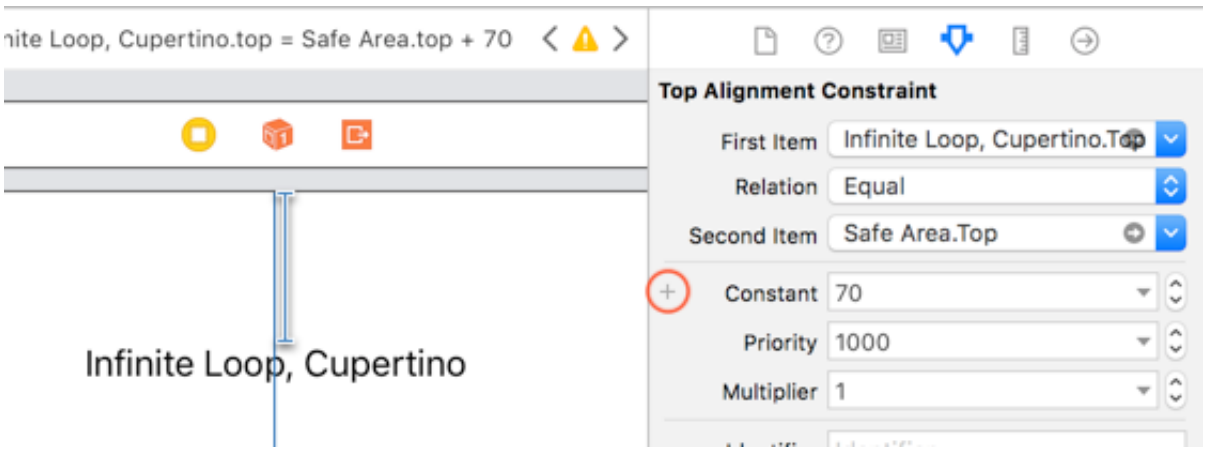


If we view it on a landscape iPhone, the top spacing seems a bit too much as the height of a landscape iPhone is a lot less than in portrait.

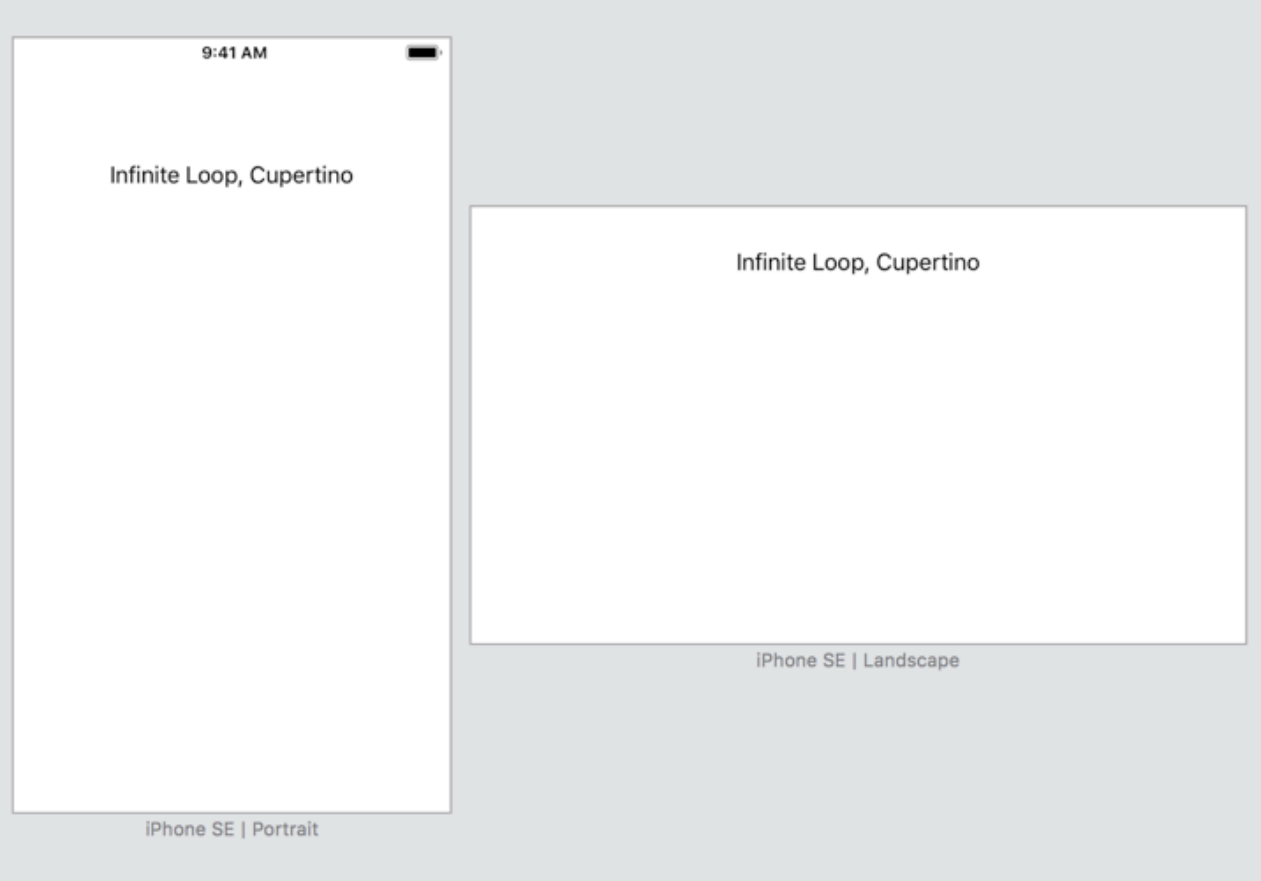


Let's reduce the top spacing of the label when viewed in iPhone portrait mode (w C h C).

Select the top constraint (either from the Document outline on the left or from the size inspector), then select the attribute inspector tab, and click on the '+' beside the constant attribute.



Select Width Compact, Height Compact and click 'Add Variation'. Then on the wC hC variation, change the constant to a smaller value (eg: 30). Now when we view it on a landscape iPhone , the top space is smaller :



Applying size classes programmatically

Apple makes it really easy to apply size classes variation using the Interface Builder / Storyboard. But if you want to do it programmatically, they also provide a property and a view controller method for detecting size classes update which you can override.

There's a **traitCollection** property for view controller which we can use it to check the current size classes of the screen. The **traitCollection** is an UITraitCollection object, which contains properties like**.horizontalSizeClass** and **.verticalSizeClass** . We can use these properties to determine current size classes of the app. The .horizontalSizeClass is the width size class whereas .verticalSizeClass is the height size class.

An example usage :

```
class YourViewController: UIViewController {

  // ....
  if(self.traitCollection.horizontalSizeClass == .compact &&
     self.traitCollection.verticalSizeClass == .regular) {
     // w C h R
     // hide the red view when in w C h R size class (iPhone portrait)
     self.redView.isHidden = true
  }
  // ....
}
```

When the size class of a device changes, either by rotation or opening split view in iPad, two lifecycle methods will be called. **willTransition(to newCollection: , with coordinator:)** and **traitCollectionDidChange(_ previousTraitCollection:)**.

willTransition(to newCollection: , with coordinator:) method will be called just before the app adjust to the new size class. Using the coordinator animate closure, we can put in custom animation / code which will be executed alongside with the system rotation / split view animation (when you rotate the phone, the system will have a default rotation animation on element that have auto layout constraints set), and also code that will be executed after the system animation is completed.

```
class YourViewController: UIViewController {
    override func willTransition(to newCollection: UITraitCollection, with
coordinator: UIViewControllerTransitionCoordinator) {
        super.willTransition(to: newCollection, with: coordinator)

        coordinator.animate(alongsideTransition: { _ in
            // temporarily grow the stack view size by 1.1x alongside with
system rotation animation
            self.stackView.transform = CGAffineTransform(scaleX: 1.1, y: 1.1)
        }, completion: { _ in
            // revert back the stack view size to its original size after the
system animation is complete
            self.stackView.transform = CGAffineTransform.identity
        })
    }
}
```

After the app has adjusted to the new size class (animation done), **traitCollectionDidChange(_ previousTraitCollection:)** will be called. The **traitCollectionDidChange** method will be called whenever the size classes of the screen changes and also on first load (when the app just finished launched).

```
class YourViewController: UIViewController {

  /* this will be called when the screen first load and also
     after the size classes of the screen changes (rotation, split view)
     */
    override func traitCollectionDidChange(_ previousTraitCollection:
UITraitCollection?) {
        super.traitCollectionDidChange(previousTraitCollection)

        print("trait collection did change")
    }
}
```

To replicate the earlier example of changing the isHidden property of Red View inside Stack View, we can do it programmatically like this:

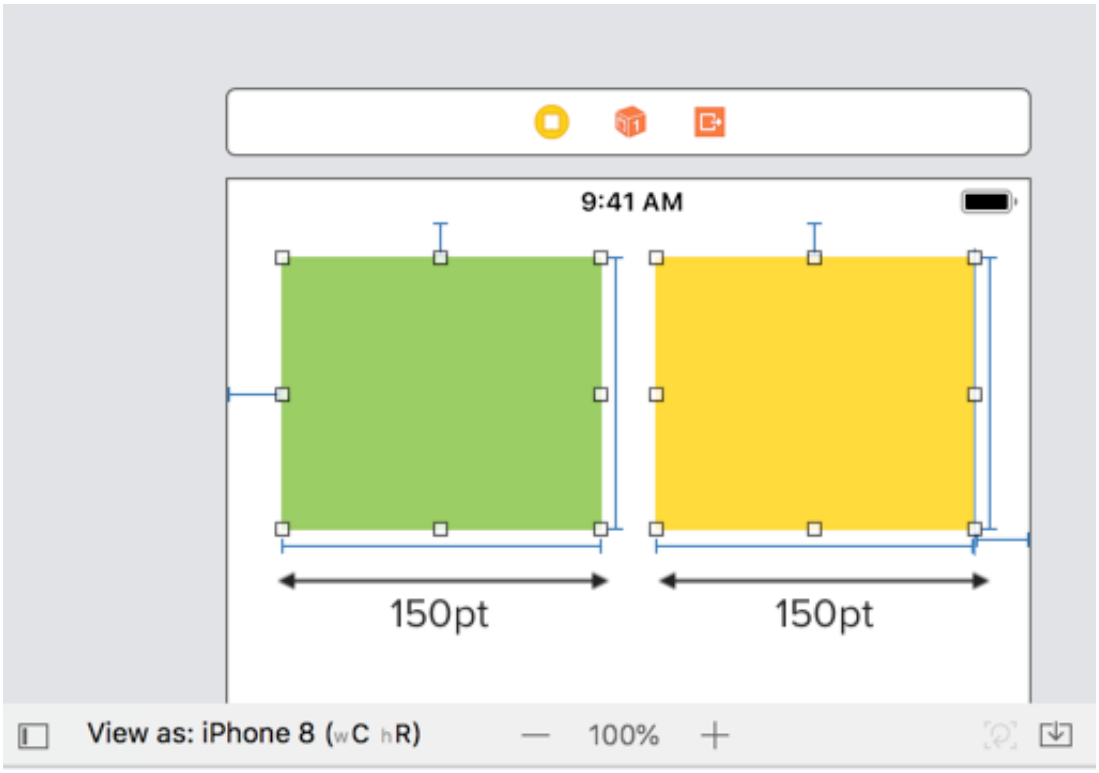
```
override func willTransition(to newCollection: UITraitCollection, with
coordinator: UIViewControllerTransitionCoordinator) {
    super.willTransition(to: newCollection, with: coordinator)

    coordinator.animate(alongsideTransition: { _ in
        // h C w R, hide the red view in the stack view
        if(newCollection.horizontalSizeClass == .compact &&
newCollection.verticalSizeClass == .regular) {
            self.redView.isHidden = true
        } else {
            self.redView.isHidden = false
        }
    }, completion: { _ in
       // ...
    })
}
```

In this chapter, we have applied size classes on hidden attribute, font and constraint's constant. There's other attribute you can try out as well like background color etc, as long as there's a "+" icon beside the attribute, you can add size classes variation to it on the interface builder! If the attribute you want to variate doesn't have a "+" icon in interface builder, you would have to do it manually in code.

15 - Thinking in proportion

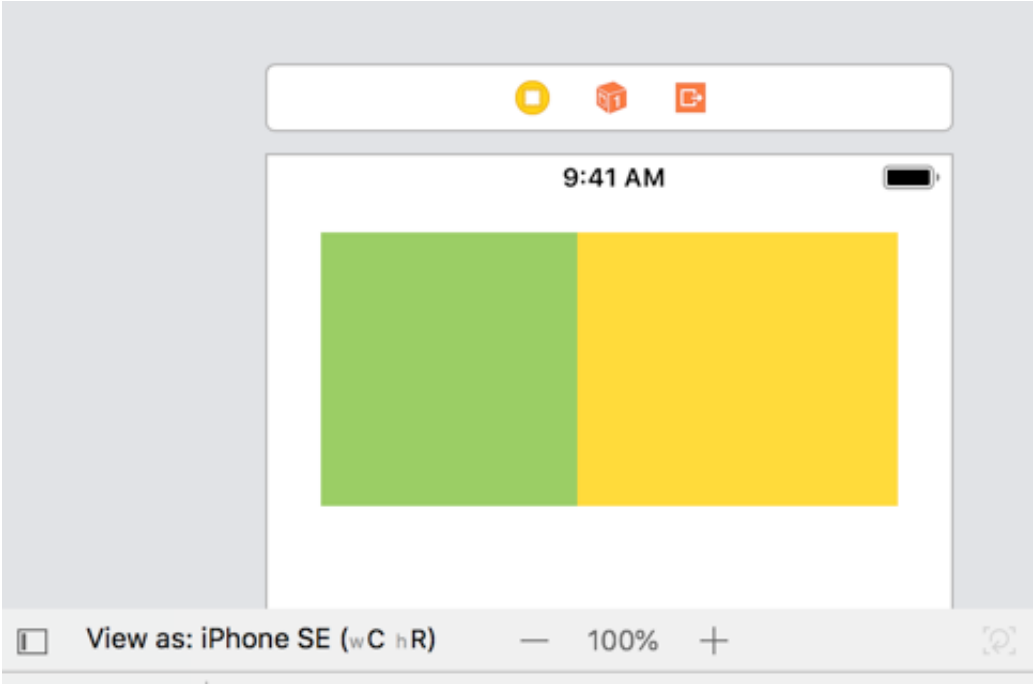
This chapter will focus on the thought process on designing responsive layout that looks good on multiple devices.

Usually when we start out with Auto Layout, we tend to set constraint with fixed value like this :

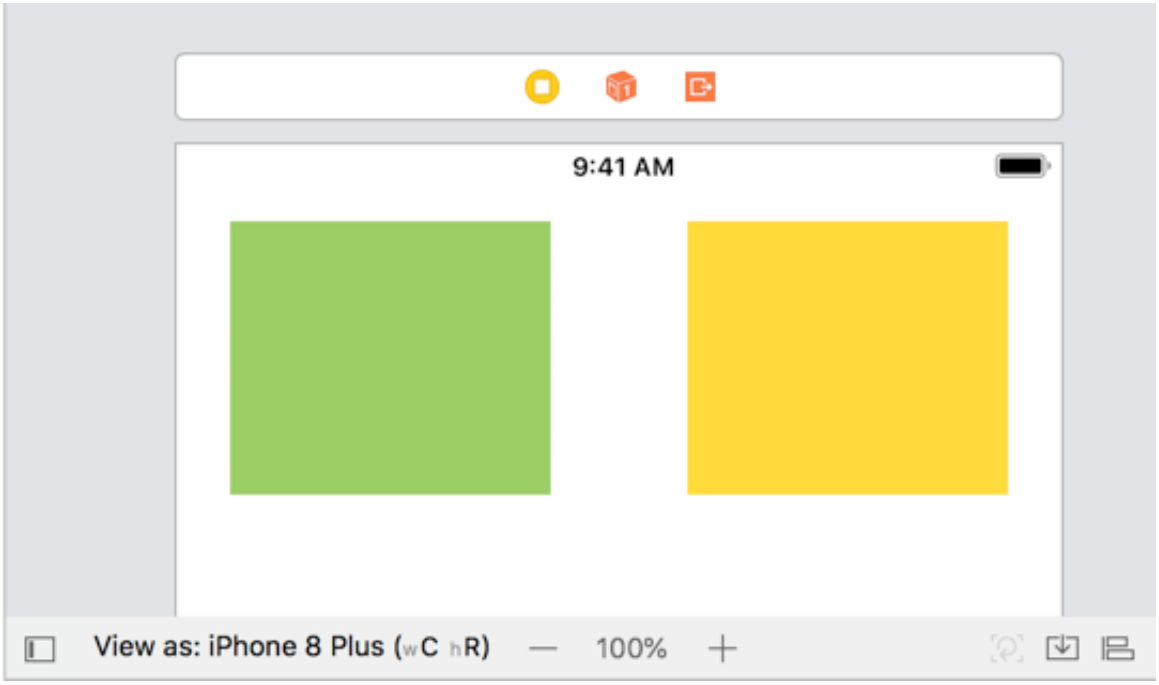


Here we have set a fixed width constraint of value 150pt for both of the green and yellow view. The green view has a leading constraint and the yellow view has a trailing constraint. This looks good on iPhone 8 as the width of iPhone 8 is 375pt.

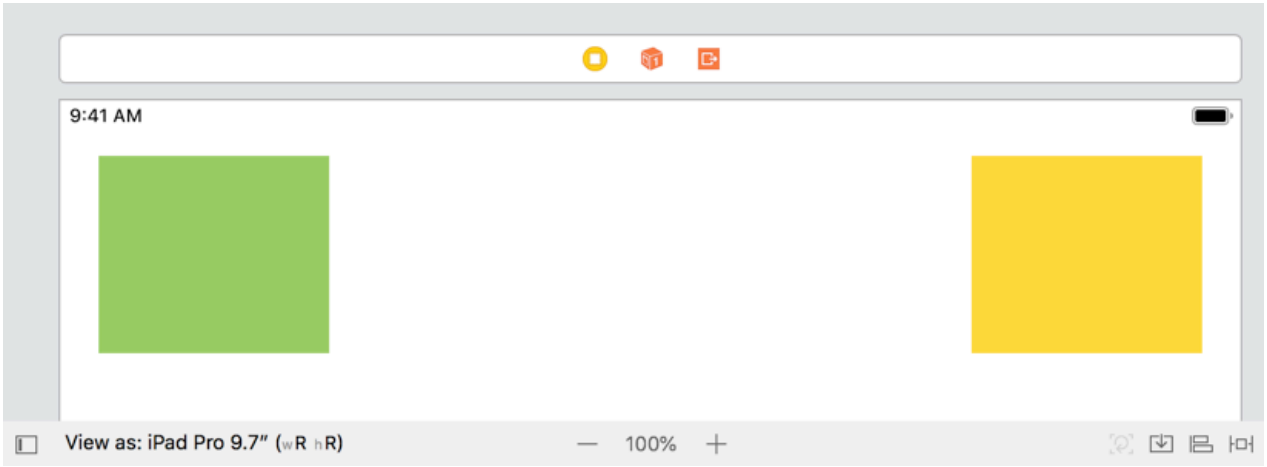
But when we view it in a smaller device like iPhone SE, the views overlap due to the smaller screen width!



Or when we view it in a larger device like iPhone 8 Plus, the gap between the two views become too big :



Heck, when you view it in iPad, the gap between the two views is enormous :

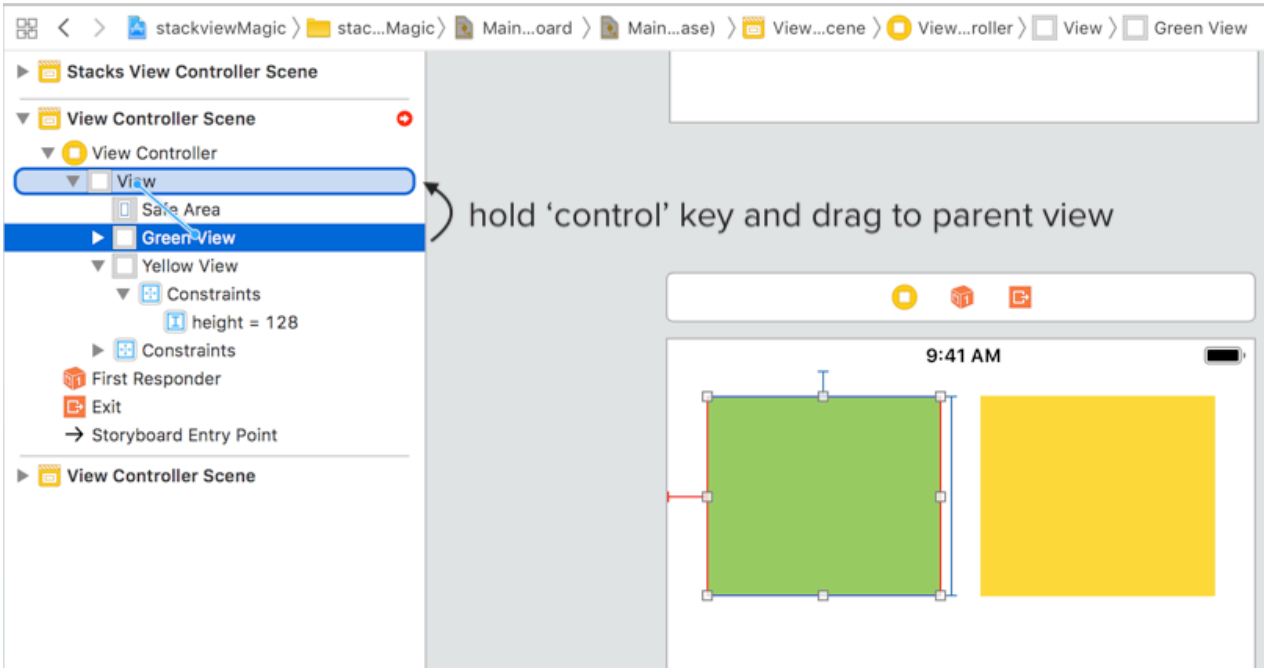


You might think of using size classes to put a different constant on the width constraint for iPad, but what if the user is using a larger iPad Pro (10.5 or 12") ? The same problem might still persist.

