Auto Layout Guide

Developer

Contents

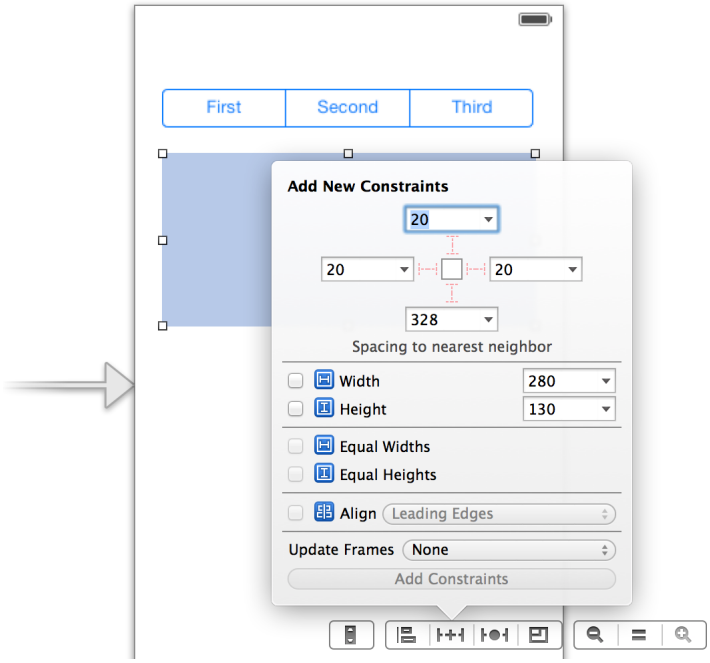
2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Introduction

Important: This is a preliminary document for an API or technology in development. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein for use on Apple-branded products. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future betas of the API or technology.

Auto Layout is a system that lets you lay out your app's user interface by creating a mathematical description of the relationships between the elements. You define these relationships in terms of constraints either on individual elements, or between sets of elements. Using Auto Layout, you can create a dynamic and versatile interface that responds appropriately to changes in screen size, device orientation, and localization.



Auto Layout is built into Interface Builder in Xcode 5, and is available to apps targeted at either iOS and OS X. Auto Layout is enabled by default when you create a new project. If you have an existing project that doesn't use Auto Layout, read Adopting Auto Layout (page 39).

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

The typical workflow for creating user interfaces starts by using Interface Builder to create, reposition, resize, and customize your views and controls. When you are satisfied with the positions and settings, you're ready to start adding Auto Layout constraints so that your interface can react to changes in orientation, size, and localization.

At a Glance

Auto Layout in Xcode 5 provides powerful workflows for rapidly and easily creating and maintaining constraint-based layouts in OS X and iOS apps. With Xcode 5, you can:

- Add constraints when you are ready
- Quickly add constraints using control-drag or menu options
- Update constraints and frames separately
- Specify placeholder constraints for dynamic views
- See, understand, and resolve issues with conflicting constraints or ambiguous views

Organization of This Document

Read the following chapters to learn how to use Auto Layout:

- Auto Layout Concepts (page 6) to learn about the main concepts you need to understand when using Auto Layout
- Working with Constraints in Interface Builder (page 9) to learn about using Interface Builder to create and edit layout constraints
- Working with Auto Layout Programmatically (page 14) to learn about working with Auto Layout in code
- Resolving Auto Layout Issues (page 17) to learn about identifying and debugging issues with your layout
- Auto Layout by Example (page 24) to see examples of common Auto Layout use cases
- Implementing a Custom View to Work with Auto Layout (page 37) to learn how to implement a custom view that interoperates with Auto Layout
- Adopting Auto Layout (page 39) to learn how to adopt Auto Layout in an existing project that doesn't use Auto Layout
- Visual Format Language (page 41) to learn about the language used to create constraints in code

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Auto Layout Concepts

The fundamental building block in Auto Layout is the **constraint**. Constraints express rules for the layout of elements in your interface; for example, you can create a constraint that specifies an element's width, or its horizontal distance from another element. You add and remove constraints, or change the properties of constraints, to affect the layout of your interface.

When calculating the runtime positions of elements in a user interface, the Auto Layout system considers all constraints at the same time, and sets positions in such a way that best satisfies all of the constraints.

Constraint Basics

You can think of a constraint as a mathematical representation of a human-expressable statement. If you're defining the position of a button, for example, you might want to say "the left edge should be 20 points from the left edge of its containing view." More formally, this translates to `button.left = (container.left + 20)`, which in turn is an expression of the form `y = m*x + b`, where:

- `y` and `x` are attributes of views.
- `m` and `b` are floating point values.

An *attribute* is one of `left`, `right`, `top`, `bottom`, `leading`, `trailing`, `width`, `height`, `centerX`, `centerY`, and `baseline`.

The attributes `leading` and `trailing` are the same as `left` and `right` for left-to-right languages such as English, but in a right-to-left environment such as Hebrew or Arabic, `leading` and `trailing` are the same as `right` and `left`. When you create constraints, `leading` and `trailing` are the default values. You should usually use `leading` and `trailing` to make sure your interface is laid out appropriately in all languages, unless you're making constraints that should remain the same regardless of language (such as the order of master and detail panes in a split view).

Constraints can have other properties set:

- **Constant value.** The physical size or offset, in points, of the constraint.
- **Relation.** Auto Layout supports more than just constant values for view attributes; you can use relations and inequalities such as greater-than-or-equal to specify, for example, that a view's `width >= 20`, or even that `textview.leading >= (superview.leading + 20)`.

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

- **Priority level.** Constraints have a priority level. Constraints with higher priority levels are satisfied before constraints with lower priority levels. The default priority level is required (`NSLayoutPriorityRequired`), which means that the constraint must be satisfied exactly. The layout system gets as close as it can to satisfying an optional constraint, even if it cannot completely achieve it.

  Priority levels allow you to express useful conditional behavior. For example, they are used to express that some controls should always be sized to fit their contents, unless something more important should take precedence. For more information about priority levels, see `NSLayoutPriority`.

Constraints are cumulative, and do not override each other. If you have an existing constraint, setting another constraint of the same type does not override the previous one. For example, setting a second width constraint for a view does not remove or alter the first width constraint—you need to remove the first constraint manually.

Constraints can, with some restrictions, cross the view hierarchy. In the Mail app in OS X, for example, by default the Delete button in the toolbar lines up with the message table; in Desktop Preferences, the checkboxes at the bottom of the window align with the split view pane they operate on.

You cannot set a constraint to cross the view hierarchy if the hierarchy includes a view that sets the frames of subviews manually in a custom implementation for the `layoutSubviews` method on `UIView` (or the `layout` method on `NSView`). It's also not possible to cross any views that have a bounds transform (such as a scroll view). You can think of such views as barriers—there's an inside world and an outside world, but the inside cannot be connected to the outside by constraints.

Intrinsic Content Size

Leaf-level views such as buttons typically know more about what size they should be than does the code that is positioning them. This is communicated through the *intrinsic content size*, which tells the layout system that a view contains some content that it doesn't natively understand, and indicates how large that content is, intrinsically.

For elements such as text labels, you should typically set the element to be its intrinsic size (select Editor > Size To Fit Content). This means that the element will grow and shrink appropriately with different content for different languages.

Application Architecture

The Auto Layout architecture distributes responsibility for layout between controllers and views. Rather than writing an omniscient controller that calculates where views need to go for a given geometry, views become more self-organizing. This approach reduces the complexity of controller logic, and makes it easier to redesign views without requiring corresponding changes to the layout code.

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

You may still want a controller object that adds, removes, or adjusts constraints at runtime. To learn more about managing constraints in code, read Working with Auto Layout Programmatically (page 14).