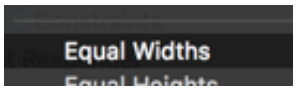
One of the key on designing responsive layout that looks great on all devices is to **design in ratio**. Using a fixed width usually will result in the layout looks good only on one screen size, which is the one you design in. 😅

For the yellow and green views above, our intent is to make both of them occupy around half of the screen width. To achieve the this, let's remove the fixed width constraint on both the views.

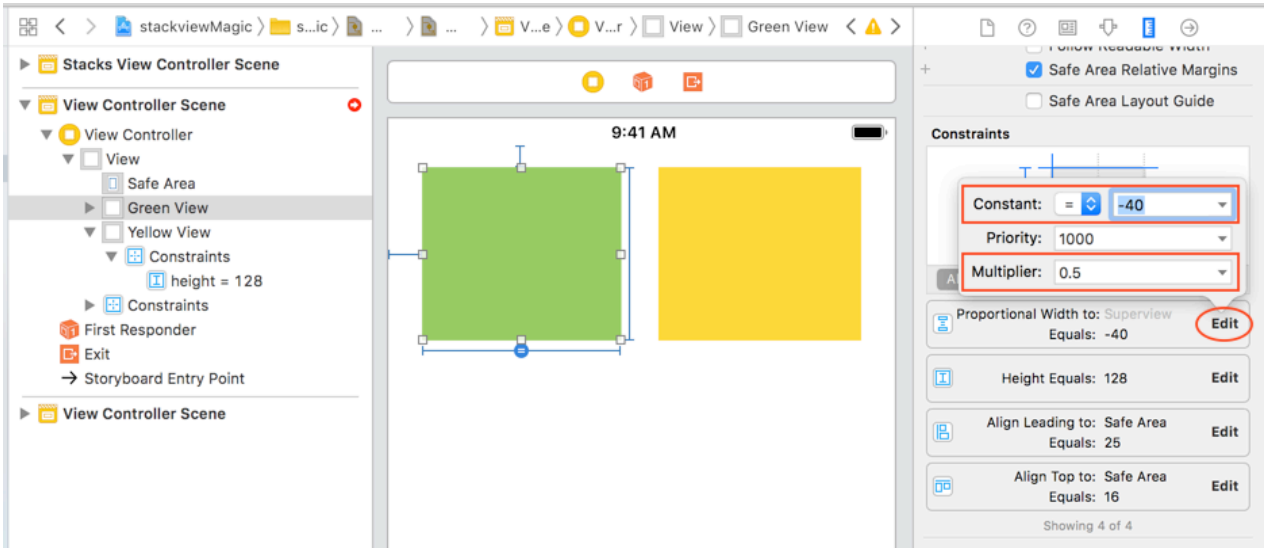
Next, create an equal width constraint from the green view to its super view (control + drag):



Select 'Equal Width'



After creating the constraint, head over to the size inspector of the green view, and edit the '**Equal Width to Superview**' constraint we have created just now.

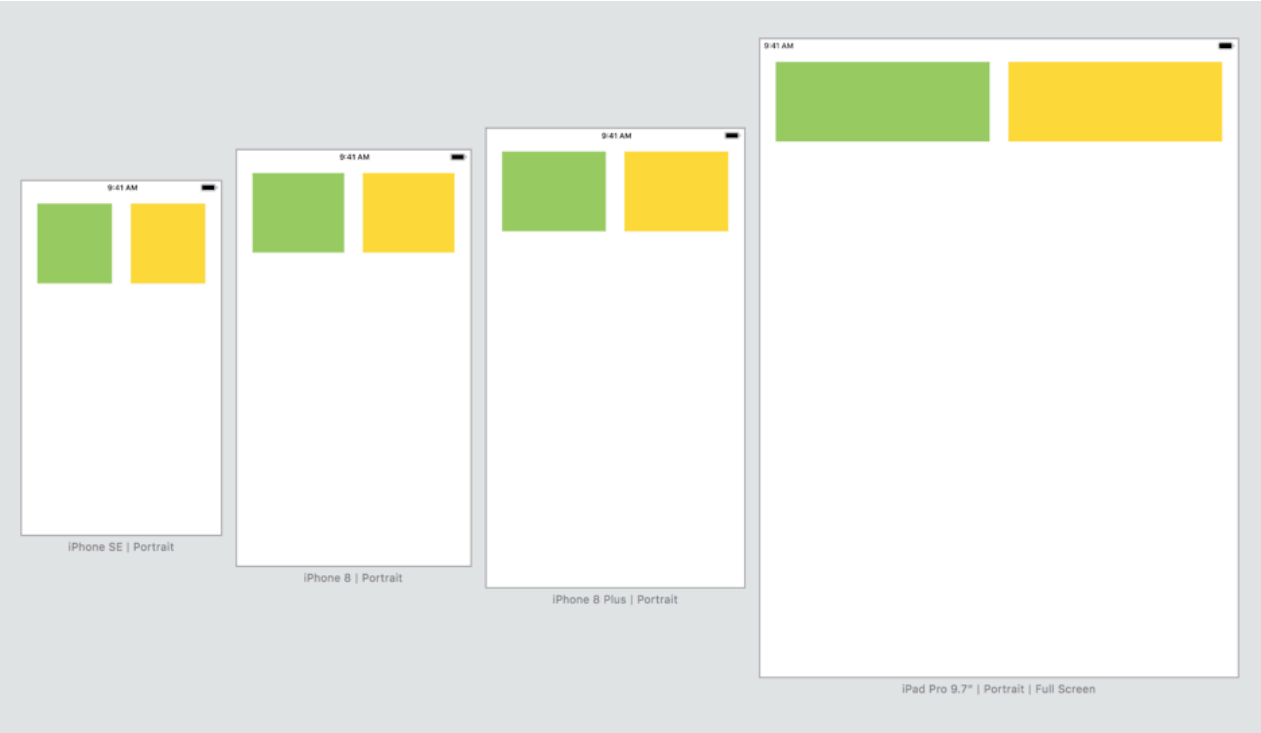


Change the multiplier to 0.5, and set the constant to -40 . This means that the green view width will be 0.5x of its super view (the view controller's root view) minus 40pt.

Green view width = 0.5x super view's width - 40 pt

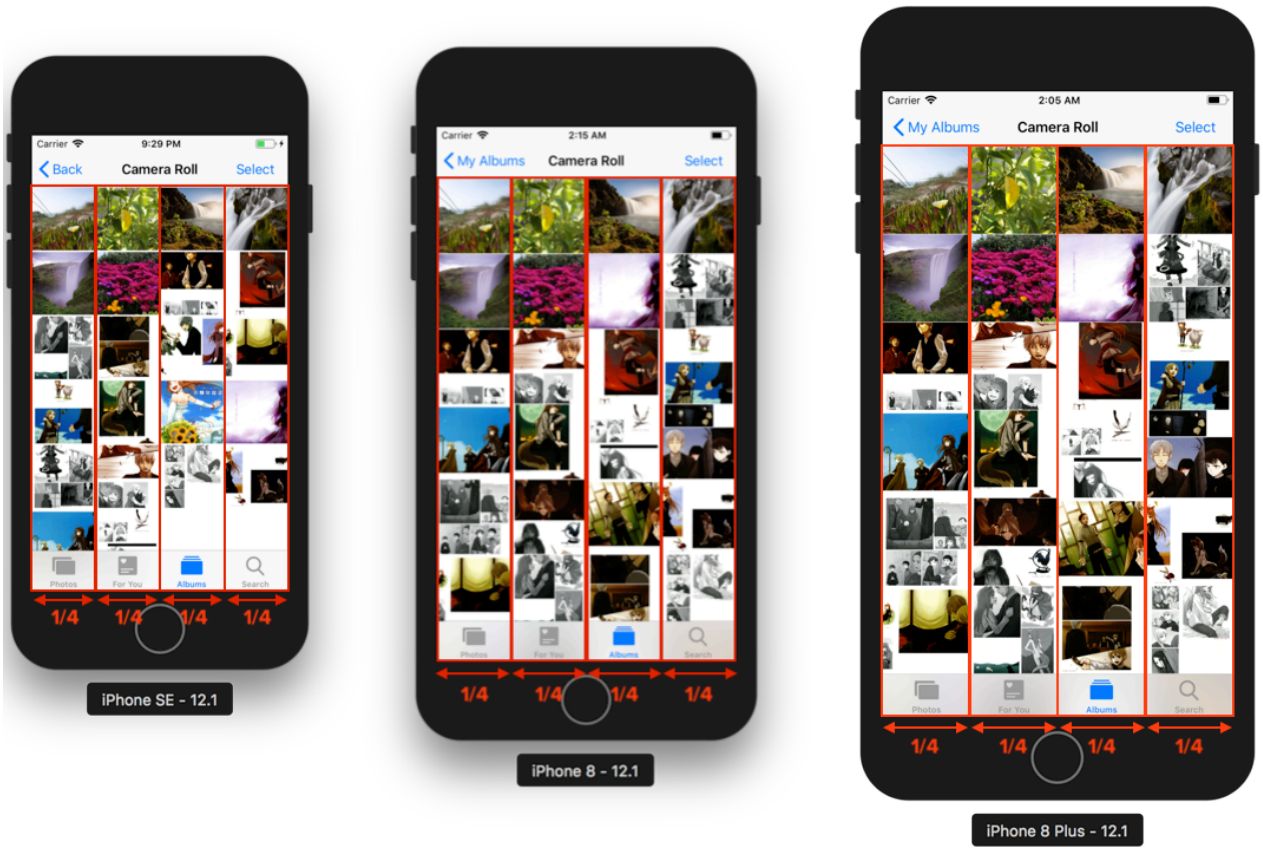
After changing the multiplier value, you will see that the constraint name has changed from 'Equal Width to Superview' to '**Proportional Width to Superview**'.

Do the same steps on the yellow view, and we will have both of them roughly equal to half the width of the screen. Let's view it again on different devices :



It now looks good from the smallest iPhone to iPad! 🙌 Using ratio / proportional constraint allows the view to be scaled up / down dynamically based on the screen width / height.

One of the example of proportional layout we can look at is the **iOS Photos app**, Photos app divided its photos layout into 4 equal width column (each photo take up around 25% of the screen width) . Each photo thumbnail occupies 1/4 of the screen width and its height is equal to its own width (ie. a square).



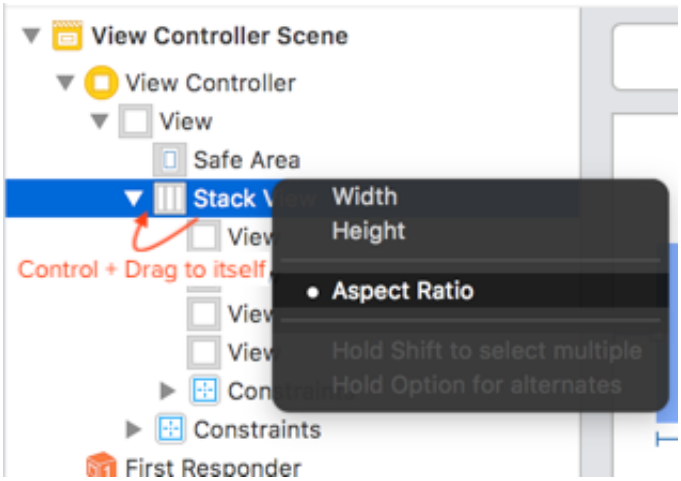
Although the devices dimension varies, the layout still looks good because the width is proportional to the screen width.

Using Stack view to simplify proportional layout

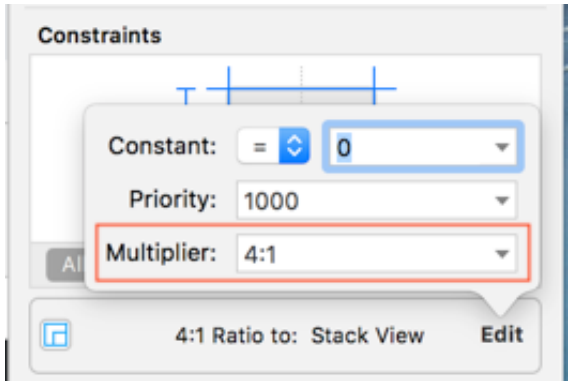
It can be time consuming to set proportional width constraint for each of the UI elements, to simplify this process, we can use a stack view with Equal distribution (ie. Distribution = Fill Equally) and insert the element into it.

In the example below, we will create a horizontal stack view that can hold 4 equal-width squares (UIView). As there will be 4 square inside the stack view, the width height ratio of the stack view will be 4:1 , we will create this ratio constraint on the stack view.

Select the stack view, and then control + drag to itself, and select '**Aspect Ratio**' to create a ratio constraint.

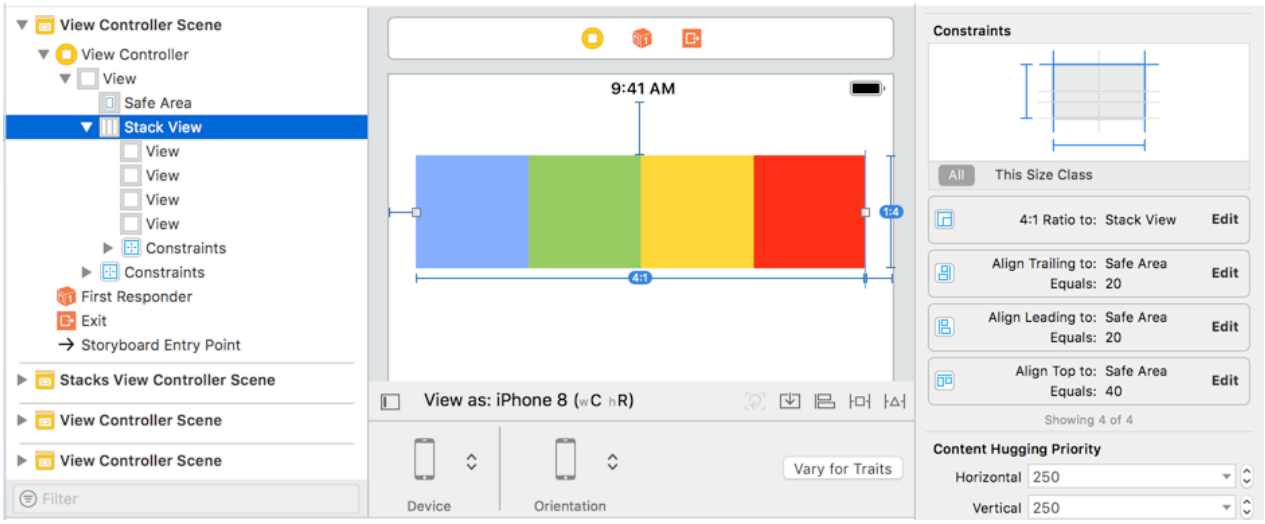


Then edit the aspect ratio constraint's multiplier to 4:1

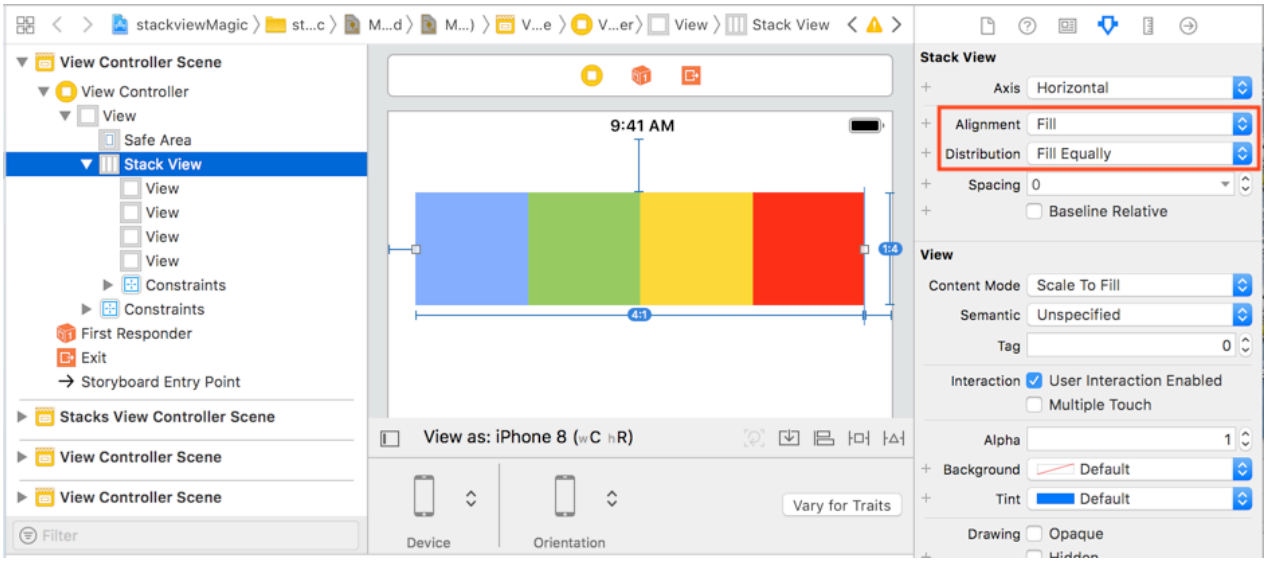


Remember that Auto Layout needs to know or be able to calculate the X position, Y position, width and height of the stack view. You'll need to add a top/bottom constraint (Y position), leading and trailing constraint (X position and also width), the height can be calculated from the width, which is 1/4th of the width.

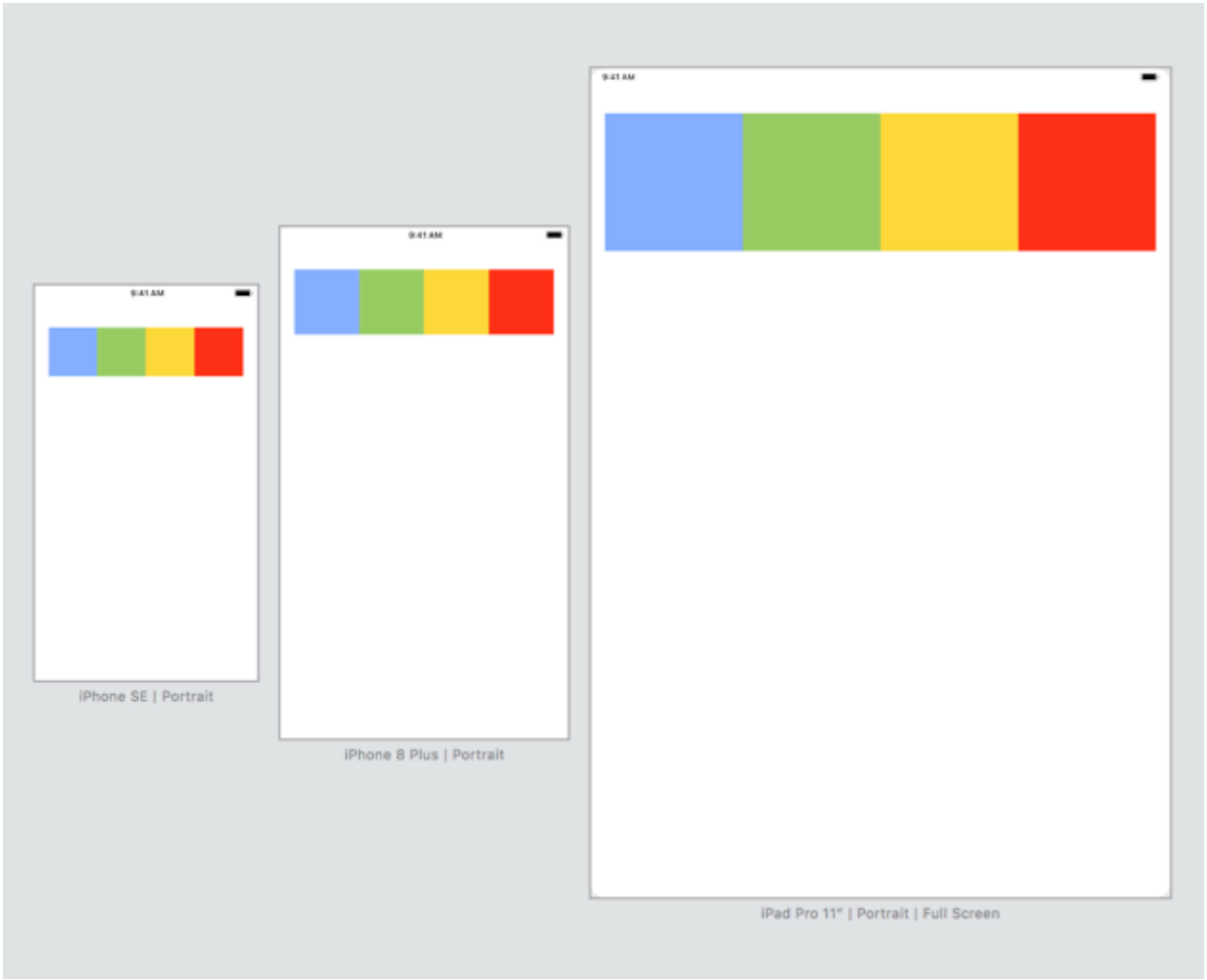
Here's the constraints I created for the stack view :