The Role of the Controller

Although a view specifies its intrinsic content size, the user of the view says how important it is. For example, by default, a button:

- Strongly wants to hug its content in the vertical direction (buttons really ought to be their natural height)
- Weakly hugs its content horizontally (extra side padding between the title and the edge of the bezel is acceptable)
- Strongly resists compressing or clipping content in both directions

In a user interface containing two buttons next to each other, for example, it's up to the controller to decide how the buttons should grow if there's extra room. Should just one of the buttons grow? Should both grow equally? Or maybe proportionally to each other? If there isn't enough room to fit both buttons without compressing or clipping the content, should one button be truncated first? Or both equally? And so on.

You set the hugging and compression priorities for a `UIView` instance using `setContentHuggingPriority:forAxis:` and `setContentCompressionResistancePriority:forAxis:` (for `NSView`, you use `setContentHuggingPriority:forOrientation:` and `setContentCompressionResistancePriority:forAxis:`). By default, all UIKit- and AppKit-supplied views have a value of either `NSLayoutPriorityDefaultHigh` or `NSLayoutPriorityDefaultLow`.

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Working with Constraints in Interface Builder

The easiest way to add, edit, or remove constraints is to use the visual layout tools in Interface Builder. Creating a constraint is as simple as Control-dragging between two views, or to add multiple constraints at once, you simply use the various pop-up windows.

Adding Constraints

When you drag out an element from the Object Library and drop it on the Interface Builder canvas, it starts out unconstrained to make it easy to prototype your interface by dragging elements around. If you build and run without adding any constraints to an element, you'll find that Interface Builder fixes the element's width and height, and pins its position relative to the top left corner of the superview; this means that resizing the window doesn't move or resize the elements. To make your interface react correctly to changes in size or orientation, you need to start adding constraints.

> **Important:** Although Xcode does not generate a warning or an error when you build a user interface that does not have appropriate constraints, you should not ship your application in such a state.

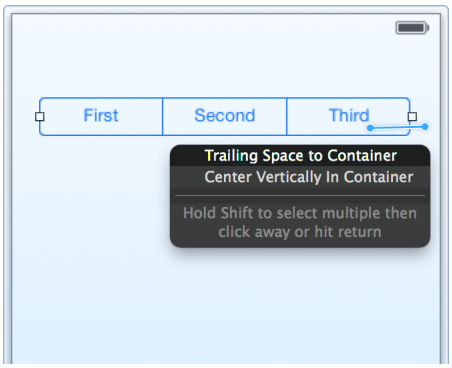
There are several ways to add constraints depending on the level of precision you want and the number of constraints you want to add at a time.

Adding Constraints with Control-Drag

The fastest way to add a constraint is by holding down the Control key and dragging from a view on the canvas, much like the way you create links to outlets or actions. This Control-drag method is a quick, precise tool for creating a single constraint when you know exactly what type of constraint you want and where you want it.

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

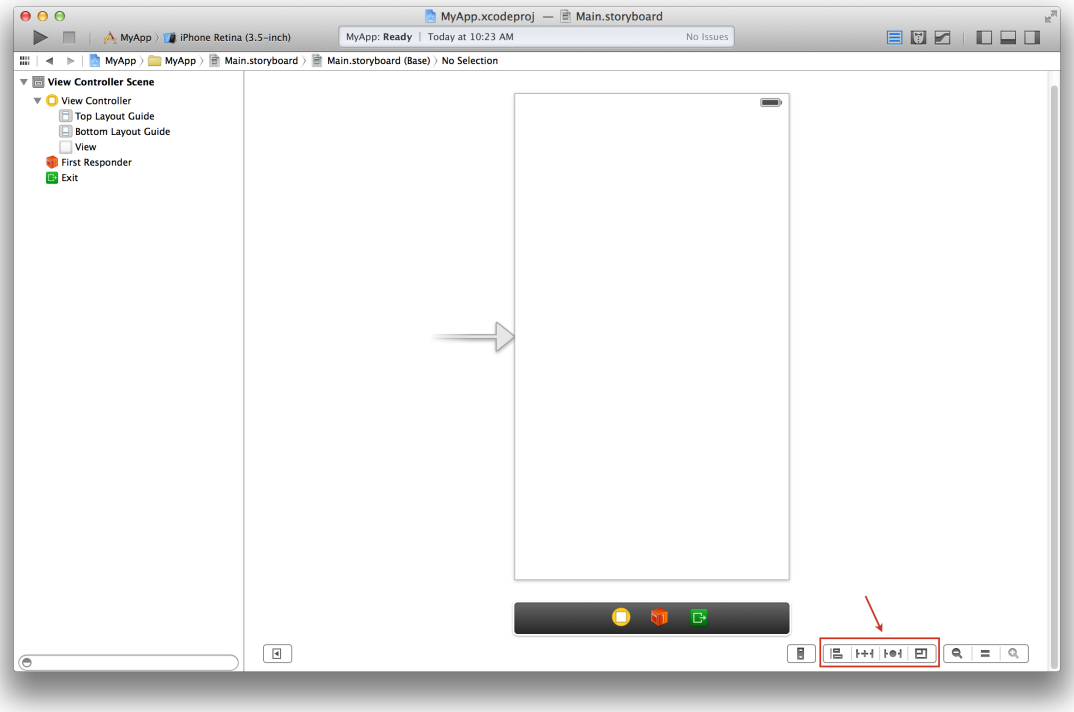
You can Control-drag from an element to itself, to its container, or to another element. Depending on what you drag to and which direction you drag in, Auto Layout limits the possibilities of constraints appropriately. For example, if you drag horizontally to the right from an element to its container, you have the options to pin the element's trailing space or to center it vertically in the container.



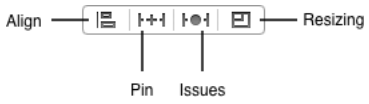
Tip: To select multiple constraints at a time from the Control-drag menu, hold down the Command or Shift key.

Adding Constraints with Align and Pin Menus

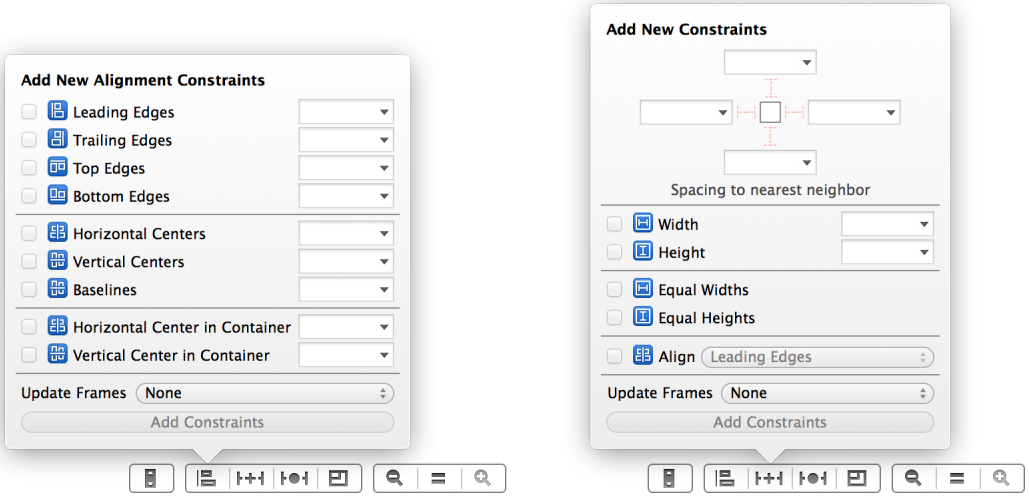
You can also add constraints using the **Auto Layout menu**, which resides on the Interface Builder canvas.



In addition to adding constraints for alignment or spacing, you can also use this menu to resolve layout issues, and determine constraint resizing behavior.



- **Align.** Create alignment constraints, such as centering a view in its container, or aligning the left edges of two views.
- **Pin.** Create spacing constraints, such as defining the height of a view, or specifying its horizontal distance from another view.
- **Issues.** Resolve layout issues by adding or resetting constraints based on suggestions (see Resolving Auto Layout Issues (page 17)).
- **Resizing.** Specify how resizing affects constraints .