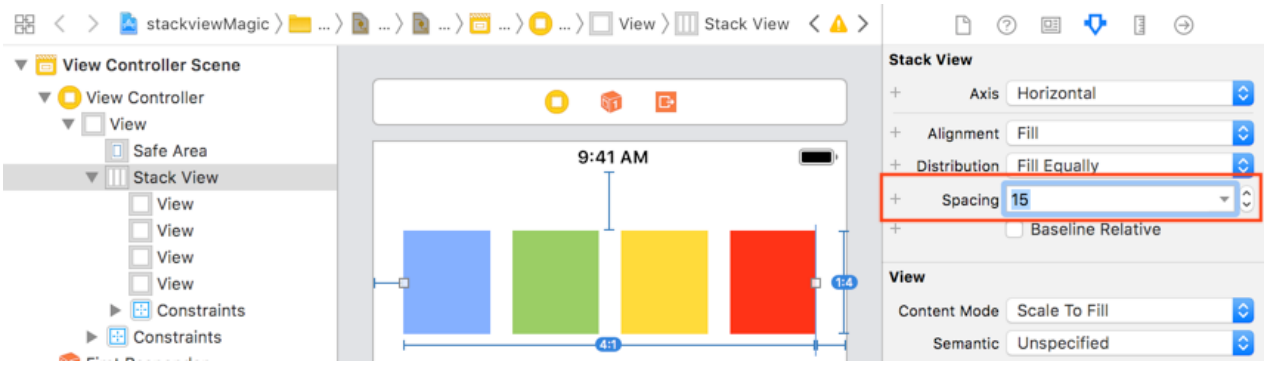
Remember to set the stack view to Fill Equally :



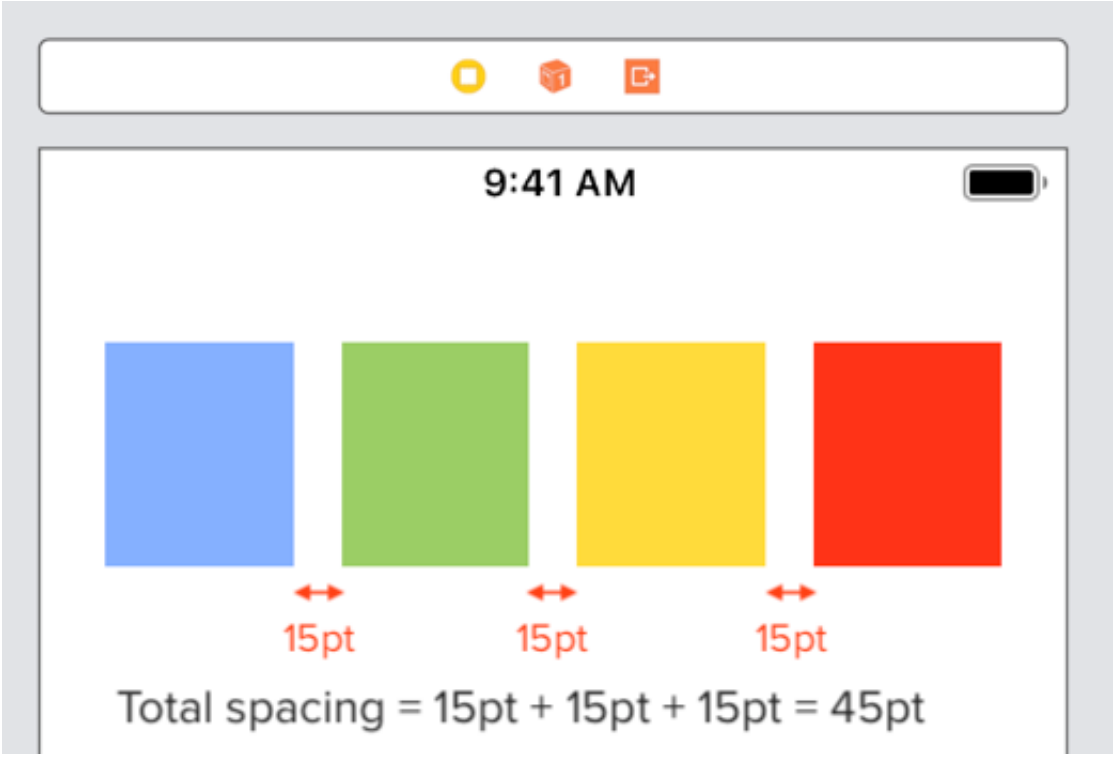
Then insert four views into it. Here's a preview of it in different devices (iPhone SE, iPhone 8 Plus and iPad Pro) :



The layout above looks good except that there's no space between each squares which make it feel kinda cramped. We can add some spacing between each of the squares by updating the 'Spacing' attribute of the stack view :

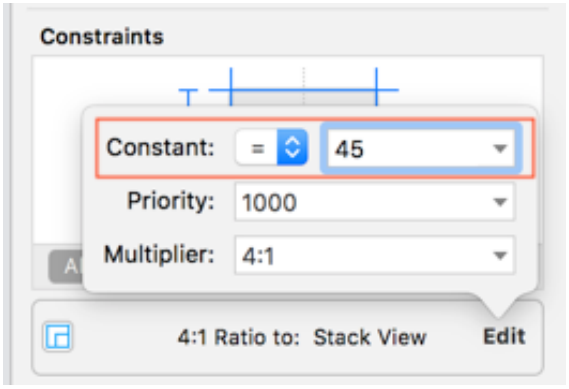


For this example, we added 15pt spacing between each elements. Notice that the view inside has become rectangular-ish (ie. width doesn't equal to height) after adding in the spacing. This is because the spacing has occupied around 45pt of width of the stack view :

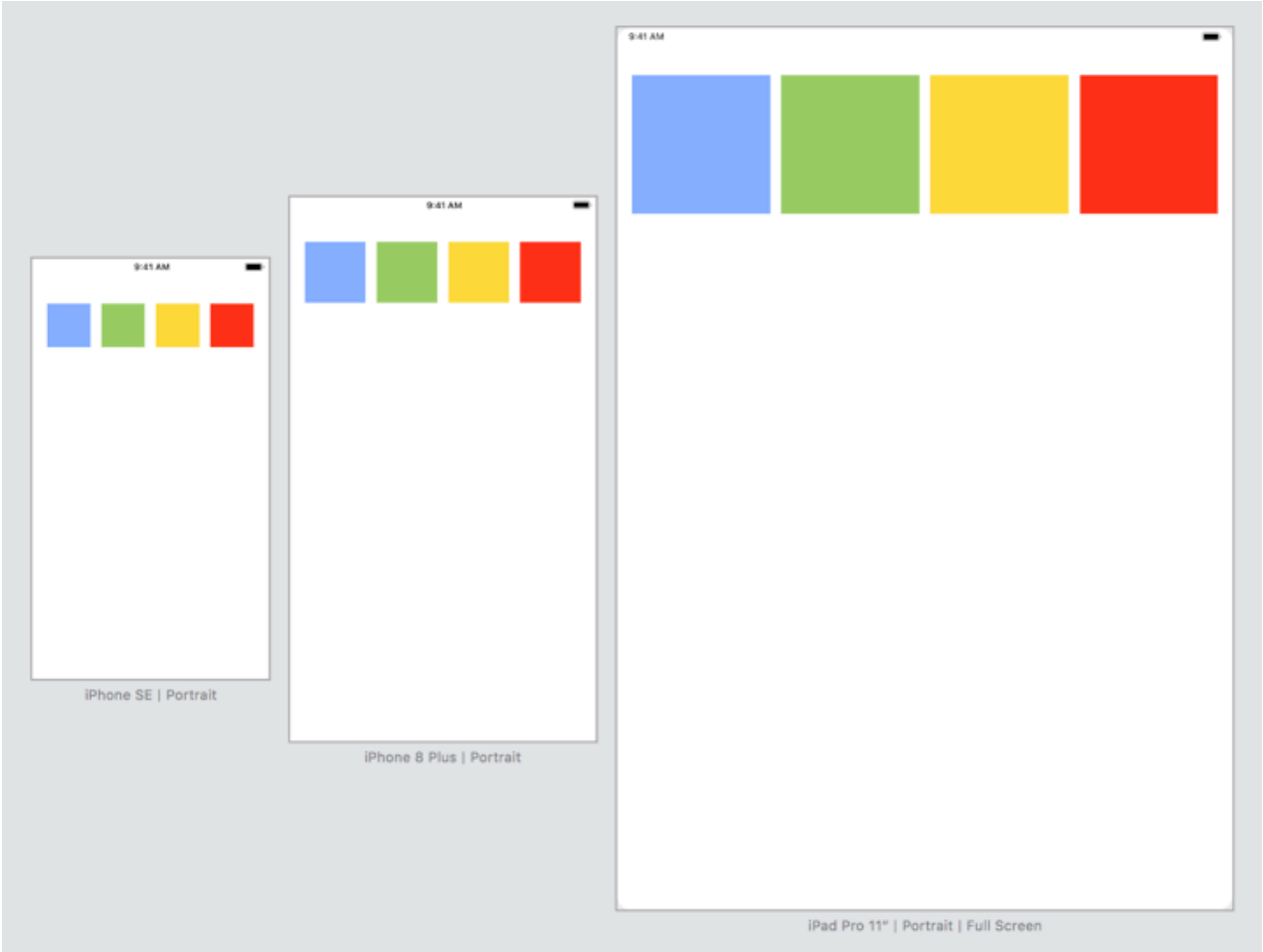


To make the child views become square, we will need to modify the Aspect Ratio constraint of the stack view. As the total spacing is 45pt, we will modify it such that the stack view width is equal to 4x height + 45pt.

Stack View Width - 45pt = 4 x Height



Now we have 4 squares with spacing that looks good across all devices :

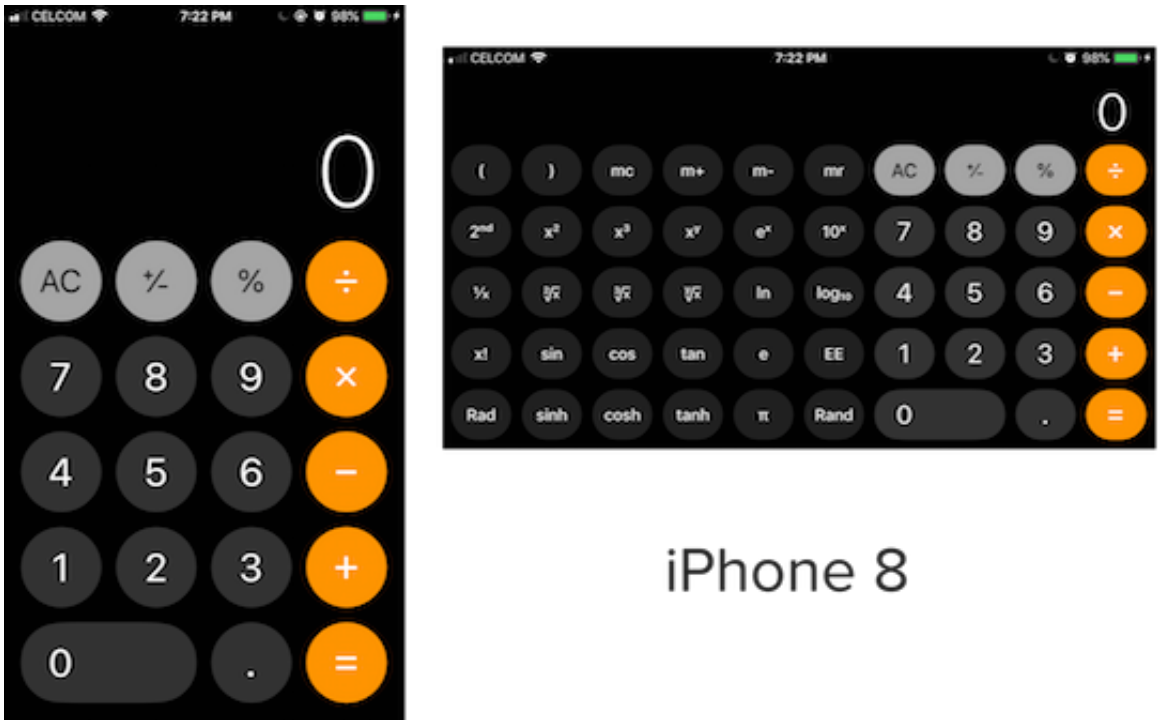


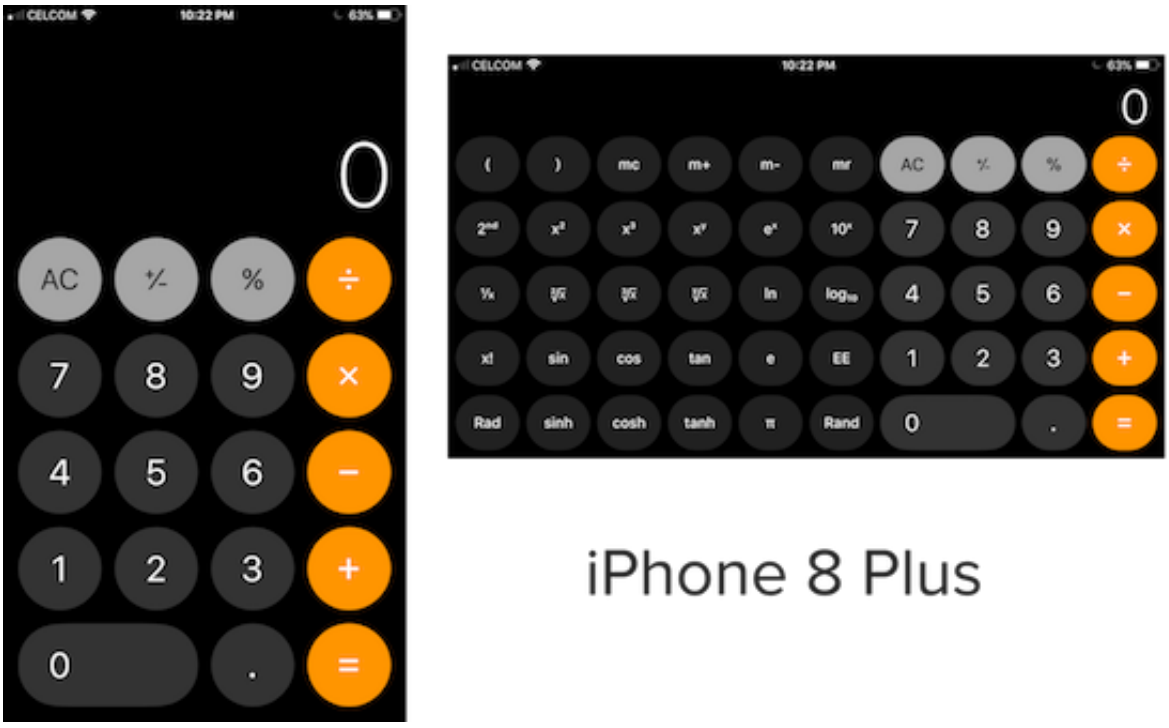
In the next chapter, we will study, decompose the iOS default calculator app into proportion / ratio and reconstruct it with Auto Layout.

16 - Calculator app design case study

In this chapter, we will breakdown the design of iOS stock calculator app, explain the building blocks of its layout and attempt to reconstruct it from scratch. Before continuing this chapter, I recommend you to read Chapter 13 - 15 (Stack View, Size classes and Thinking in proportion) if you haven't read them yet, as many techniques explained in those chapters will be used.

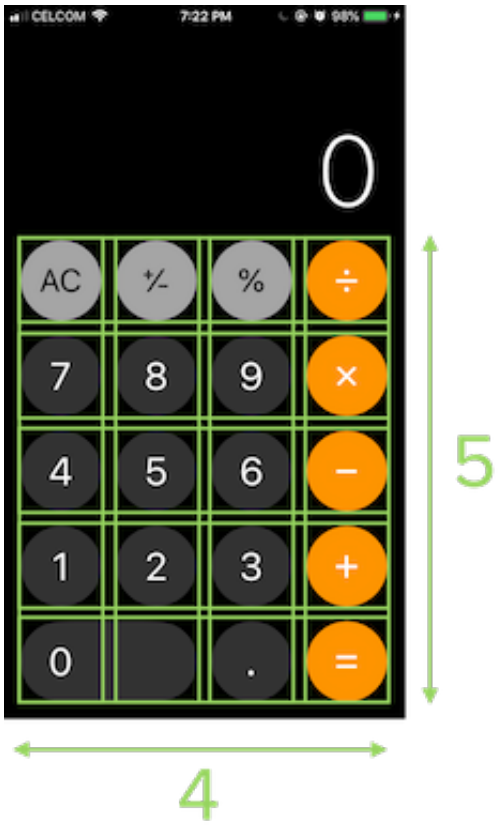
Here's how the iOS calculator app look in iPhone 8, iPhone 8 Plus and iPhone XR :



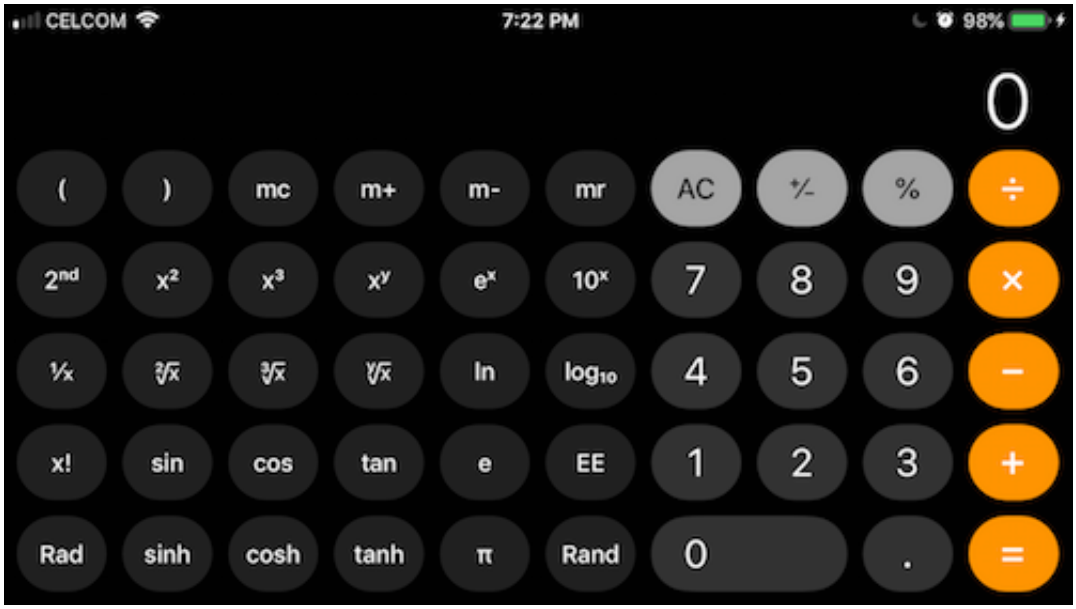


Notice that the buttons in the calculator app is proportional to the screen width / height.

In the portrait mode, the buttons are displayed in a 4:5 ratio (4x5 grid) with spacing between them, we can implement this using stack view with spacing, we also set a ratio constraint on the stack view. The buttons have width equal to their own height, making them a square.



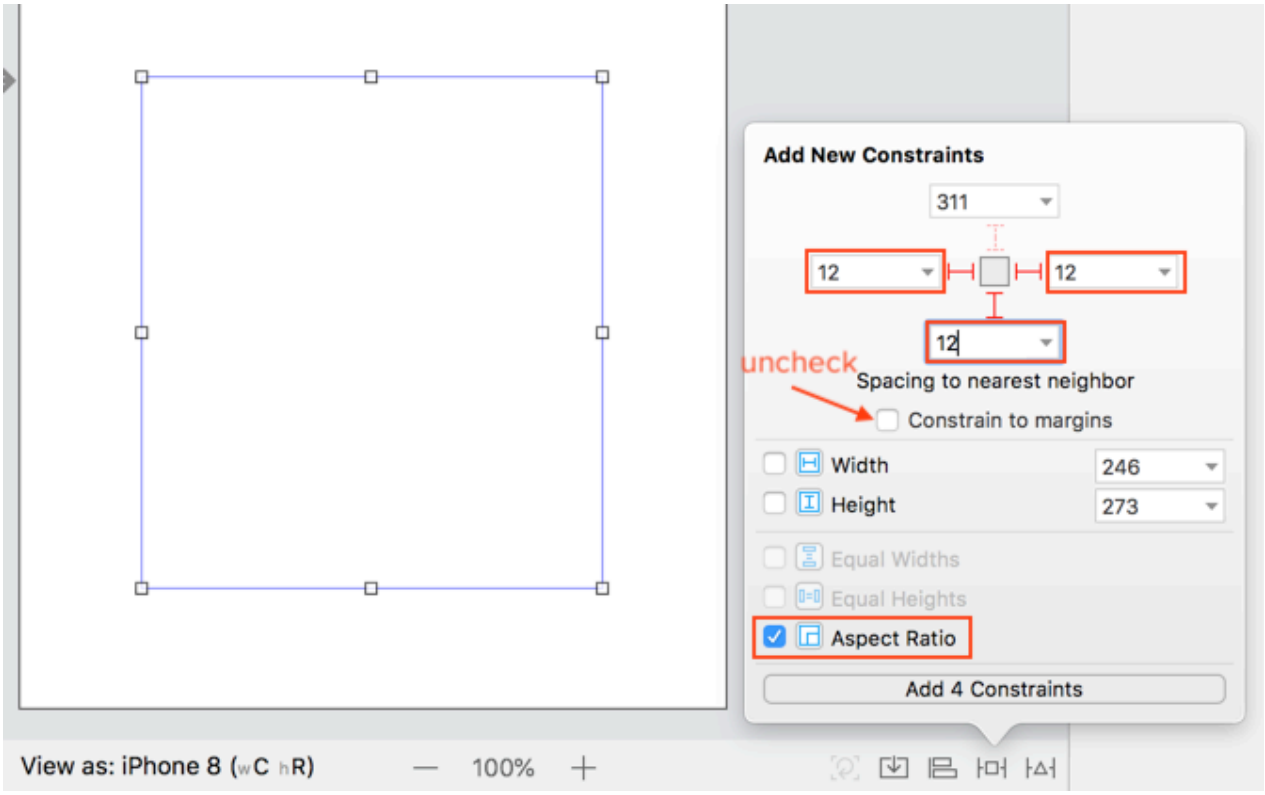
In the landscape mode, more buttons are displayed including advanced math functions like log, sin etc. The buttons in landscape are not square anymore (rectangular ish).