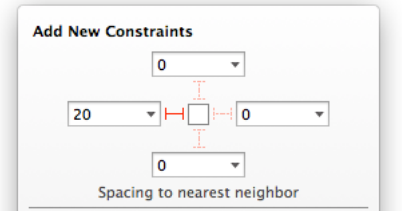
Constraint options that require multiple elements are disabled if you have only a single element selected.

To add a constraint from the Align or Pin menu

1. Select the checkbox next to the appropriate constraint.

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

To select a "Spacing to nearest neighbor" constraint, select the red constraint corresponding to the appropriate side of the element.



If you need to create a constraint related to another view that is not the nearest neighbor, click the black disclosure triangle in the value text field to display a drop-down menu of other nearby views.

2. Enter a corresponding constant value.
3. Press a button to create the constraints.
   - The Add Constraints button adds the new constraints to the selected elements.
   - The Add and Update Frames button adds the new constraints to the selected elements, and moves the elements in your interface to satisfy every constraint as well as possible.

Note: You're adding brand new constraints every time you click one of the two buttons. You are not editing existing constraints. For information about editing existing constraints, see Editing Constraints (page 13).

Adding Missing or Suggested Constraints

Use the **Issues menu** to add constraints if you need a starting point for your layout, or if you need to make a lot of changes quickly.

If you need to add a large set of constraints to describe your interface layout and you don't want to add constraints one at a time, choose Issues > Add Missing Constraints to add a nonambiguous set of constraints. This command infers constraints based on where things are laid out.

If you need to revert to a set of constraints without errors, or you just want to start over, choose Issues > Reset to Suggested Constraints to remove erroneous constraints and add a nonambiguous set of constraints. This is equivalent to Clear Constraints followed by Add Missing Constraints.

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Editing Constraints

You can change the constant, relation, and priority of a constraint. You can edit these properties either by double-clicking the constraint on the canvas and editing the value, or by selecting the constraint and using the Attributes inspector. You cannot, however, change the type of a constraint (for example, you can't change a width constraint into a height constraint).

Deleting Constraints

Delete any constraint at any time by selecting it on the canvas or in the outline view and pressing the Delete key.

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Working with Auto Layout Programmatically

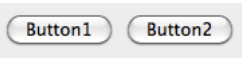
Even though Interface Builder provides a convenient visual interface for working with Auto Layout, you can also create, add, remove, and adjust constraints through code. If you add or remove views at runtime, for example, you'll need to add constraints programmatically to ensure that your interface responds correctly to changes in size or orientation.

Creating Constraints Programmatically

You represent constraints using instances of `NSLayoutConstraint`. To create constraints, you typically use `constraintsWithVisualFormat:options:metrics:views:`.

The first argument to this method, a **visual format string**, provides a visual representation of the layout you want to describe. This **visual format language** is designed to be as readable as possible; a view is represented in square brackets, and a connection between views is represented using a hyphen (or two hyphens separated by a number to represent the number of points apart the views should be). For more examples and to learn the grammar for the visual format language, see Visual Format Language (page 41).

As an example, you might represent the constraint between two buttons:



using the following visual format string:

```
[button1]-12-[button2]
```

A single hyphen denotes the standard Aqua space, so you can also represent the relationship like this:

```
[button1]-[button2]
```

The names of the views come from the `views` dictionary—the keys are the names you use in the format string, and the values are the corresponding view objects. As a convenience, `NSDictionaryOfVariableBindings` creates a dictionary where the keys are the same as the corresponding value's variable name. The code to create the constraints becomes:

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

```
NSDictionary *viewsDictionary =
                NSDictionaryOfVariableBindings(self.button1, self.button2);
NSArray *constraints =
        [NSLayoutConstraint constraintsWithVisualFormat:@"[button1]-[button2]"
                                options:0 metrics:nil views:viewsDictionary];
```

The visual format language prefers good visualization over completeness of expressibility. Although most of the constraints that are useful in real user interfaces can be expressed using the language, some cannot. One useful constraint that cannot be expressed is a fixed aspect ratio (for example, `imageView.width = 2 * imageView.height`). To create such a constraint, you can use `constraintWithItem:attribute:relatedBy:toItem:attribute:multiplier:constant:`.

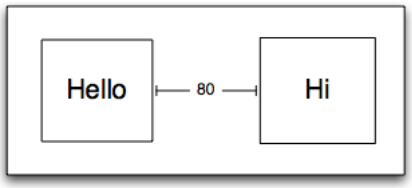
As an example, you could also use this method to create the earlier `[button1]-[button2]` constraint:

```
[NSLayoutConstraint constraintWithItem:self.button1 attribute:NSLayoutAttributeRight
                    relatedBy:NSLayoutRelationEqual toItem:self.button2
                  attribute:NSLayoutAttributeLeft multiplier:1.0 constant:-12.0];
```

Installing Constraints

To make a constraint active, you must add it to a view. The view that holds the constraint must be an ancestor of the views the constraint involves, and should usually be the closest common ancestor. (This is in the existing `NSView` API sense of the word *ancestor*, where a view is an ancestor of itself.) The constraint is interpreted in the coordinate system of that view.

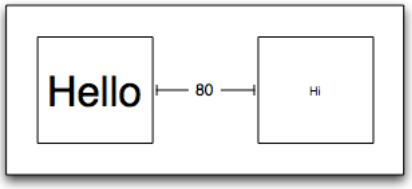
Suppose you install `[zoomableView1]-80-[zoomableView2]` on the common ancestor of `zoomableView1` and `zoomableView2`.



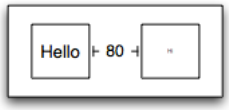
The value 80 is in the coordinate system of the container, which is what it looks like if you draw the constraint.

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

If the bounds transform of either of the zoomable views changes, the space between them remains fixed.



If the bounds transform in the container itself is changed, the space scales, too.



`NSView` provides a method—`addConstraint:`—to add a constraint, and methods to remove or inspect existing constraints—`removeConstraint:` and `constraints`—as well as other related methods. `NSView` also provides a method, `fittingSize`, which is similar to the `sizeToFit` method of `NSControl` but for arbitrary views rather than for controls.

The `fittingSize` method returns the ideal size for the view *considering only those constraints installed on the receiver's subtree together with a preference for the view to be as small as possible*. The fitting size is not the "best" size for a view in a general sense—in the constraint-based system, a view's "best" size (if you consider everything) is by definition its current size.

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

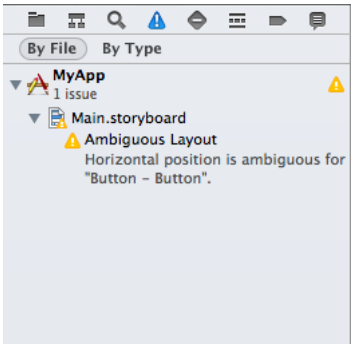
Resolving Auto Layout Issues

Auto Layout issues occur when you create conflicting constraints, when you don't provide enough constraints, or when the final layout contains a set of constraints that are ambiguous. When an issue occurs, the Auto Layout system will try to degrade as gracefully as possible so that your app remains usable, but it's important to understand how to catch layout problems in development. Auto Layout offers several features to help you find and fix the source of errors, including visual hints in Interface Builder that identify which issue is occurring so that you can resolve it with the appropriate solution.

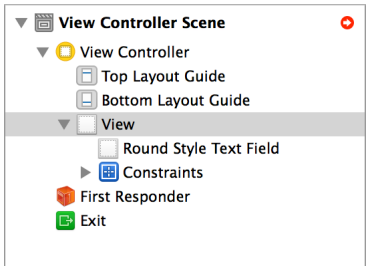
Identifying Issues

Interface Builder displays Auto Layout issues in a number of different places:

In the Issues Navigator. The Navigator groups issues by type.

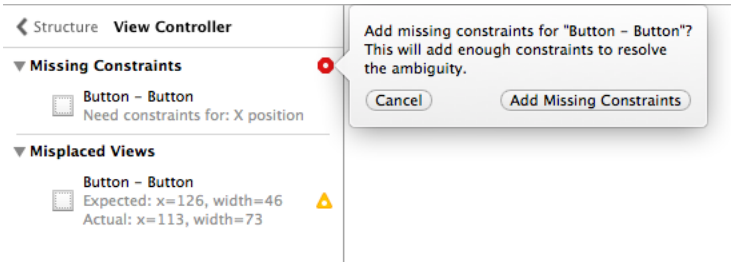


In the Interface Builder outline view. If there are any issues in the top-level view that you're editing on the canvas, the Interface Builder outline view shows a disclosure arrow. This arrow is red if there are conflicts or ambiguities, and yellow if constraints result in view misplacement.



2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Click the disclosure arrow, and you'll see a list of the issues separated by type, with information about the relevant controls and constraints. Click the error or warning symbol next to any issue to see a description of what's happening and a recommended course of action.

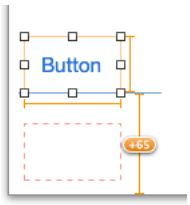


On the canvas. Misplaced or ambiguous constraints are shown in orange, conflicting constraints are red, and red dotted frames show the predicted runtime positions of misplaced or ambiguous views; these issues are described in the following sections.

Resolving View Misplacement

In Interface Builder, constraints and frames are separate. If there is a mismatch for a view between the frame you're seeing on the canvas and the runtime position based on the current set of constraints, you have **view misplacement**.

Interface Builder shows misplaced views by drawing constraints that are no longer satisfied in orange, with a misplacement delta. It also displays a dotted red frame to show you where the view will be at runtime.



To resolve misplacement

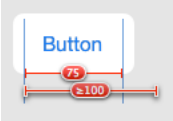
Do one of the following:

- Choose Issues > Update Frames. This causes the element to move back to where it was before it was misplaced. The frames change to satisfy existing constraints.
- Choose Issues > Update Constraints. This causes the constraints to be updated to the new location of the element. The constraints change to match the current frames.

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Resolving Constraint Conflicts

Conflict occurs when you have a set of constraints that Auto Layout can't satisfy, such as two different widths for an element. Conflicting constraints show up on the canvas in red, drawn at actual size.



To resolve a constraint conflict

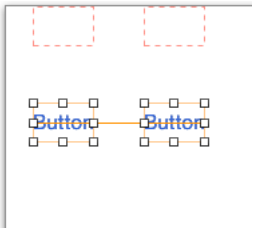
- Delete one of the conflicting constraints.

Tip: If you want to remove *all* constraints and start over, choose Issues > Clear Constraints.

Resolving Ambiguity

Ambiguity occurs when the constraints you've defined result in multiple possible solutions for views' size and placement; this may be because, for example, there are not enough constraints, or because a view's content size is undefined.

Interface Builder shows ambiguity by drawing frames with orange borders; to learn more about the ambiguity, use the Issues navigator. The way you resolve ambiguity depends on the condition that's causing it.



One or More Constraints Are Missing

In the simplest case, there are just not enough constraints—for example, you constrained the horizontal position of an element, but didn't add constraints for the vertical position. To resolve the ambiguity, you need to add the missing constraints

To add missing constraints

- Choose Issues > Add Missing Constraints.

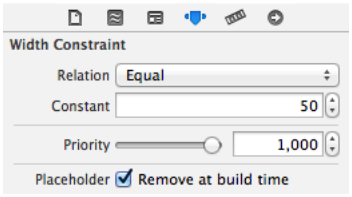
2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Content Size Is Undefined

Some container views—such as a stack view—depend on the size of their content to determine their own size at run time, which means the size is unknown at design time. To address this problem, you should set placeholder constraints, such as a placeholder minimum width for a view (this placeholder is removed at build time, when your view defines its size).

To create a placeholder constraint

1. Add the desired constraint as normal.
2. Select the constraint, then open the Attributes inspector.
3. Select the checkbox next to the Placeholder option.



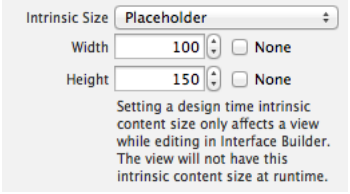
The Size of Custom View Content Is Unknown

Unlike standard views, custom views have no defined intrinsic content size. During design time, Interface Builder doesn't know what size to expect a custom view to be. To address this problem, you should set a **placeholder intrinsic content size** to indicate the custom view's content size.

To set a view's placeholder intrinsic content size

1. With the view selected, open the Size inspector.

2. Under the Intrinsic Size option, select Placeholder.
3. Set appropriate values for the width and height.



2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Debugging in Code

Broadly speaking, there are two phases to debugging a layout problem:

1. Map from "this view is in the wrong place" to "this constraint (or these constraints) is (are) incorrect."
2. Map from "this constraint is incorrect" to "this code is incorrect."

To debug a problem with Auto Layout

1. Identify the view with an incorrect frame.

   It may be obvious which view has the problem; if it is not, you may find it helpful to use the `NSView` method `_subtreeDescription` to create a textual description of the view hierarchy.

   > **Important:** The `_subtreeDescription` method is not public API; it is, however, permissible to use for debugging purposes.

2. If possible, reproduce the issue while running under the Auto Layout template in Instruments.
3. Find the bad constraint or constraints.

   To get the constraints affecting a particular view, use `constraintsAffectingLayoutForOrientation:` in OS X or `constraintsAffectingLayoutForAxis:` in iOS.

   You can then inspect the constraints in the debugger, which prints constraints using the visual format notation described in Visual Format Language (page 41). If your views have identifiers, they are printed out in the description, like this:

   ```
   <NSLayoutConstraint: 0x400bef8e0
   H:[refreshSegmentedControl]-(8)-[selectViewSegmentedControl] (Names:
   refreshSegmentedControl:0x40048a600,
   selectViewSegmentedControl:0x400487cc0 ) >
   ```

   otherwise the output looks like this:

   ```
   <NSLayoutConstraint: 0x400cbf1c0
   H:[NSSegmentedControl:0x40048a600]-(8)-[NSSegmentedControl:0x400487cc0]>
   ```

4. If it's not obvious which constraint is wrong at this point, visualize the constraints on screen by passing the constraints to the window using `visualizeConstraints:`.

   When you click a constraint, it is printed in the console. In this way you can you can determine which is which, and typically identify which is incorrect.

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

At this point you may be informed that the layout is ambiguous.

5. Find the code that's wrong.

   Sometimes, once you have identified the incorrect constraint, you will know what to do to fix it.

   If this is not the case, then use Instruments to search for the pointer of the constraint (or some of its description). This will show you interesting events in that constraint's lifecycle—its creation, modification, installation into windows, and so on. For each of these you can see the backtrace where it happened. Find the stage at which things went awry, and look at the backtrace. This is the code with the problem.

Auto Layout Degrades Gracefully with Unsatisfiable Constraints

It is a programming error to configure constraints that cannot be satisfied. Faced with unsatisfiable constraints, however, the Auto Layout system attempts to degrade gracefully.

1. The `addConstraint:` method (or the `setConstant:` method of `NSLayoutConstraint`) logs the mutually unsatisfiable constraints and (because this is a programmer error) throws an exception.
2. The system catches the exception immediately.

   Although adding a constraint that cannot be satisfied is a programmer error, it's an error that one can readily imagine occurring on a user's computer, and from which the system can recover more gracefully than crashing the program. The exception is thrown so that you notice the problem, and caught because it's easier to debug the constraint system if the system does still function.
3. To allow layout to proceed, the system selects a constraint from among the mutually unsatisfiable set and lowers its priority from "required" to an internal value that is not required, but that is higher priority than anything you can externally specify. The effect is that as things change going forward, this incorrectly broken constraint is the first that the system attempts to satisfy. (Note that every constraint in the set is required to begin with, because otherwise it wouldn't be causing a problem.)

```
2010-08-30 03:48:18.589 ak_runner[22206:110b] Unable to simultaneously satisfy
constraints:

(

   "<NSLayoutConstraint: 0x40082d820 H:[NSButton:0x4004ea720'OK']-(20)-|   (Names:
 '|':NSView:0x4004ea9a0 ) >",

   "<NSLayoutConstraint: 0x400821b40 H:[NSButton:0x4004ea720'OK']-(29)-|   (Names:
 '|':NSView:0x4004ea9a0 ) >"

)
```

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

```
Will attempt to recover by breaking constraint

<NSLayoutConstraint: 0x40082d820 H:[NSButton:0x4004ea720'OK']-(20)-|   (Names:
'|':NSView:0x4004ea9a0 ) >
```

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Auto Layout by Example

Auto Layout makes it easy to solve many complex layout problems automatically, without the need for manual view manipulation. By creating the right combination of constraints, you can create layouts that are traditionally difficult to manage in code, such as equally spaced views that adjust to changes in orientation or size, elements inside scroll views that affect the size of the scrolling content, or elements inside scroll views that don't scroll with the rest of the contents.