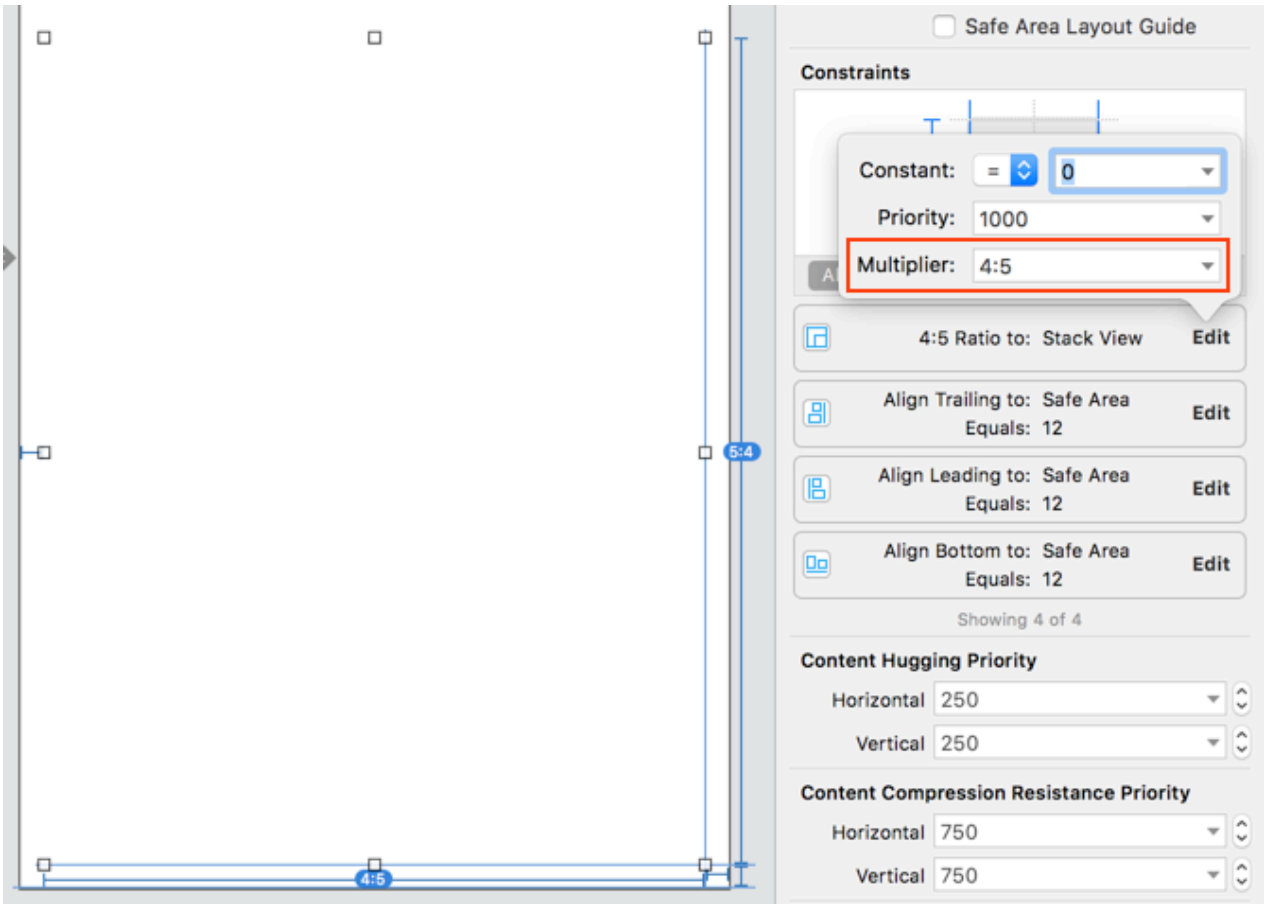
In the size classes chapter, we have talked about showing / hiding certain elements in stack view based on different size class (iPhone portrait is wC hR, iPhone landscape except 8 plus is wC hC). We will be hiding the extra advanced math function buttons on iPhone portrait mode (wC hR) by creating a variant for the '**hidden**' property for each of the buttons. We will demonstrate how to do this later on in the chapter.

As a start, create a new Xcode project, open the storyboard, drag a **vertical** stack view into the view controller and set constraints as following :

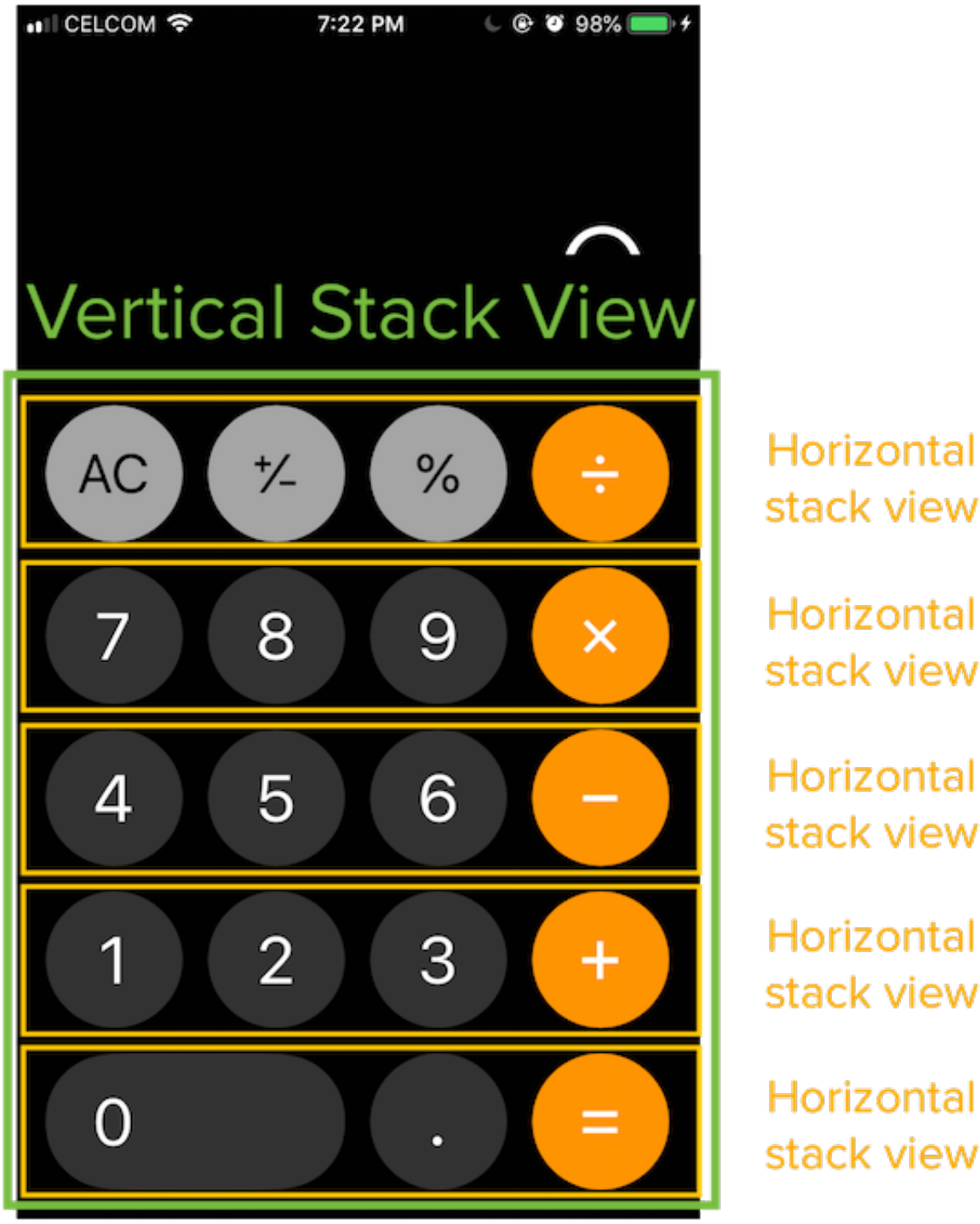
1. Leading constraint to Safe Area 12pt
2. Trailing constraint to Safe Area 12 pt
3. Bottom constraint to Safe Area 12 pt
4. Aspect Ratio constraint of 4:5 (4 width, 5 height)



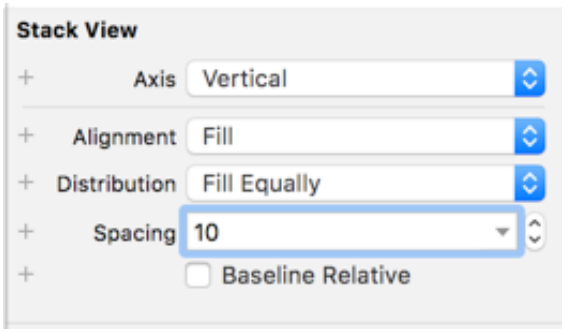
Edit the Aspect Ratio constraint to 4:5 (4 is width, 5 is height)



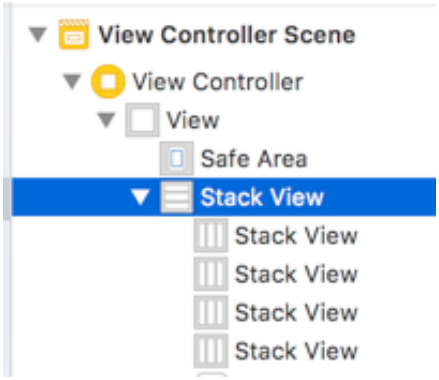
Now we have the main vertical stack view set, next we are going to insert a few horizontal stack views inside the vertical stack view.



Set the vertical stack view's **Distribution** property to **Fill Equally** so that each of the horizontal stack views will have equal height, and also set the spacing as 10 pt so that each horizontal stack views have a 10 pt distance between them.



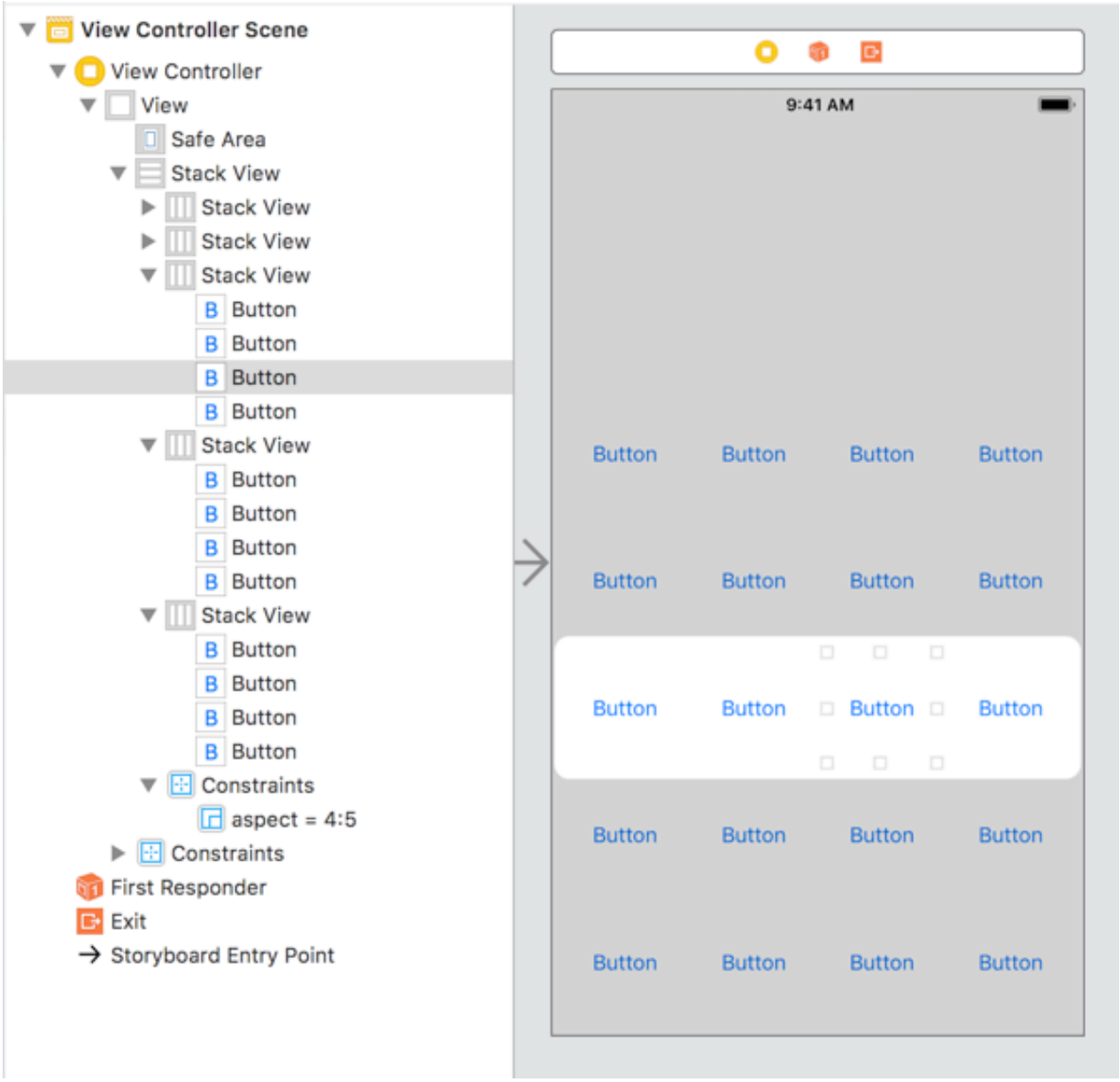
Next, drag **5** horizontal stack views into the vertical stack view.



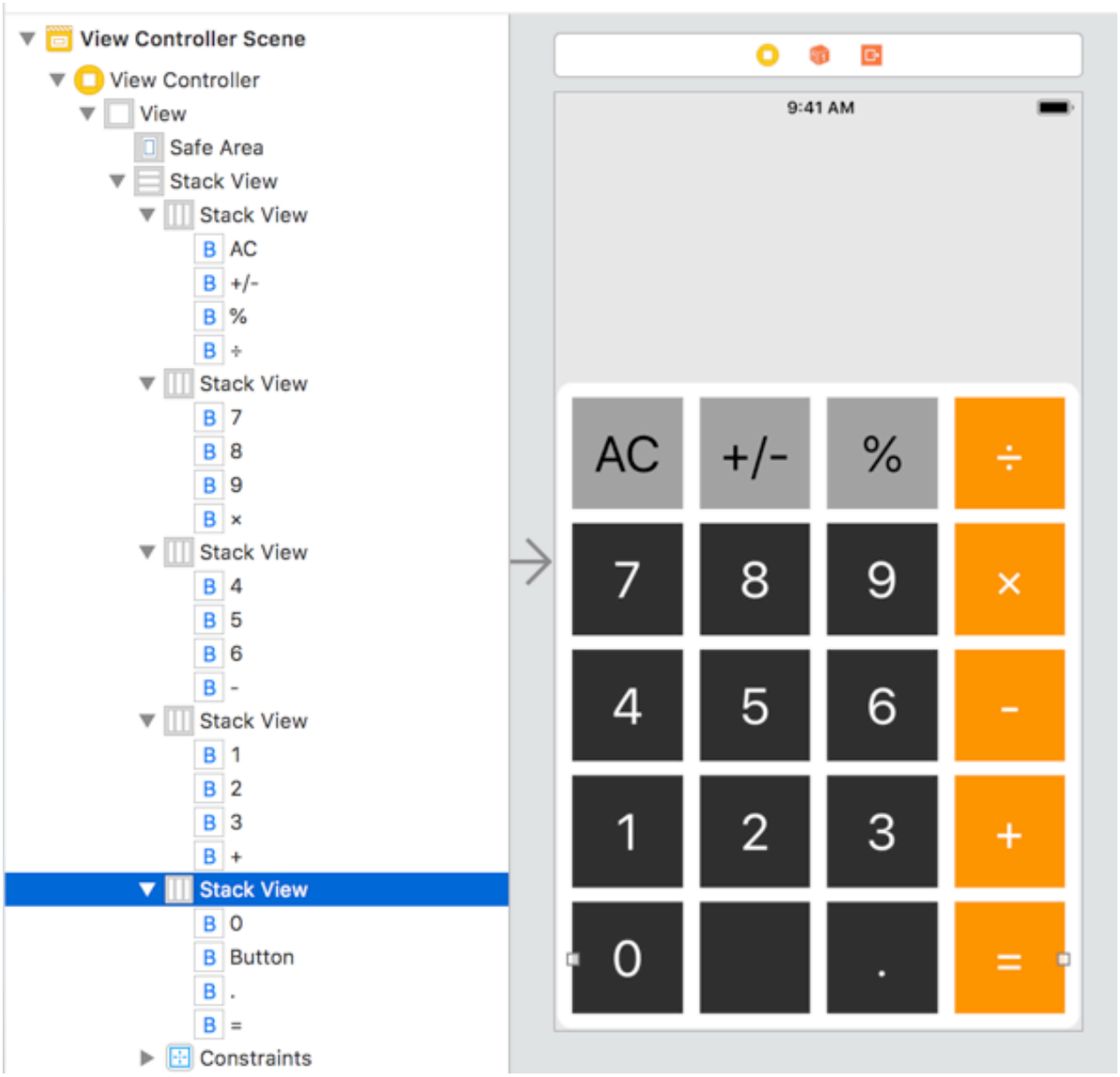
Set **Distribution** property to **Fill Equally** for all of those horizontal stack views as well. Set the spacing to 10 on these horizontal stack views to match the spacing of the vertical stack view, so that these buttons will look square.

Next, drag 4 buttons to each of the horizontal stack views, type the corresponding button text and style its background color as you like.

Here's how the layout look like (before styling) :



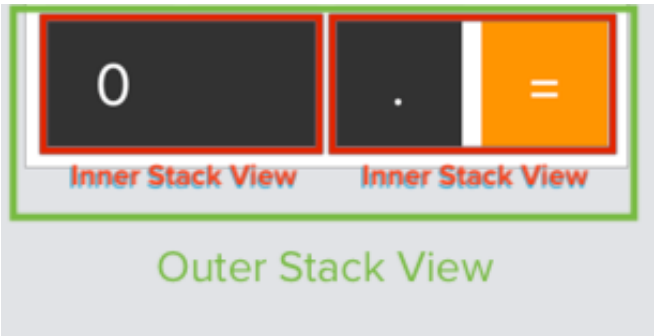
After styling the button text (I have set the font of button to **System Font with size 36.0**) and color, it looks like this :



Looking good! Except that the "0" button is not prolonged, the "0" button should be longer and taking half of the width of the bottom row, but now we have a gap and an empty button :

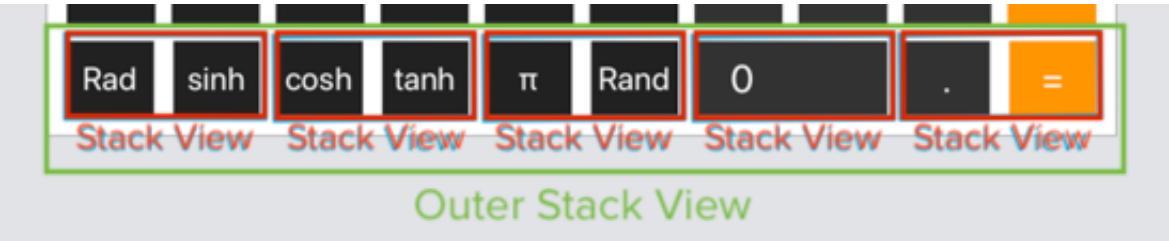
As the fill distribution to set to equal, we can't set a button to take 2 portion of the width, what should we do to elongate the 0 button?