Using Scroll Views with Auto Layout

When you are creating an app using Auto Layout, scroll views can present a unique challenge. The size of the scrolling content must be set correctly so that the user can scroll through all of the available content, for example, and if you need to lock a contextual view in place on top of a scroll view, such as the scale and legend for a map, it's difficult to ensure that the element doesn't scroll with the rest of the content.

Controlling Scroll View Content Size

The size of the content inside of a scroll view is determined by the constraints of its descendants.

To set the size of a scroll view

1. Create the scroll view.
2. Place the UI element inside it.
3. Create constraints that fully define the width and height of the scroll view content.

You must make sure you create constraints for all the subviews inside a scroll view. For example, when defining the constraints for a view that doesn't have an intrinsic content size, you'll need more than just a leading edge constraint—you must also create trailing edge, width, and height constraints. There cannot be any missing constraints, starting from one edge of the scroll view to the other.

Creating Anchored Views Inside a Scroll View

You may find you want to create an area inside a scroll view that doesn't move when a user scrolls the contents of the scroll view. You accomplish this by using a separate **container view**.

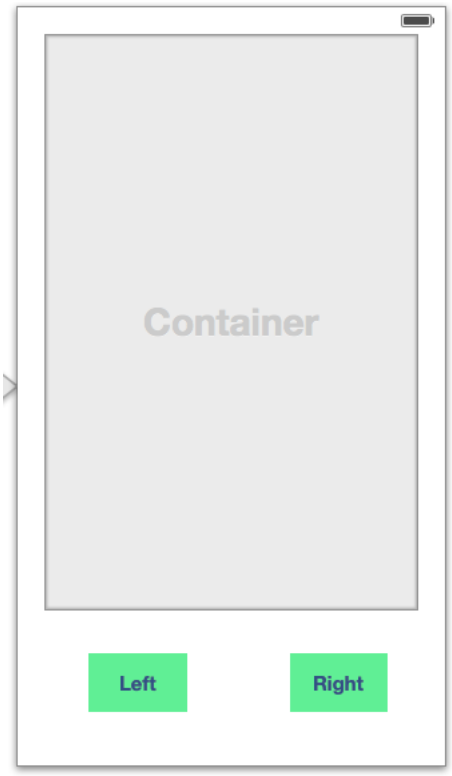
2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

To lock a view inside a scroll view

1. Create a container view to hold the scroll view.
2. Create the scroll view and place it in the container view with all edges equal to zero points.
3. Create and place a subview inside of the scroll view.
4. Create constraints from the subview to the container view.

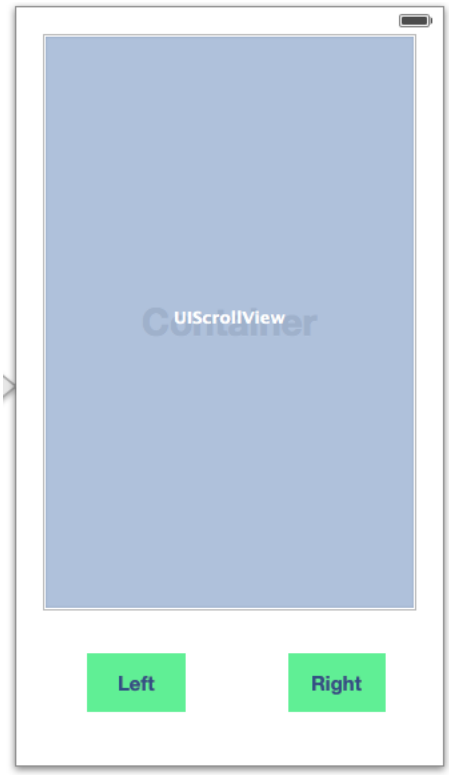
The following example uses the steps in the above task to show how to position a text view inside of a scroll view. In this example, the text view stays at the bottom of the scroll view and doesn't move when the scroll view contents are moved.

First, create the container view that will contain the scroll view. Set the size of the container view equal to the desired size of the scroll view.



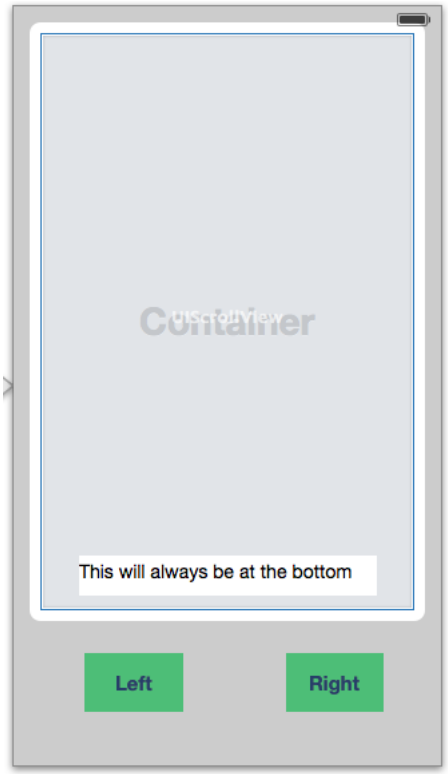
2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

After the container view is created, create a scroll view and place it inside of the container view. Resize the scroll view so that all of the edges are flush with the container view's edges, by setting the distance to 0.



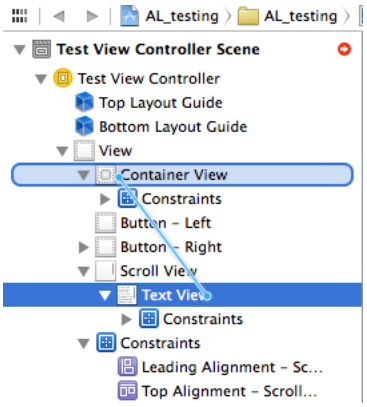
2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Create another view and place it inside of the scroll view. In this example, a text view is placed inside of the scroll view.



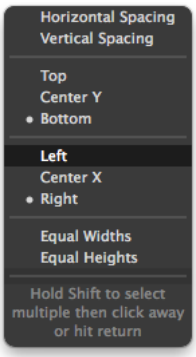
After placing the text view, create constraints from the text view to the container view. Creating constraints that anchor the text view to the container view (skipping the scroll view) anchors the text view relative to the container view, which ensures that the scroll view won't scroll the text view.

To create a constraint that crosses multiple views in the view hierarchy, it is generally easier to Control-drag from the view to the container view in the Interface Builder outline view.

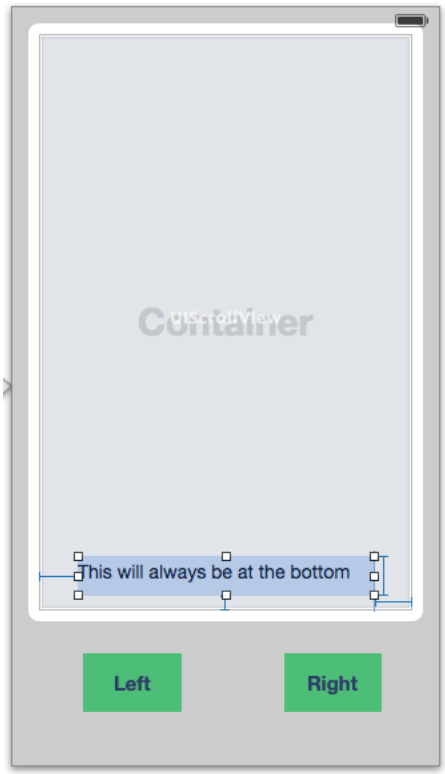


Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

In the constraint overlay that appears, set the required constraints for the view.