One of the easiest way is to group two buttons into a bigger horizontal stack view, like this :



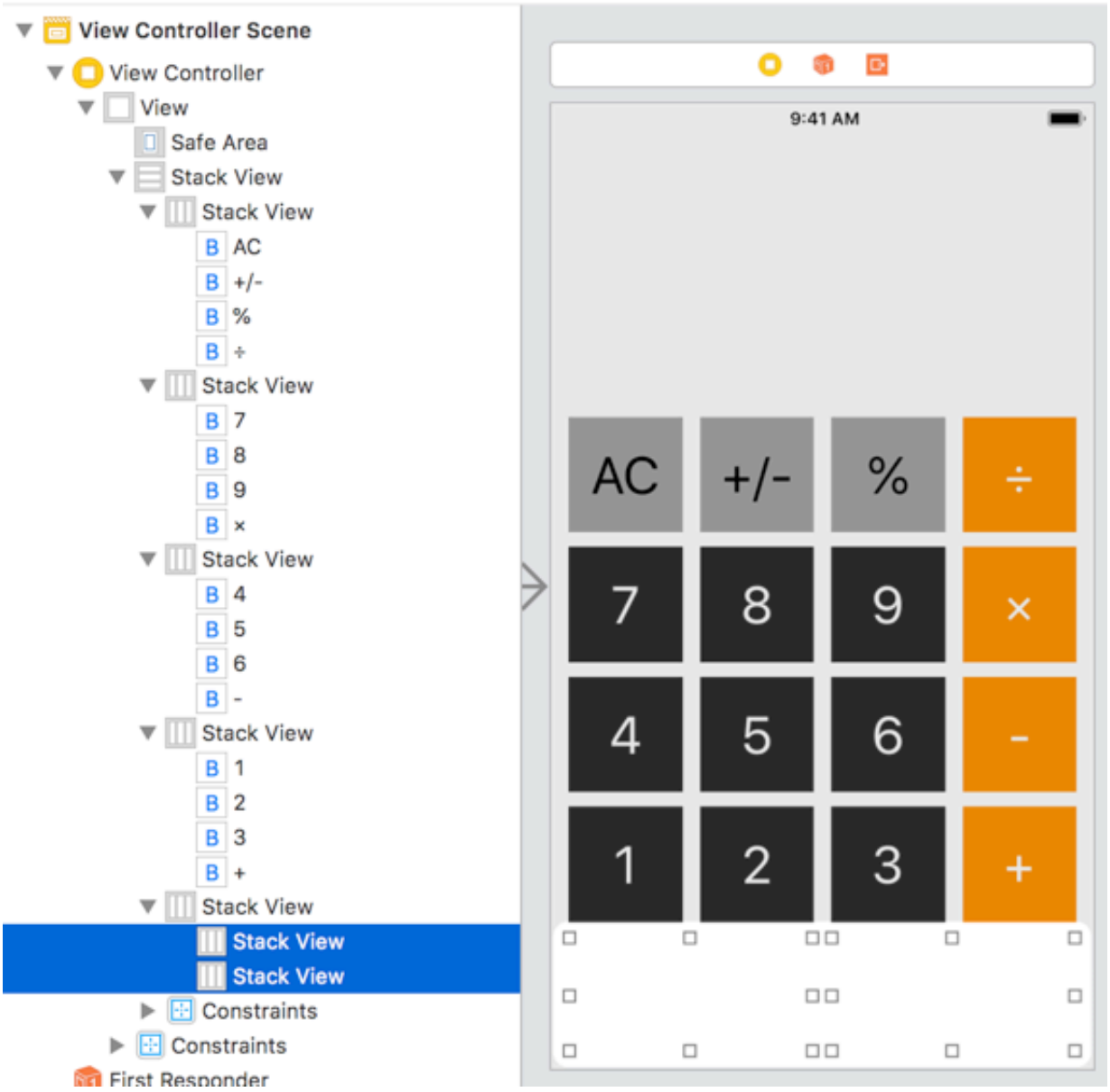
As the outer horizontal stack view has equal fill distribution, the inner stack views will hold two buttons, except for the one containing "0" button.

In landscape mode, the layout is like this:

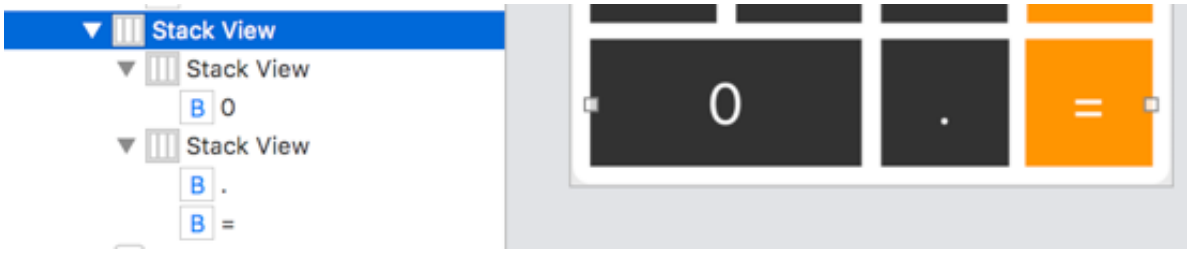


Each of the inner (blue) horizontal stack views have equal width, and they hold space to put two buttons. The inner (blue) horizontal stack views have "Fill Equally" set for the Distribution property as well.

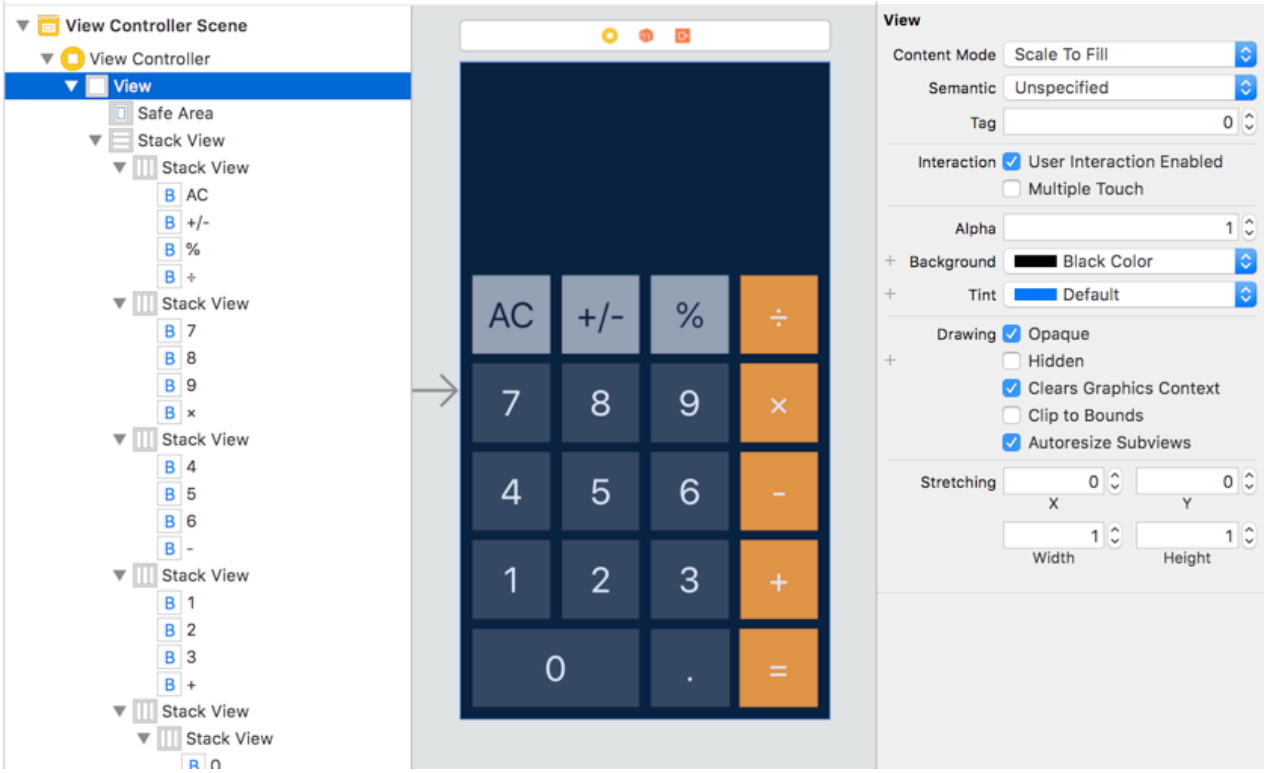
To fix the "0" button issue, remove the four buttons at the last row, and drag two horizontal stack views into the horizontal stack view at the last row.



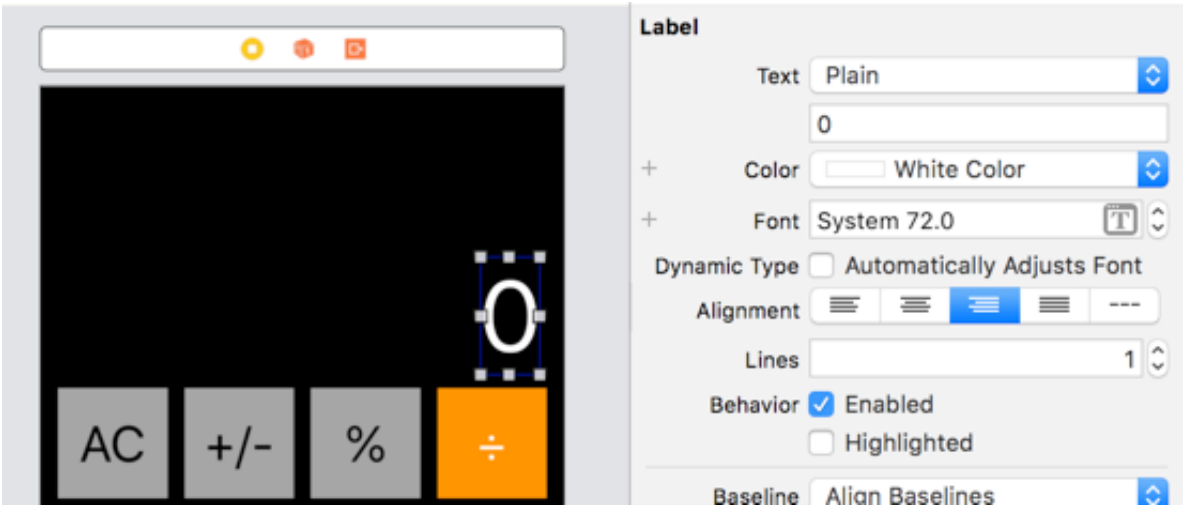
Set the two inner stack views' **Distribution** to '**Fill Equally**', and set the **spacing to 10** to match the spacing of the outer horizontal stack view. Put "0" into the left inner stack view, and put "." and "=" to the right inner stack view like this :



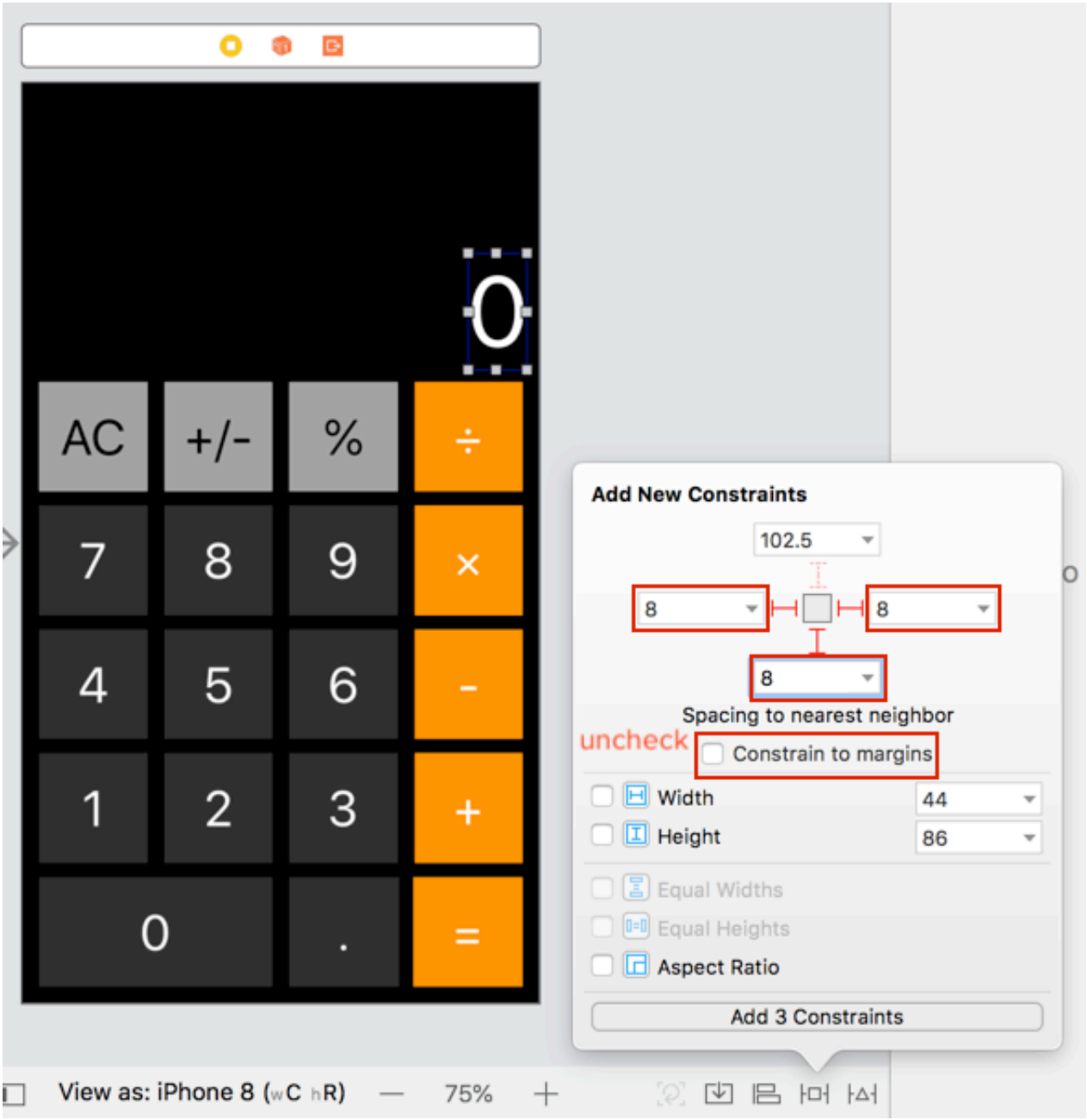
Now we are done with the layout of the buttons in portrait mode, we will need to place the result label on top of the button. Before placing the label, make the background of the view controller's root view to black color :



Next, drag a label and place it on top of the divide symbol (÷) button, I have used the System font with size 72 for the label, and make it white color, also set the Alignment of the label to right.



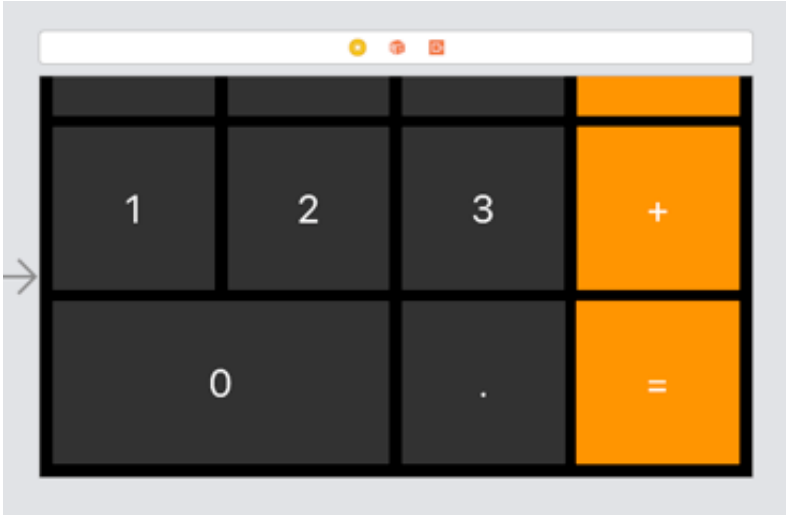
Set the constraint for the result label like this :



1. Leading 8pt to the Safe Area
2. Trailing 8pt to the Safe Area
3. Bottom 8 pt to the main vertical stack view

Now we have implemented the portrait mode layout for the calculator app! Try to view this layout in different phone size (iPhone SE, iPhone 8 plus, iPhone XS Max etc) and you see the buttons size adapt / in proportion to the screen width 🙌.

If we will the layout in iPhone landscape mode now, it looks like this



The buttons are too big! and the 4:5 ratio doesn't fit well in the landscape mode 😅, in the next section, we will make some changes to the layout and use size classes to change constraint properties.

Adapting to landscape mode

When the calculator app is in landscape mode, it doesn't follow the 4:5 ratio we have set earlier and have a lot more buttons. One common pattern across multiple devices is that there seems like a fixed space from the top of screen to the button stack view (annotated with red arrow).

iPhone 8

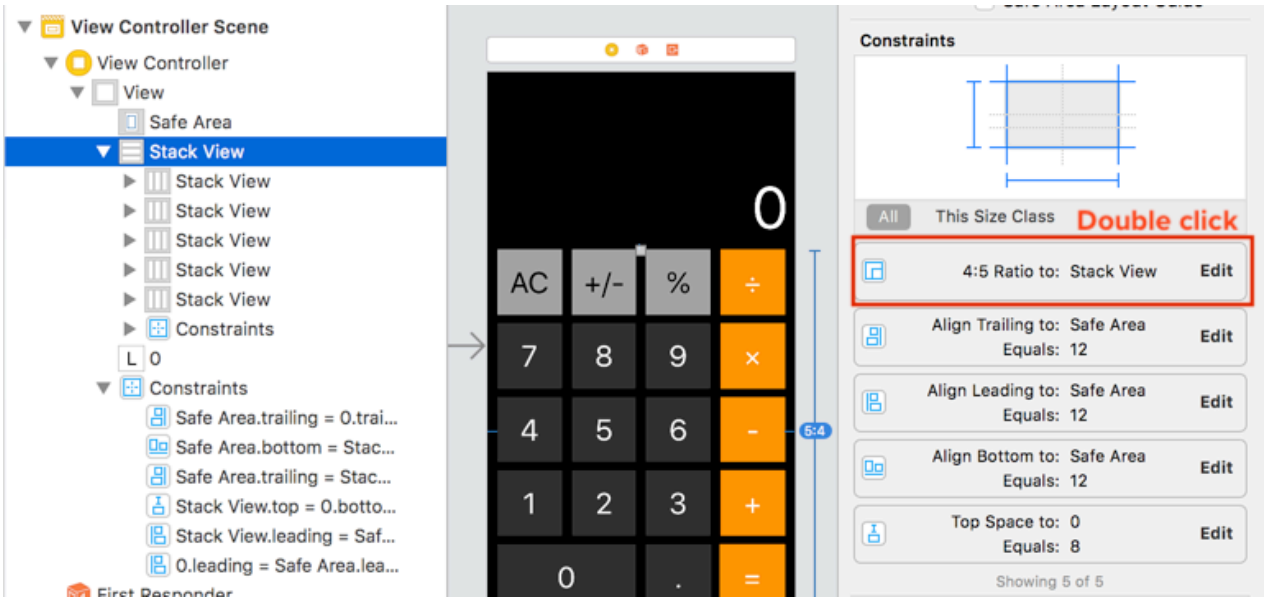
iPhone 8 plus

iPhone XR

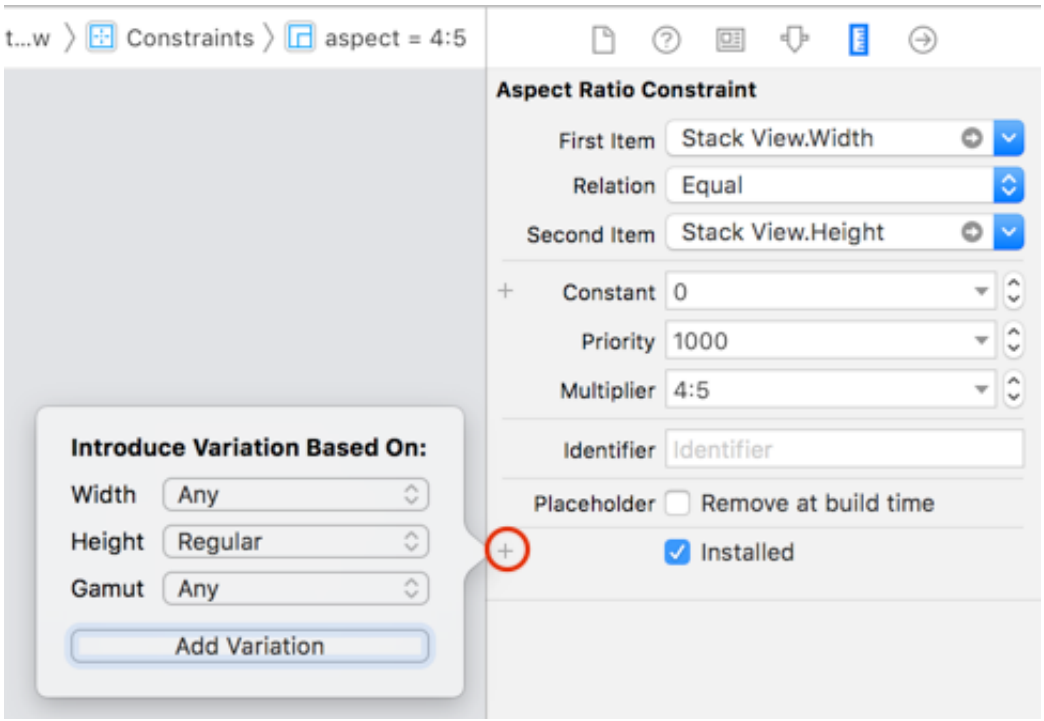
As the vertical stack view doesn't follow the 4:5 ratio in landscape mode, we should not apply this ratio constraint when the iPhone is in landscape mode.