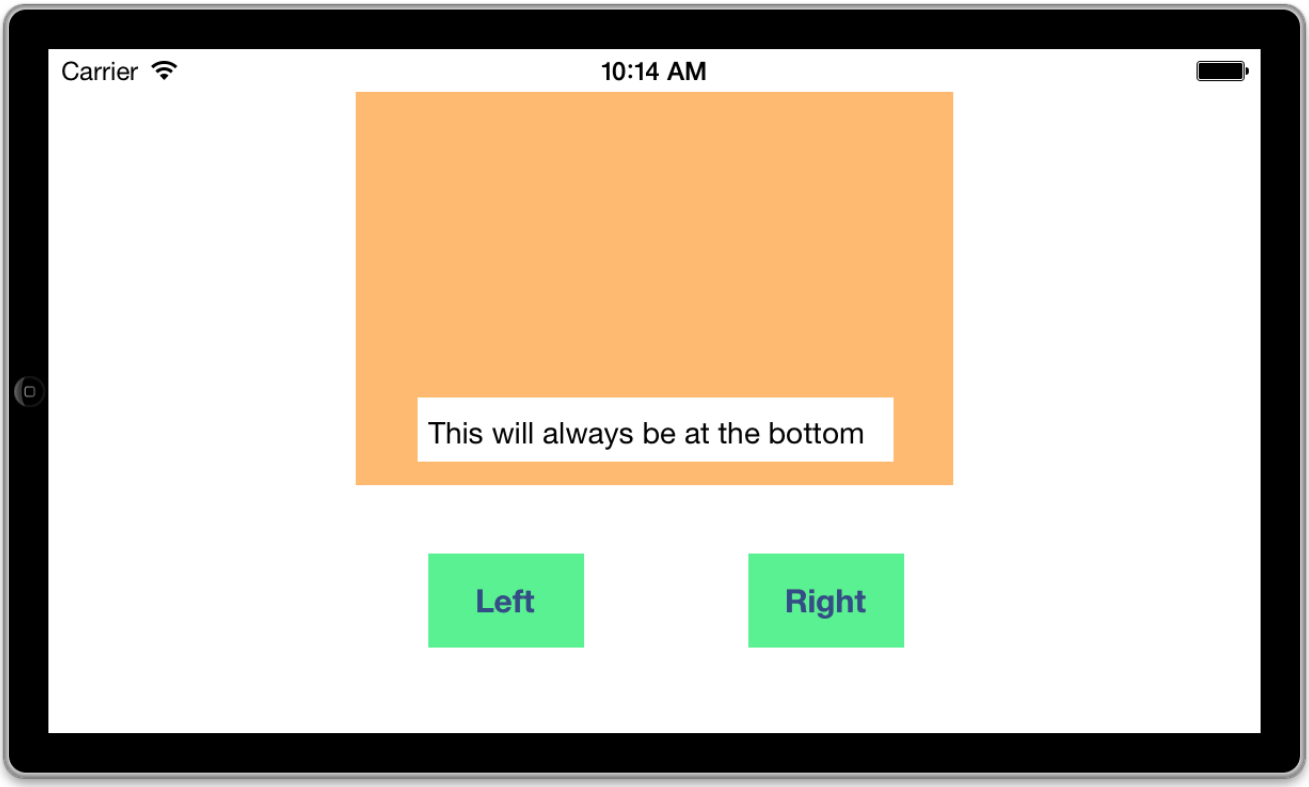


In this example, constraints are created from the leading, trailing, and bottom edges of the text view to the container view. The height of the text view is also constrained.

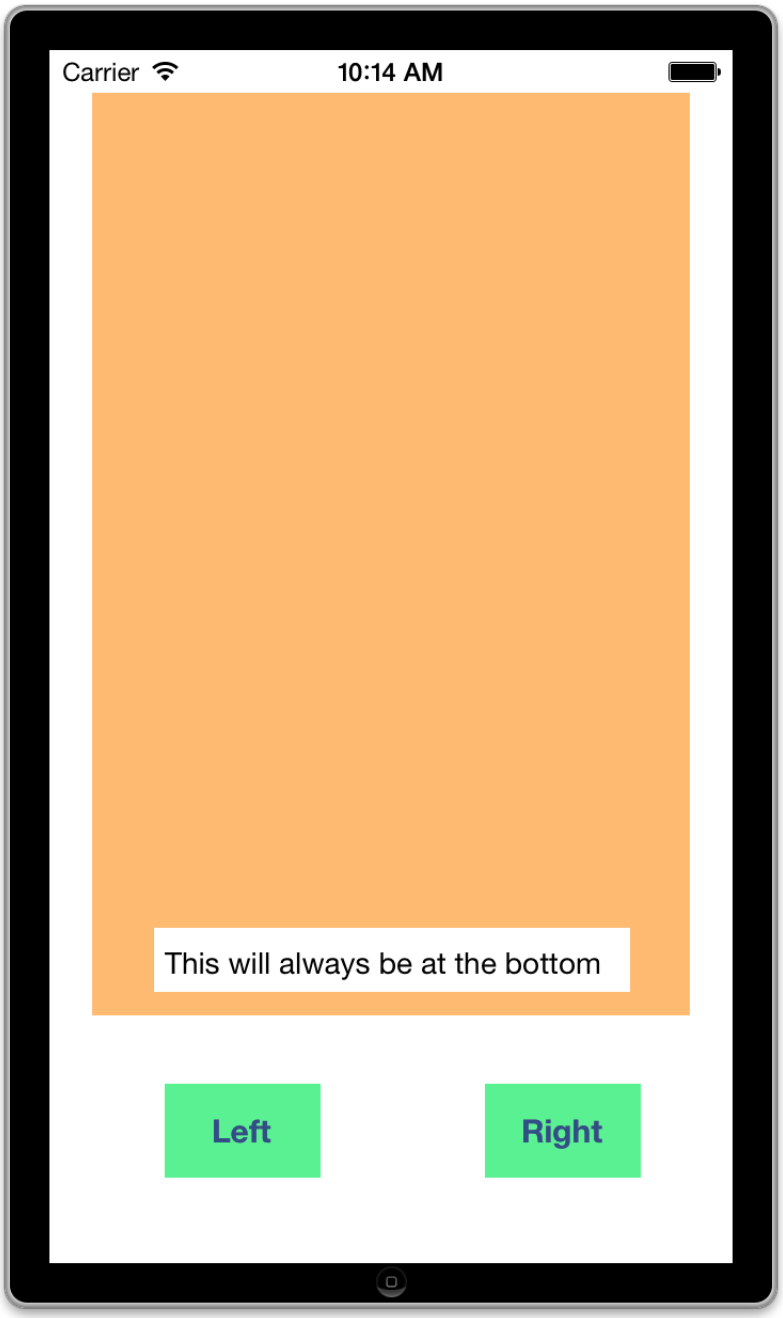


2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

The following two figures show the app in iOS Simulator, both in normal and landscape positions. The text view is constrained at the bottom of the scroll view and doesn't move as the scroll view is moved.



2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.



Spacing and Wrapping

Auto Layout provides several techniques for automatically spacing views and resizing items based on their content. The following sections describe how to create constraints that keep visible views proportionally spaced based on the orientation of the device.

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Creating Equal Spacing Between Views

To lay out several views that are proportionally spaced based on the orientation of a device, create spacer views between the visible views. Set the constraints of these spacer views correctly to ensure that the visible views are able to stay spaced apart based on the orientation of the device.

To space views proportionally

1. Create the visible views.
2. Create the spacer views equal to the number of visible views plus one.
3. Alternate placing your views, starting with a spacer view.