When iPhone is in portrait mode, its height size class is regular (hR), wheras in landscape mode, the height size class is compact (hC). We should only apply the ratio constraint when the height size class is regular (hR).

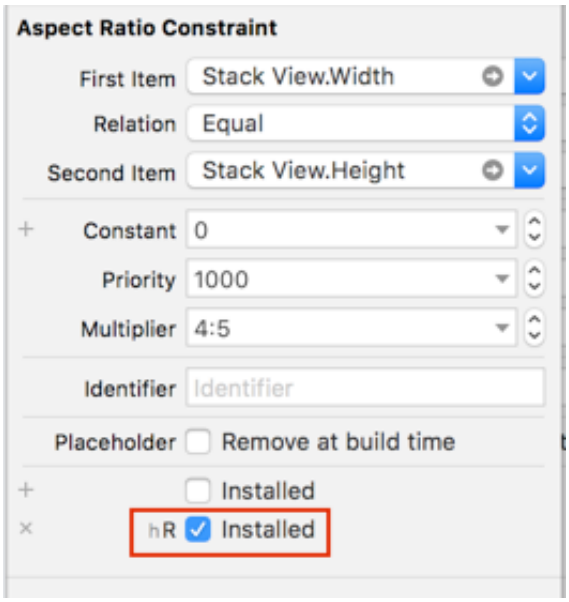
Select the vertical stack view and open its Size inspector, double click the ratio constraint.



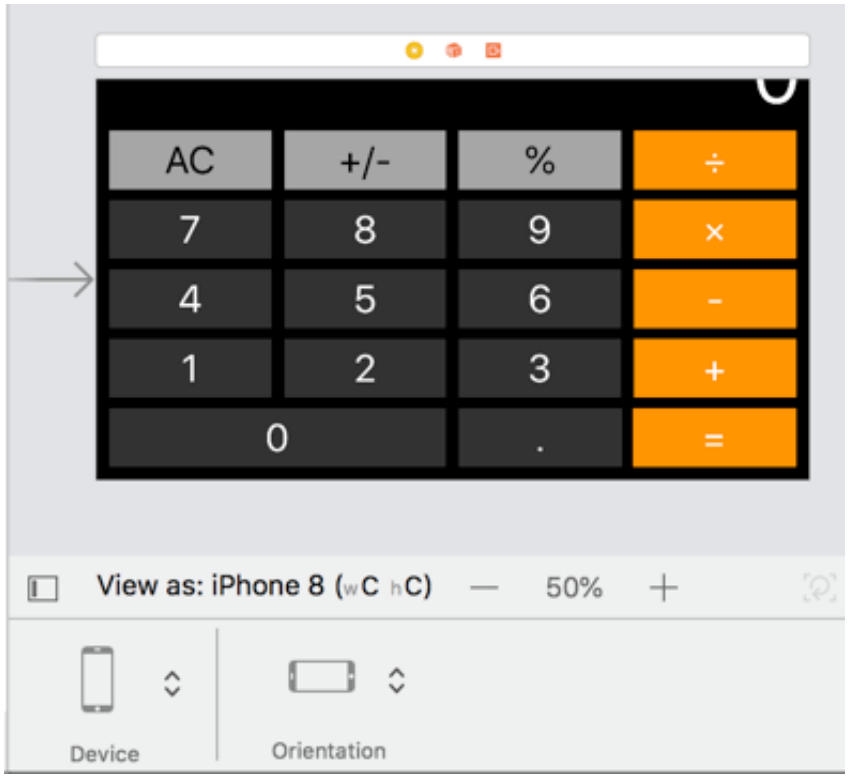
In the constraint properties, click the "+" beside the **Installed** attribute to add size class variation. Select Width > Any, Height > Regular , Gamut > Any for the variation :



The **Installed** attribute means that if the constraint will be installed/activated on runtime or not. We want it to only install in hR (iPhone portrait mode). Uncheck the default **Installed** property as this will install the constraint regardless of size classes, then check the hR variation of **Installed**, to make this constraint only install in hR (height regular size class).



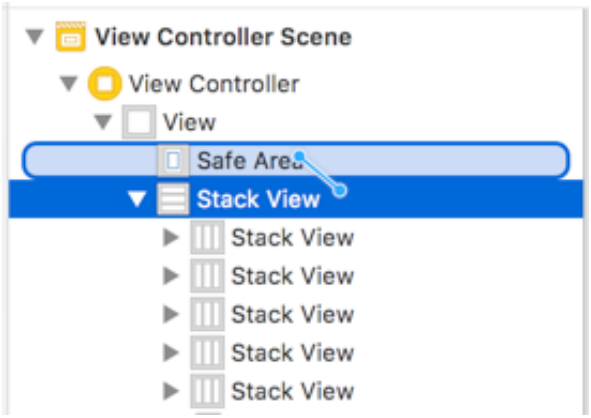
Now when we view the app in landscape mode, it looks slightly better :



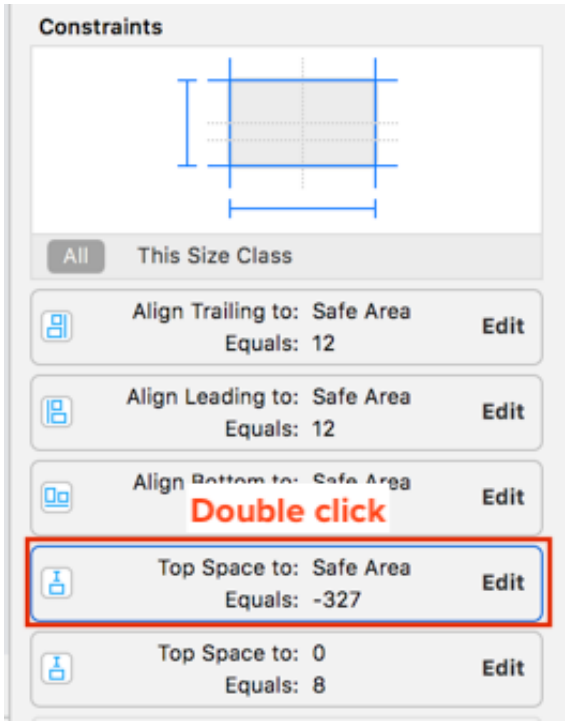
Notice that the result label ('0') is clipped due to the height of the vertical stack view. There is no constraint error even if we didn't set the height or top constraint for the vertical stack view here, it's because Auto Layout can calculate the height of the stack view by using the intrinsic size of the buttons placed inside. (Intrinsic size of the button can be calculated from the font size and text inside)

As mentioned earlier, there's a fixed space from the screen top to the vertical stack view, let's create a top constraint for the vertical stack view to the screen safe area top.

Select the vertical stack view from the Document outline, then hold the control key and drag it to Safe Area, and select "Vertical spacing".

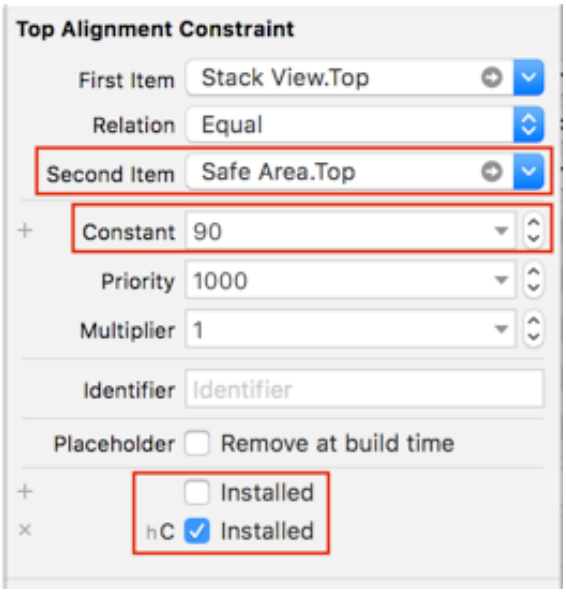


Next, open the size inspector and double click the "Top space to: Safe Area" constraint we created earier :

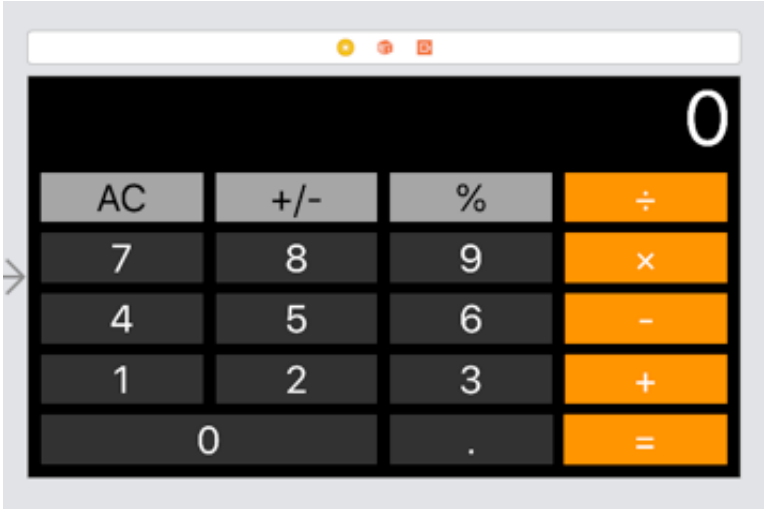


Ensure that the constraint items are **Stack View.Top** to **Safe Area.Top** (this mean the spacing between the top of the stack view to the top of the safe area). I have set the constant to 90 but you can tweak it to any value you like.

Remember to add a hC (height compact, iPhone landscape) variation for it, and only install this constraint on hC size class since this top constraint only exist in iPhone landscape mode.

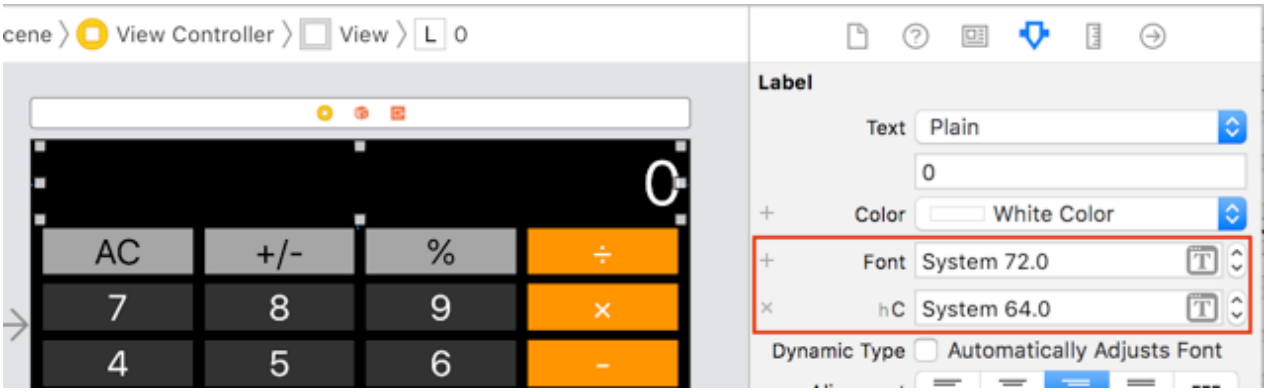


Now it looks slight better on landscape mode :

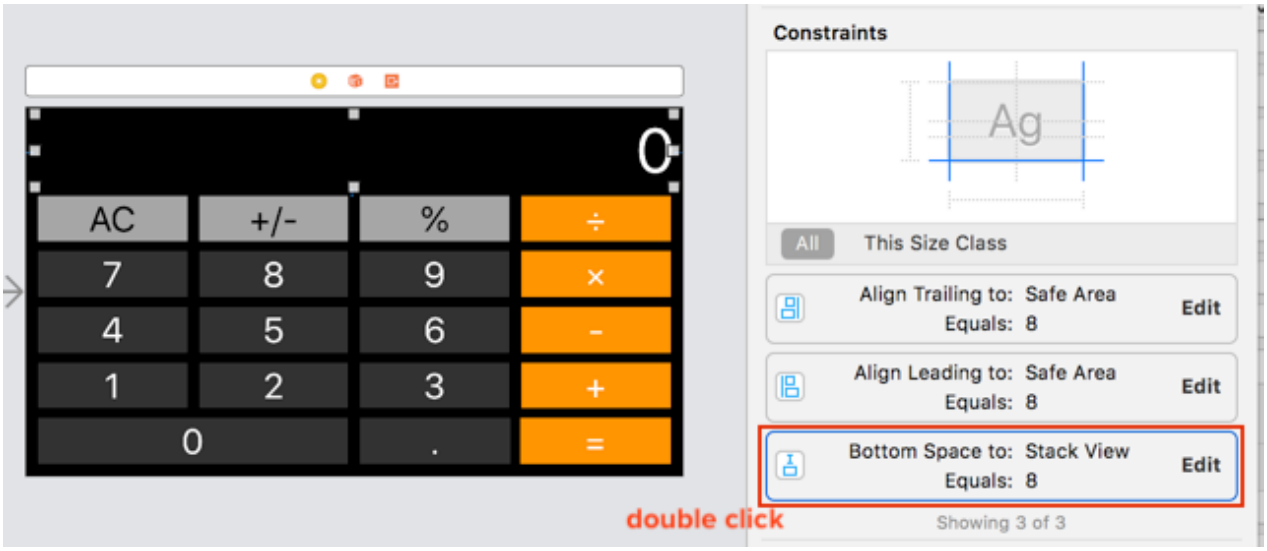


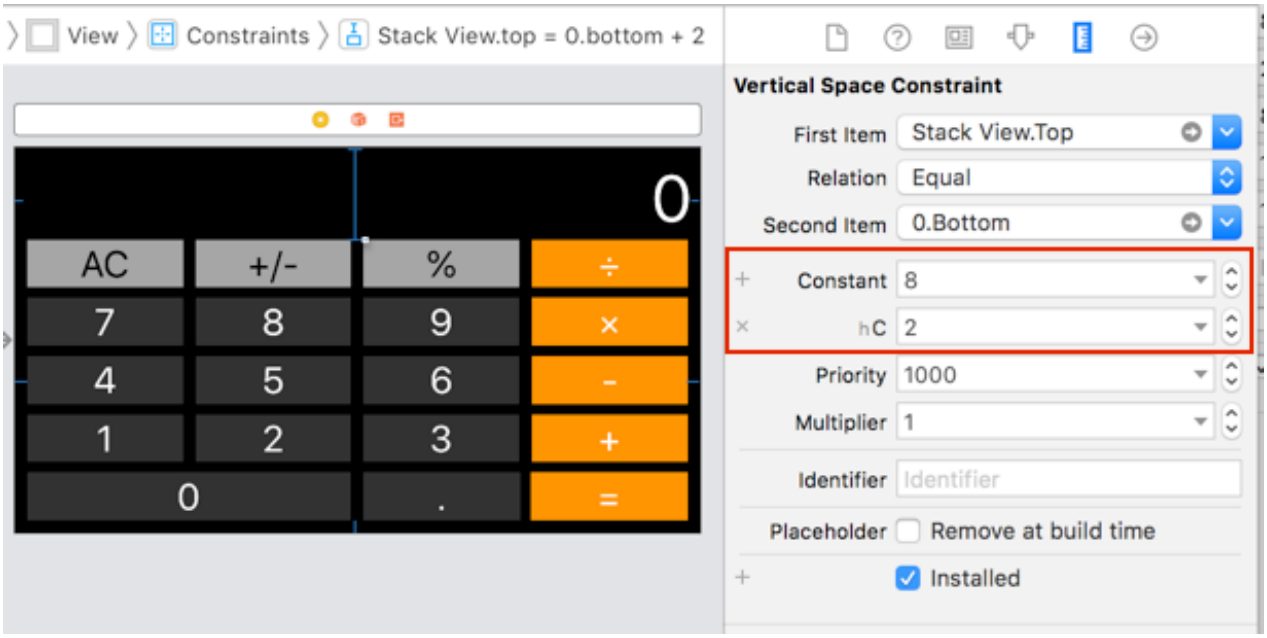
At a closer glance, the result label ('0') seems a bit too large and a bit distant from the buttons. Let's adjust it for landscape mode (hC).

Select the result label ('0'), open its attribute inspector tab and add a font variation for the hC size class (height compact, iPhone landscape). Choose a smaller font size for it (I used 64.0) :



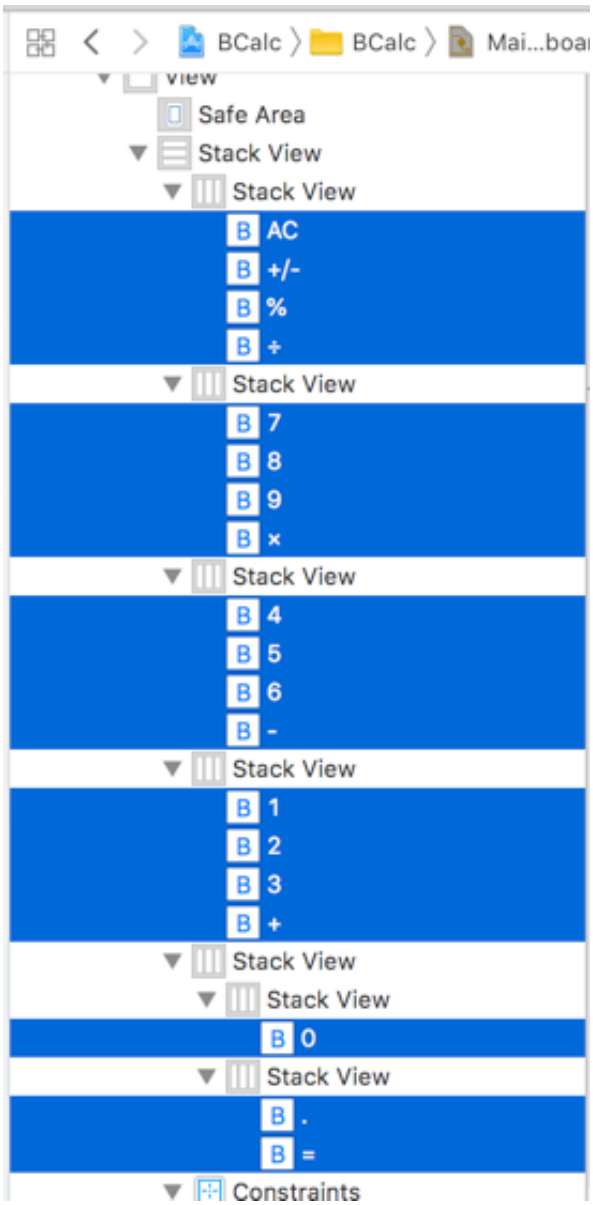
Next, select the bottom constraint of the result label, then add a variation for the hC size class (height compact, iPhone landscape) as well. Choose a smaller constant value for it (I used 2) :



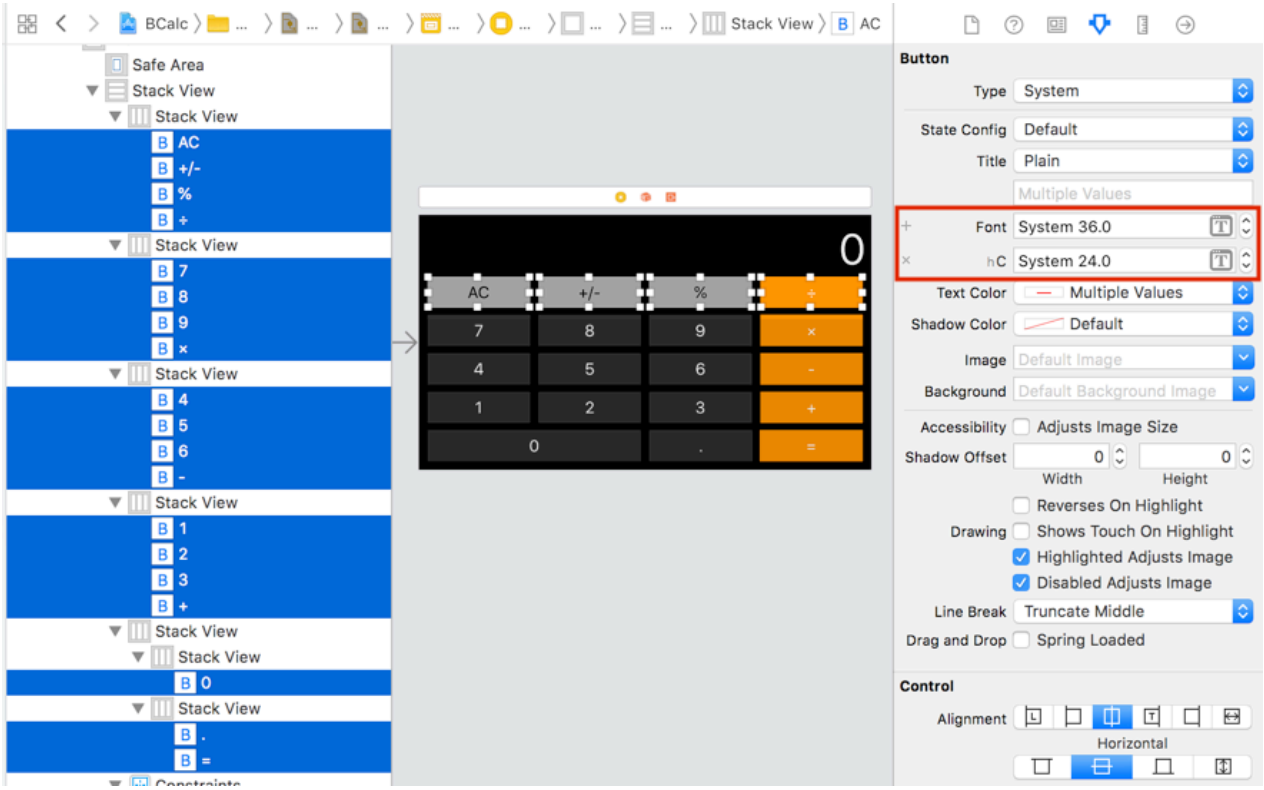


Now it looks a bit better! Just that the buttons text seems a bit too large in landscape mode, we should use a smaller font for the button since the button height is smaller when in landscape mode.

In the document outline, select all the buttons (you can hold the "command" key and click each button to select them all)



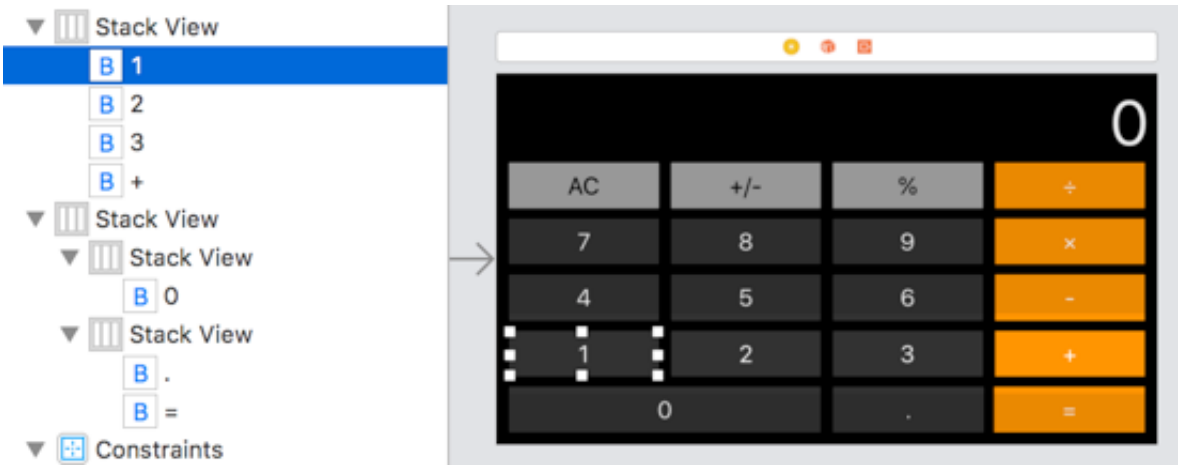
Then in the attributes inspector, add a size class variation to the font, select hC (Height compact), and use a smaller font (I used 24.0) :



Now we have a smaller button fonts / spacing when viewed in landscape mode, the calculator app looks great in landscape mode! Feel free to build and run on different iPhone simulators and see the layout remain consistent on all iPhone screen sizes.

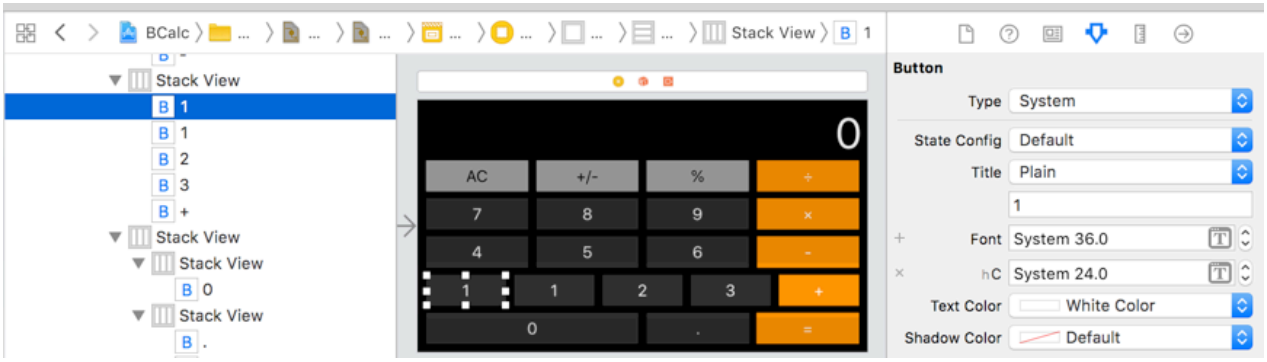
Next, we are going to add the advanced math function buttons that is only visible in iPhone landscape mode.

If we drag a new button from the library, we will need to set up the font color/size and size class variation again and this can be quite tedious. A faster way to do this would be selecting an existing button in the storyboard, hold "**option**" key , and drag it to any of the horizontal stack view to create a copy of the button (you will see a green + icon that indicates you are making a copy of the button).



Select a button (eg: the "1" button), press and hold "**option**" key, then drag it to make a copy.

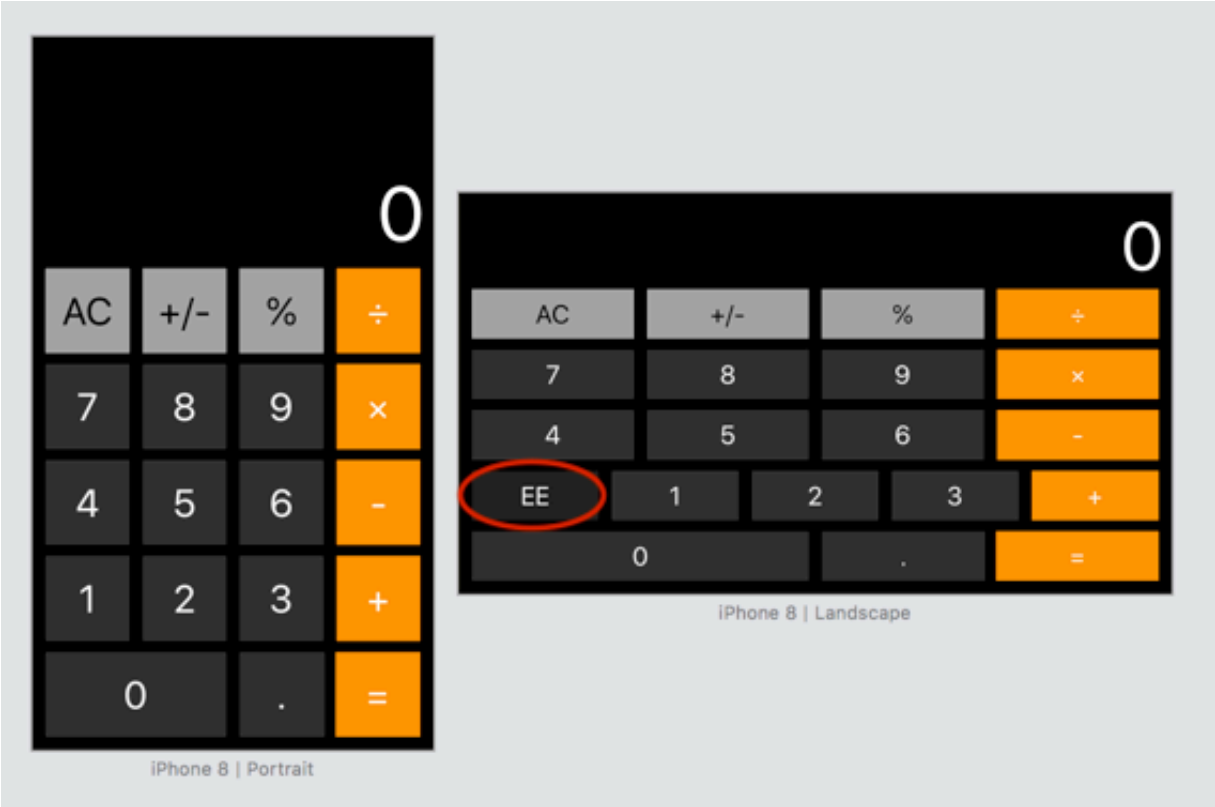
The button copy will have the same exact attribute (color, font, size class variation, etc).



We can keep repeating this process to duplicate buttons quickly and then change their text afterwards. But before you move on to duplicate them, let's modify the first button copy we created earlier.

We only want these advanced math function buttons to appear in landscape mode, it will be hidden in portrait mode. To achieve this, create a size class variation for its **Hidden** attribute, and select hC (height compact, iPhone landscape mode).

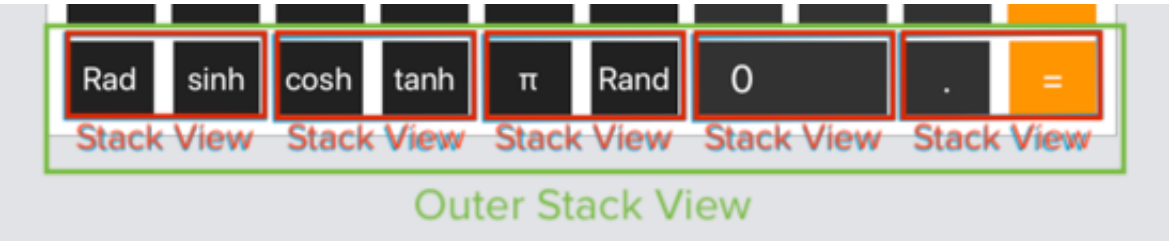
Check the default **Hidden** attribute and leave the hC variant of **Hidden** attribute unchecked. This will make the button to be hidden by default and only visible in hC size class (iPhone landscape).



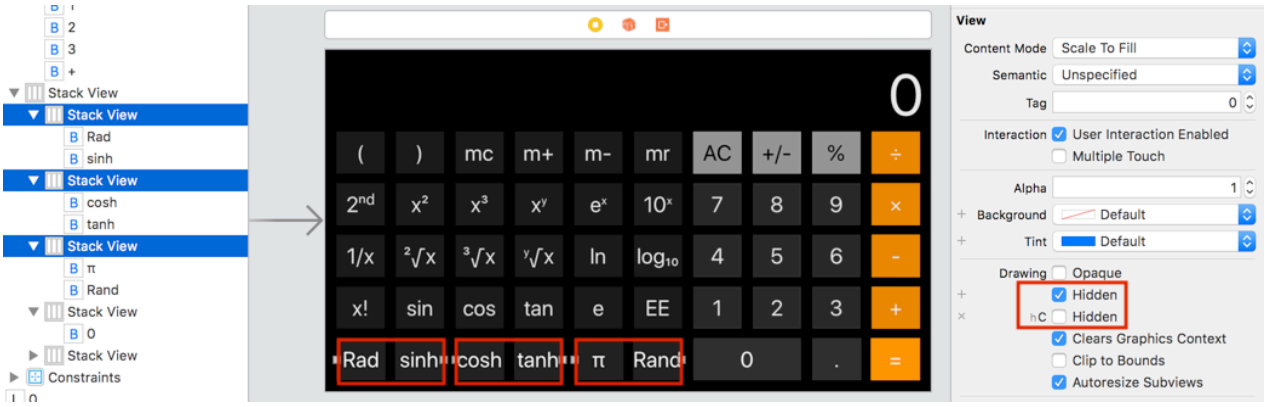
(notice the 'EE' button is only visible in landscape mode)

We can now proceed to copy / duplicate this button ('EE') by holding '**option**' key + drag. This will copy its hC hidden size class variation as well.

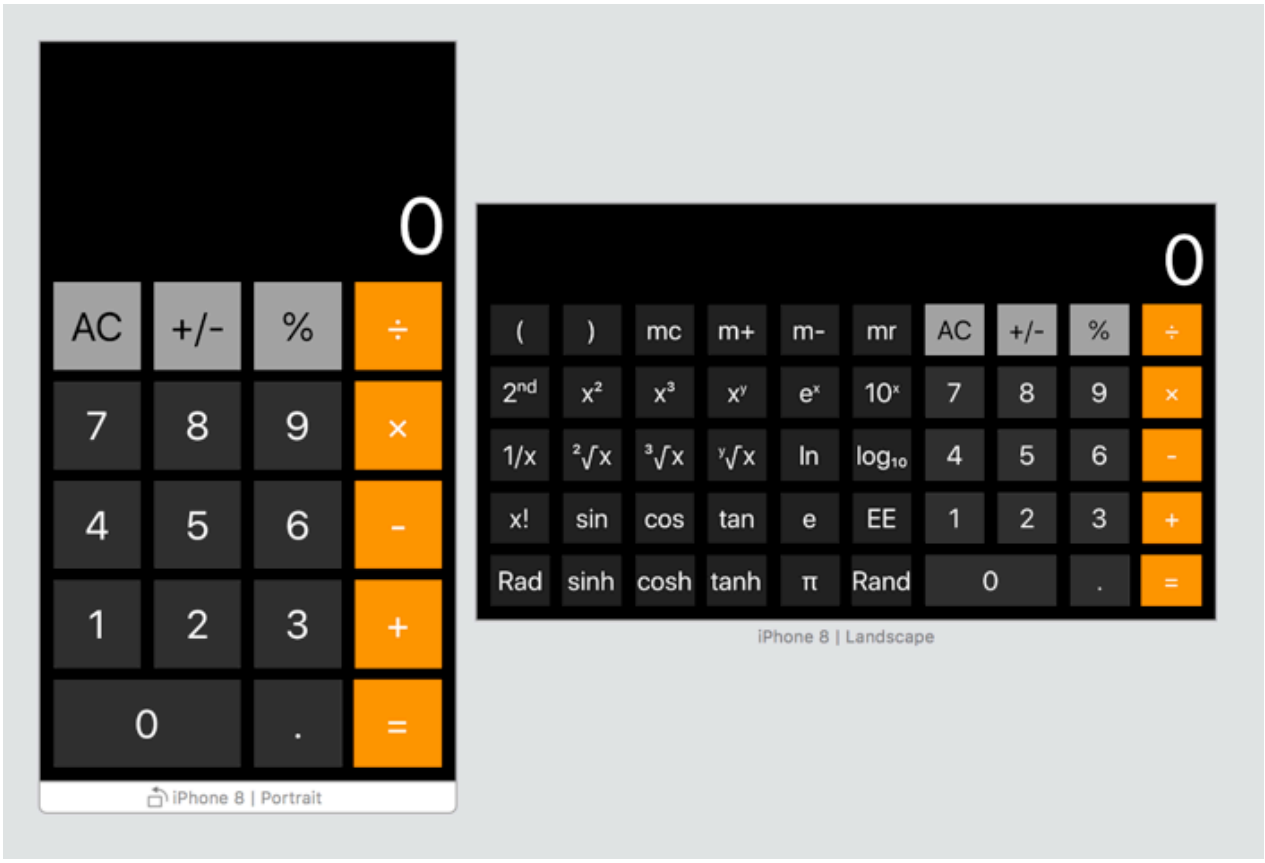
Remember that you'll need to group two buttons in a horizontal stack view for the last row :



Remember to add variation for the **Hidden** property for hC size class for these horizontal stack views as well :



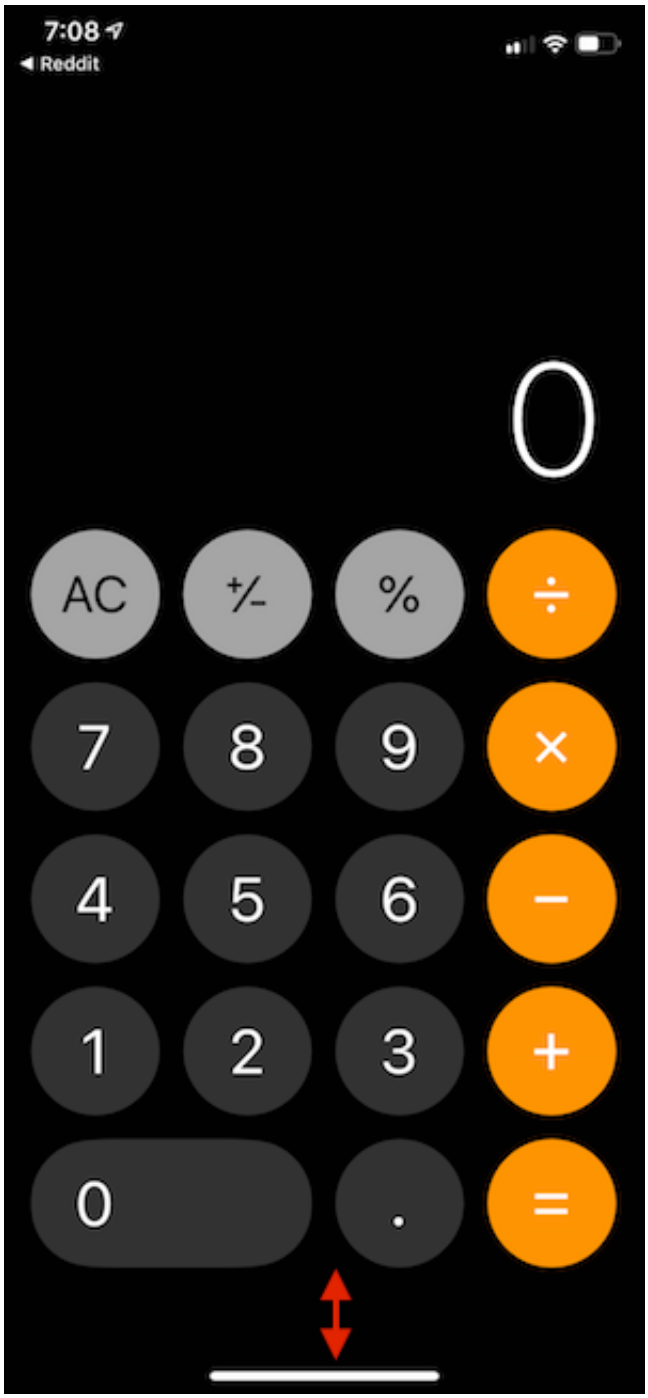
Build and run the app, and try rotating the device (press **command** + **left arrow key** / **right arrow key** to rotate the simulator), you will see that Auto Layout handles the rotating animation and hiding/showing buttons for you. Now we have successfully replicated the layout of calculator app using Auto Layout, Stack Views and Size classes (almost)! 🙌



Still, there is still some minor adjustments (rounded buttons, more bottom spacing for iPhone X) we can tweak to make it look more alike. We will go over these adjustments in the next section.

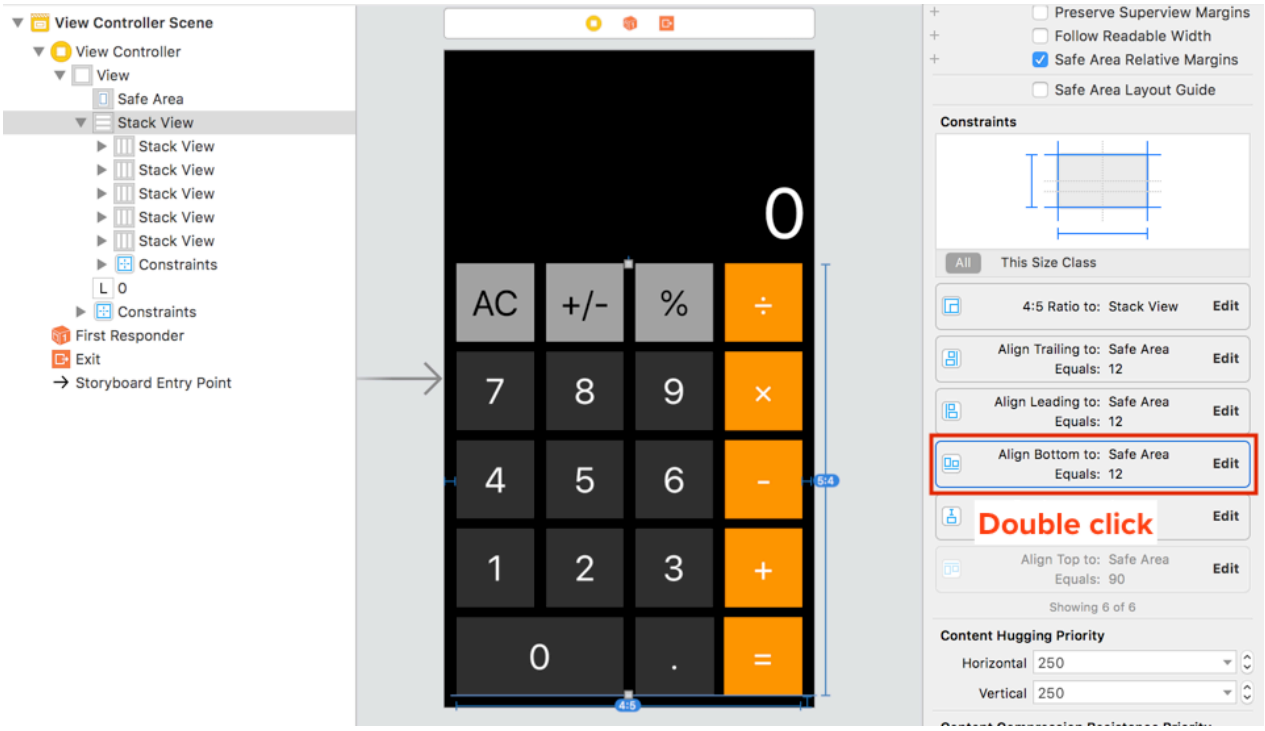
Manual tweak for iPhone X , XS, XS Max and XR

If we open the calculator app in iPhone X (or any iPhone that has notch), there will be a bottom distance from the buttons to the bottom edge of the screen. This is to cater for the swipe up to open multitasking gesture available in iPhone X, the space at the bottom is to prevent accidental swipe up when user tap the bottom row.