   To space two visible views, place all of the views in the following pattern, starting from the left side of the screen and moving right:

   `spacer1` | `view1` | `spacer2` | `view2` | `spacer3`.

4. Constrain the spacer views so that their lengths are equal to each other.

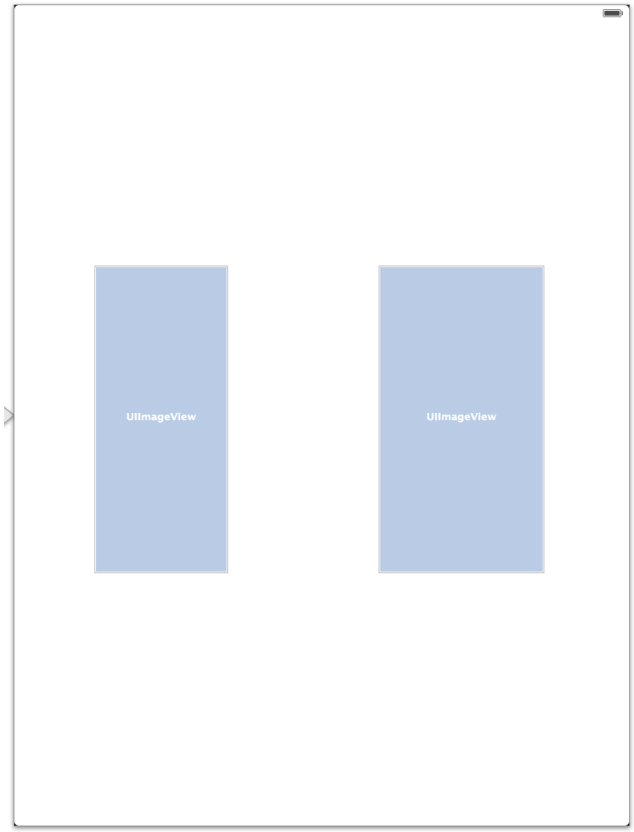
   Note: The height of the spacer views can be any value, including `0`. However, you must create constraints for the height of the views—don't leave the height ambiguous.

5. Create a leading constraint from the first spacer view to the container view.
6. Create a trailing constraint from the last spacer view to the container view.
7. Create constraints between the spacer views and the visible views.

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

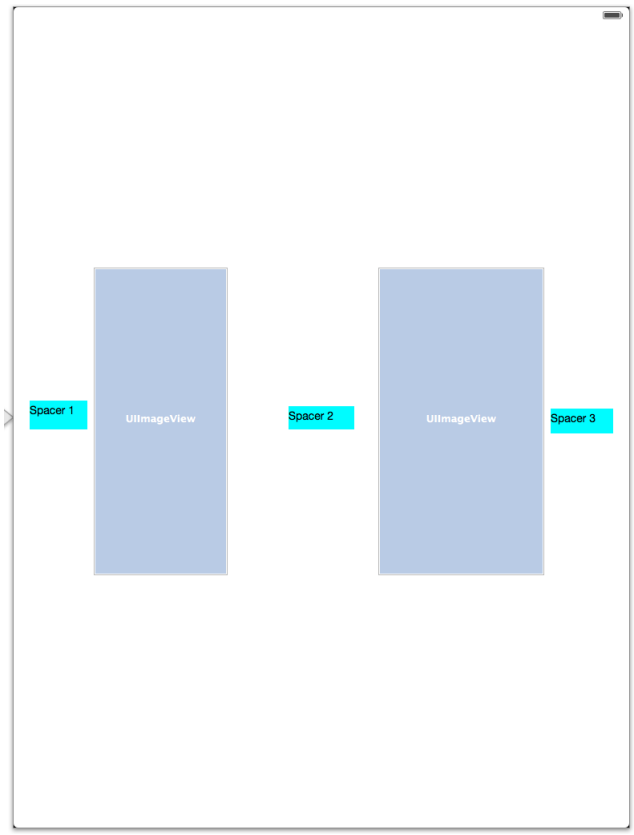
> **Note:** When spacing views vertically, start from the top of the screen and place each view below the previous view. Set the heights of the spacer views equal to each other.

The following example uses the steps in the above task to show how to position two views proportionally spaced. The spacer views are annotated for the example, but are normally left empty with no background. First, create the two views and place them in the storyboard.



2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Add the three spacer views—one to the left of the leftmost view, one between the two views, and one to the right of the rightmost view. The spacer views don't have to be the same size at this time because their size will be set through constraints.

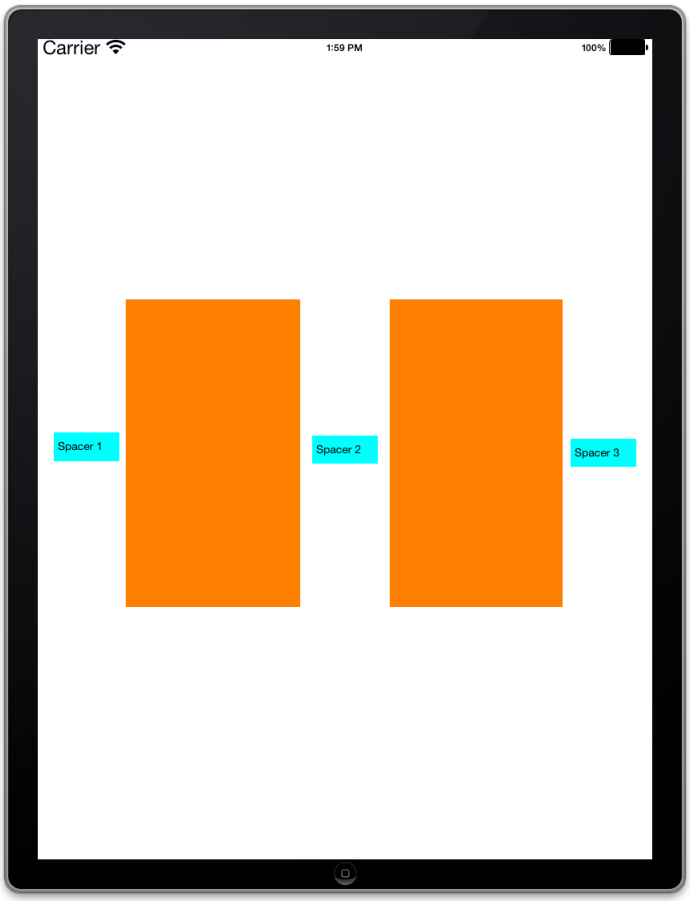


Create the following constraints for the spacer views:

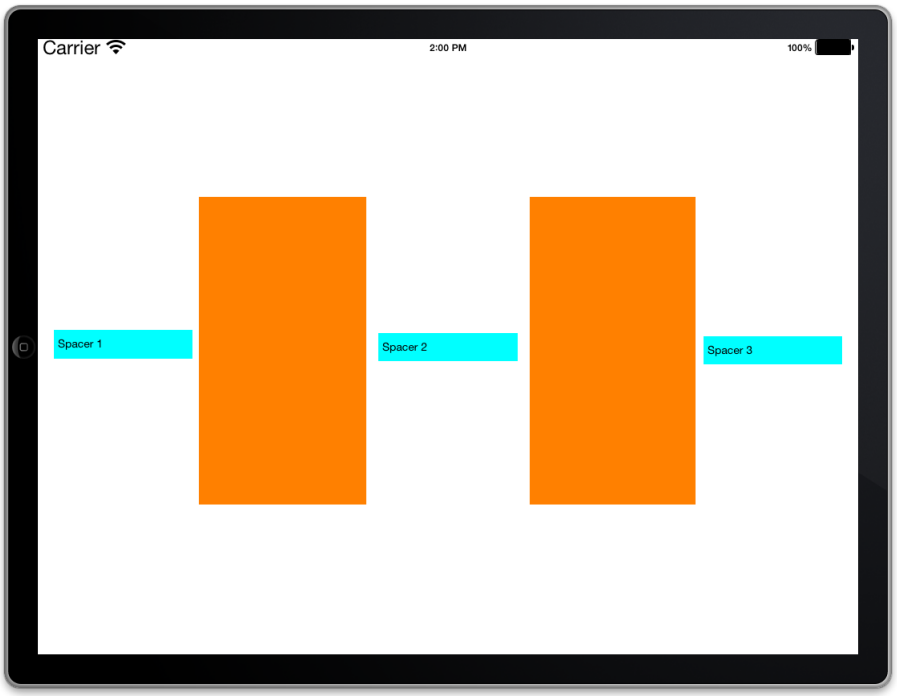
- Constrain the width of spacer view 2 and spacer view 3 to be equal to the width of spacer view 1.
- Constrain the width of spacer view 1 to be greater than or equal to the minimum desired width.
- Create a Leading Space to Container constraint from spacer view 1 to the container.
- Create a Horizontal Spacing constraint from spacer view 1 to view 1. Set this constraint to be a less-than-or-equal-to constraint with a priority of 1000.
- Create Horizontal Spacing constraints from spacer view 2 to view 1 and view 2. Set these constraints to be a less-than-or-equal-to constraint with a priority of 999.
- Create a Horizontal Spacing constraint from spacer view 3 to view 2. Set this constraint to be a less-than-or-equal-to constraint with a priority of 1000.
- Create a Trailing Space to Container constraint from spacer view 3 to the container.