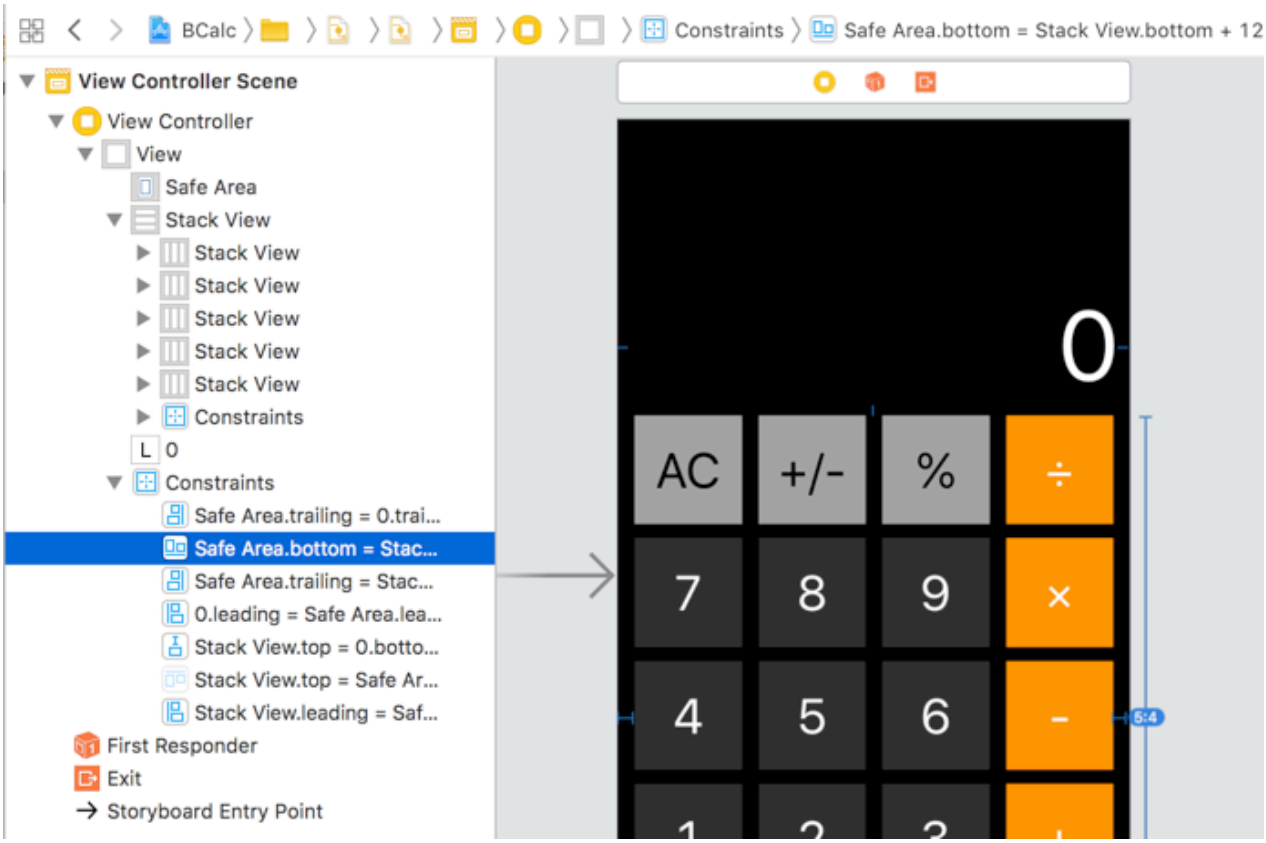


This bottom space however, does not exist in iPhone SE / 8 / 8 Plus. Meaning that using size class won't help and we have to do some manual check on if the device has a top notch, and modify the bottom space.

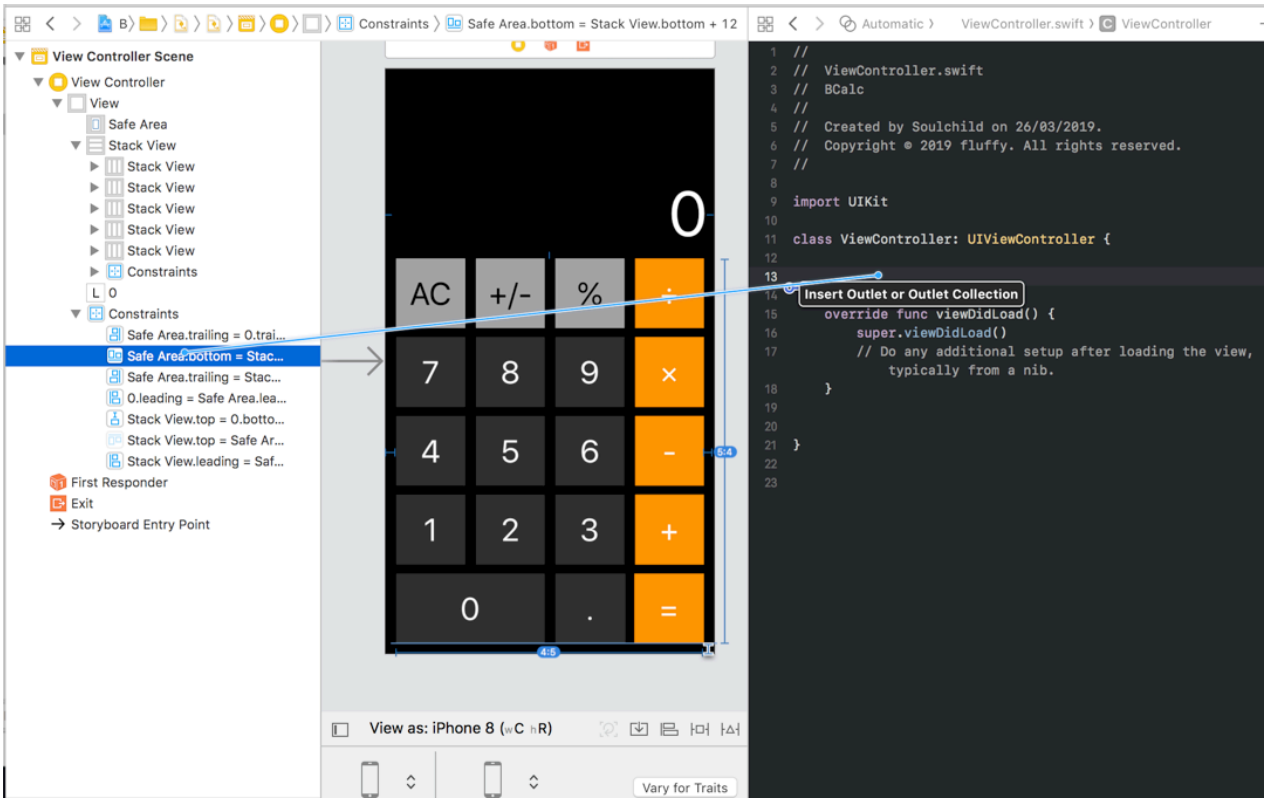
First, select the bottom constraint for the vertical stack view :



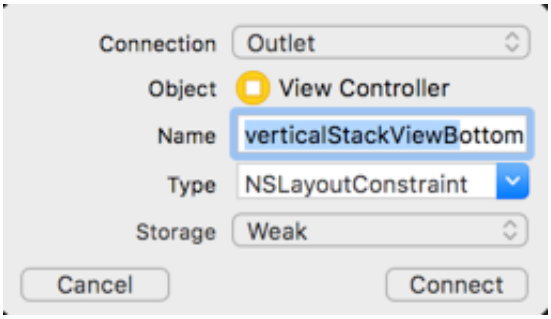
Then you see the constraint highlighted in the Document Outline like this :



Open Assistant Editor, hold **control** key and drag the constraint to the View Controller file to create an IBOutlet :



Type in the name you want for this constraint IBOutlet, I used the name **verticalStackViewBottomConstraint** .



After creating the IBOutlet for the constraint, add the code below to ViewController.swift :

```
// ViewController.swift

class ViewController: UIViewController {

    @IBOutlet weak var verticalStackViewBottomConstraint: NSLayoutConstraint!

    // check if the device has top notch (iPhone X / XS / XR)
    var hasTopNotch: Bool {
```

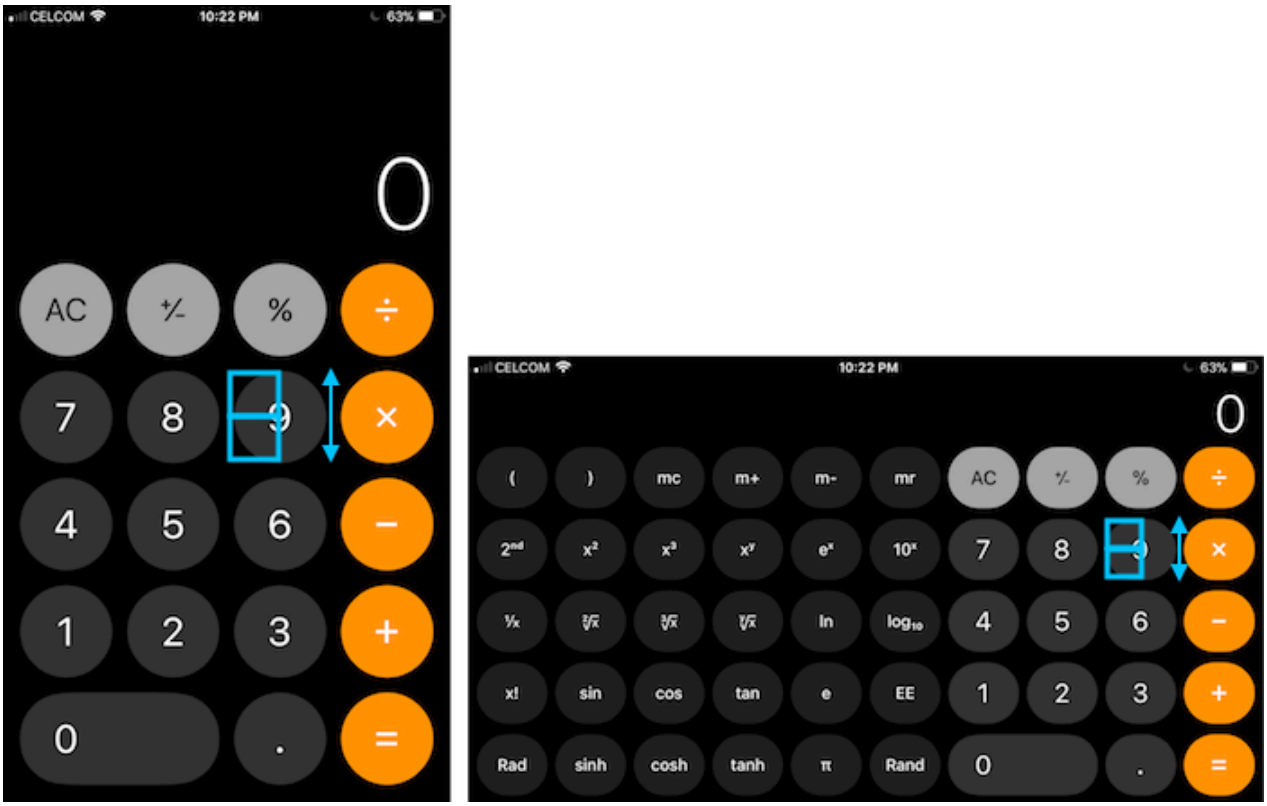
```
        // device with top notch has safe area insets larger than 20 pt
        // the code below use '0' as value if there is no safe area top inset
        if #available(iOS 11.0,  *) {
            return UIApplication.shared.delegate?.window??.safeAreaInsets.top
?? 0 > 20
        }

        return false
    }

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view, typically from a
nib.
    }

    override func viewDidLayoutSubviews() {
        super.viewDidLayoutSubviews()

        // if is iPhone X and in portrait mode, add more bottom spacing
        if hasTopNotch && UIDevice.current.orientation.isPortrait {
            verticalStackViewBottomConstraint.constant = 36.0
        } else {
            // else revert back to the original spacing
            verticalStackViewBottomConstraint.constant = 12.0
        }
    }
}
```

Build and run the app in different iPhone simulators, you will notice that now there is an additional bottom spacing for iPhone X / XS / XR when in portrait mode! 🙌

This is a situation where Auto Layout is not enough and manual coding is required. Next, we will look into making the button rounded.

Rounded buttons

To make button become rounded, we can use the **button.layer.cornerRadius** attribute to set the radius for the corner. Notice that in both portrait and landscape mode, corner radius is half of the button's height :

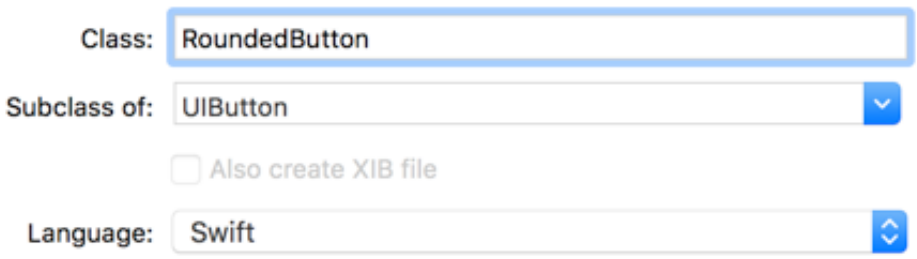


The code to make the button corner radius to become half of its height is as follow:

```
button.layer.cornerRadius = button.bounds.height / 2.0
```

It would become really tedious if we create an outlet for each buttons and set the attribute one by one, instead, we are going to create a subclass of UIButton that contain the rounded corner code and change all buttons to the new subclass in Storyboard.

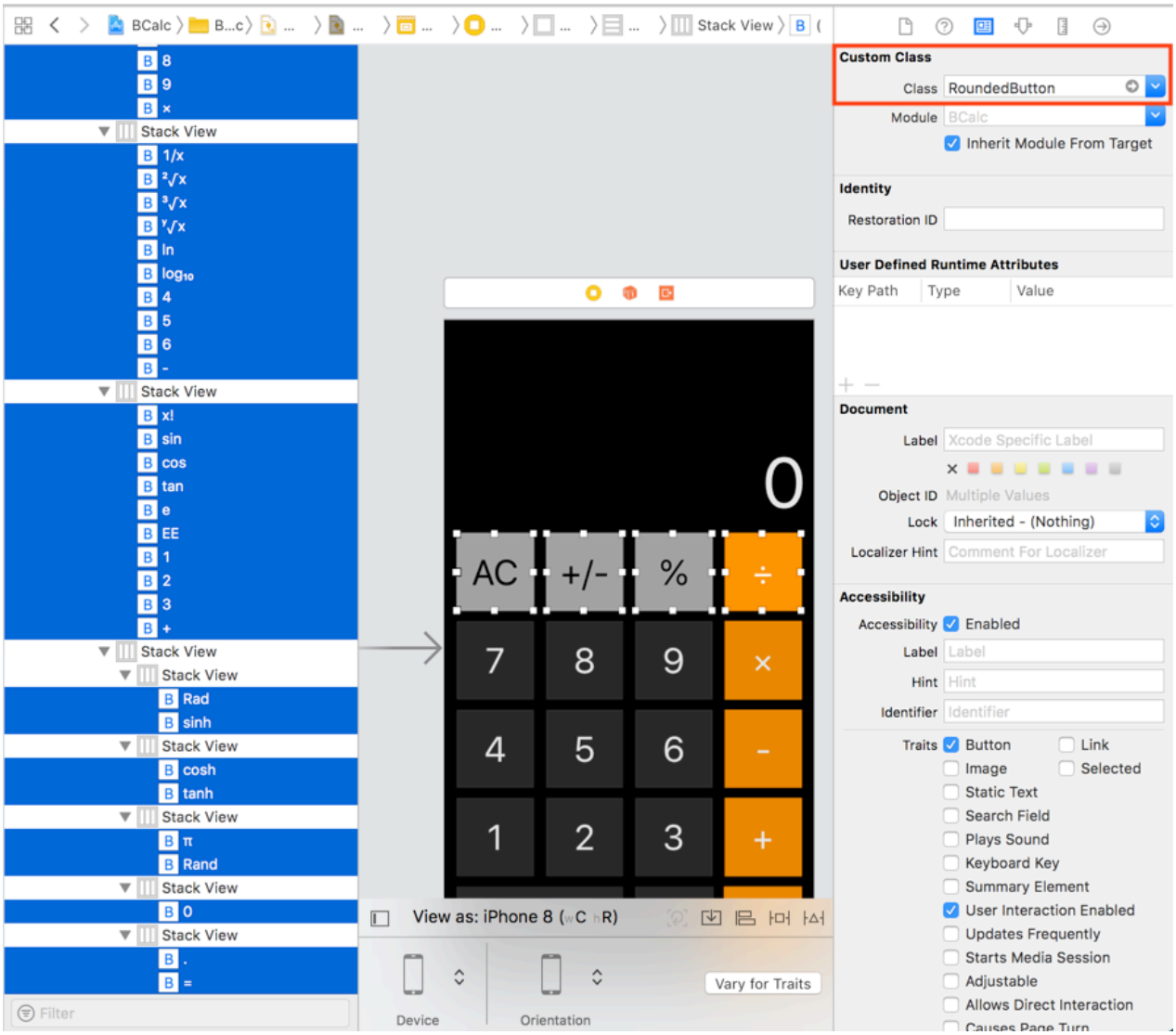
Create a new file with a subclass of 'UIButton', name it as **RoundedButton**.



Put the rounding code inside layoutSubviews() method :

```
// RoundedButton.swift

class RoundedButton: UIButton {

    // Put the corner rounding code in layoutSubviews
    // as this will be called when the screen first shown and when its bound
changes (screen rotation, etc)
    // when bound changes, we want to recalculate the radius when the height
has changed
    override func layoutSubviews() {
        super.layoutSubviews()

        self.layer.cornerRadius = self.bounds.height / 2.0
    }
}
```

Then in storyboard, select all the buttons (hold down **command** key and select them all), and change their class to **RoundedButton**



Build and run the app, now we have a calculator app clone!

In this chapter, we have successfully replicated the layout of the stock calculator app, and used proportional sizing to make it look good across multiple devices. Stack view (auto remove hidden subview) combined with Size class variation are really powerful tool that enables us to build adaptive layout.

Before starting to design layout, always ask yourself what is the common traits (eg: vertical stack view with 5 rows, corner radius is half of button height etc) that exhibit across different screen size? Start from there and then find out what is different (eg: advanced math buttons only show in landscape, top constraint only install in landscape, 4:5 ratio only in portrait, smaller font size in landscape etc) , then use Size class variation to help . And finally when size class doesn't cut it, move to manual tweaking in code.

Although Auto Layout / Stack View / Size class provide great help on designing layout, there might still be cases where we need to manually tweak the layout / constraint value in code, like for the bottom space for iPhone X / XS / XR.

You might have tried to run this app in iPad, and found that the layout doesn't suit it as iPad is more square-ish, causing some buttons to get clipped, and iPad is wRhR regardless of portrait or landscape.

As Apple doesn't provide calculator app for iPad, it's hard to gauge how it should look like in iPad. Now that this chapter has guided you to make the iPhone calculator app, my challenge for you is to adjust the existing layout so that it looks OK (not necessary need to look great, as long as all buttons and result label are visible) on iPad. You can use the size class variation (wRhR) for iPad screen size. Have fun!

A - Appendix

Here are some links I think that might be useful for learning more on Auto Layout :

Best Practices for Mastering Auto Layout (WWDC 2012) (https://developer.apple.com/videos/play/wwdc2012/228/)

Auto Layout Techniques in Interface Builder (WWDC 2017) (https://developer.apple.com/videos/play/wwdc2017/412/)

Auto Layout Guide (2015, removed by Apple) (https://web.archive.org/web/20150323123055/https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/AutolayoutPG/Introduction/Introduction.html#//apple_ref/doc/uid/TP40010853-CH13-SW1)

Auto Layout Guide (current) (https://developer.apple.com/library/archive/documentation/UserExperience/Conceptual/AutolayoutPG/index.html#//apple_ref/doc/uid/TP40010853-CH7-SW1)

Making Apps Adaptive (WWDC 2016) (https://developer.apple.com/videos/play/wwdc2016/222/)

This is a free (and open source too!) tool for translating cryptic Auto Layout error in console into visual format:

WTFAutoLayout (http://www.wtfautolayout.com)

Always remember to set `view.translatesAutoresizingMaskIntoConstraints = false` for a programmatically created view if you want to apply Auto Layout Constraint into it.

B - Acknowledgements

I would like to thank my friend Sergio Utama and Alex Kluew for taking the time to read and review this book.

And thank you, dear reader, for choosing to spend your time reading this book. I hope this book has made you understand better on Auto Layout and gained more confidence on designing user interface .

If you've used any concept taught in this book for a project, I'd love to hear what you've done.

If you have any questions or comments, I'd like to hear from you. Feel free to email me at axel@fluffy.es.

Axel Kee