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

These constraints create two visible views and three invisible views (spacer views). These spacer views automatically resize as the orientation of the device changes, keeping the visible views proportionally spaced, as shown in the following two figures:



2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.



Animating Changes Made by Auto Layout

If you need full control over animating changes made by Auto Layout, you must make your constraint changes programmatically. The basic concept is the same for both iOS and OS X, but there are a few minor differences.

In an iOS app, your code would look something like the following:

```
[containerView layoutIfNeeded]; // Ensures that all pending layout operations have
 been completed

[UIView animateWithDuration:1.0 animations:^{

     // Make all constraint changes here

     [containerView layoutIfNeeded]; // Forces the layout of the subtree animation
 block and then captures all of the frame changes

}];
```

In OS X, use the following code when using layer-backed animations:

```
[containterView layoutIfNeeded];

NSAnimationContext runAnimationGroup:^(NSAnimationContext *context) {

     [context setAllowsImplicitAnimation: YES];

     // Make all constraint changes here
```

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

```
    [containerView layoutIfNeeded];
}];
```

When you aren't using layer-backed animations, you must animate the constant using the constraint's animator:

```
[[constraint animator] setConstant:42];
```

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Implementing a Custom View to Work with Auto Layout

Auto Layout reduces the burden on controller classes by making views more self-organizing. If you implement a custom view class, you must provide enough information so that the Auto Layout system can make the correct calculations and satisfy the constraints. In particular, you should determine whether the view has an intrinsic size, and if so, implement `intrinsicContentSize` to return a suitable value.

A View Specifies Its Intrinsic Content Size

Leaf-level views, such as buttons, typically know more about what size they should be than does the code that is positioning them. This is communicated through the method `intrinsicContentSize`, which tells the layout system that there is some content it doesn't natively understand in a view, and which provides to the layout system the intrinsic size of that content.

A typical example is a single-line text field. The layout system does not understand text—it must just be told that there's something in the view, and that that something will require a certain amount of space in order not to clip the content. The layout system calls `intrinsicContentSize`, and sets up constraints that specify (1) that the opaque content should not be compressed or clipped and (2) that the view prefers to hug tightly to its content.

A view can implement `intrinsicContentSize` to return absolute values for its width and height, or to return `NSViewNoInstrinsicMetric` for either or both to indicate that it has no intrinsic metric for a given dimension.

For further examples, consider the following implementations of `intrinsicContentSize`:

- The intrinsic size for a button is dictated by its title and image. A button's `intrinsicContentSize` method should return a size with a width and height large enough to ensure that the title text is not clipped and that its entire image is visible.
- A horizontal slider has an intrinsic height but no intrinsic width because the slider artwork has no intrinsic "best width." A horizontal `NSSlider` object returns a `CGSize` that is set to `{NSViewNoInstrinsicMetric, <slider height>}`. Any user of a slider will have to provide constraints that govern the width of the slider.
- A container, such as an instance of `NSBox`, has *no* intrinsic content size and returns `{ NSViewNoInstrinsicMetric, NSViewNoInstrinsicMetric }`. Its subviews may have intrinsic content sizes, but the subviews' content is not intrinsic to the box itself.

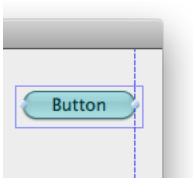
2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Views Must Notify Auto Layout If Their Intrinsic Size Changes

If any property of your view changes, and that change affects the intrinsic content size, your view must call `invalidateIntrinsicContentSize` so that the layout system notices the change and can recalculate the layout. For example, a text field calls `invalidateIntrinsicContentSize` if the string value changes.

Layout Operates on Alignment Rectangles, Not on Frames

When considering layout, keep in mind that the frame of a control is less important than its visual extent. As a result, ornaments such as shadows and engraving lines should typically be ignored for the purposes of layout. Interface Builder ignores them when positioning views on the canvas—in the following example, the Aqua guide (the dashed blue line) aligns with the visual extent of the button, not with the button's frame (the solid blue rectangle).



To allow layout based on the presentation of the content rather than the frame, views provide an **alignment rectangle**, which is what the layout actually operates on. To determine whether your override is correct on OS X, you can set the `NSViewShowAlignmentRects` default to `YES` to draw the alignment rects.

Views Indicate Baseline Offsets

Frequently, you want to align controls by baseline. Although you have always been able to align against baselines in Interface Builder, it was not previously possible to do this programatically. You can do so now using `NSLayoutAttributeBaseline` in a constraint.

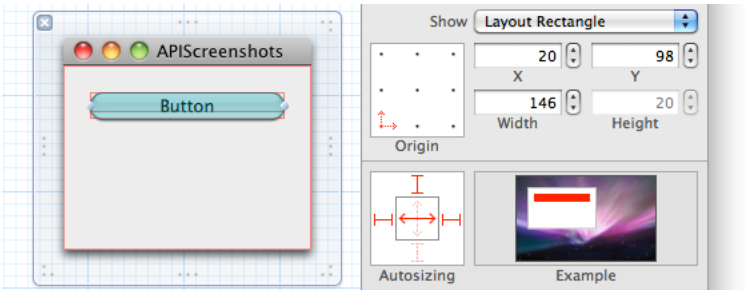
The method `baselineOffsetFromBottom` returns the distance between `NSLayoutAttributeBottom` and `NSLayoutAttributeBaseline`. This default return value for `NSView` is `0`; subclasses override this method when it makes sense to do so.

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Adopting Auto Layout

Views that are aware of Auto Layout can coexist in a window with views that are not. That is, an existing project can incrementally adopt Auto Layout—you don't have to make it work in your entire app all at once. Instead, you can transition your app to use Auto Layout one view at a time using the property `translatesAutoresizingMaskIntoConstraints`.

When this property is `YES`, which it is by default, the **autoresizing mask** of a view is translated into constraints. For example, if a view is configured like the one below, and `translatesAutoresizingMaskIntoConstraints` is `YES`, then the constraints `|-20-[button]-20-|` and `V:|-20-[button(20)]` are added to the view's superview. The net effect is that unaware views behave as they did in versions of OS X prior to version 10.7.



If you move the button `15` points to the left (including by calling `setFrame:` at runtime), the new constraints would be `|-5-[button]-35-|` and `V:|-20-[button(20)]`.

For views that are aware of Auto Layout, in most circumstances you want `translatesAutoresizingMaskIntoConstraints` to be `NO`. The reason is that the constraints generated by translating the autoresizing mask are already sufficient to completely specify the frame of a view given its superview's frame, which is generally too much. For example, this translation would prevent a button from automatically assuming its optimal width when its title is changed.

The principal circumstance in which you should *not* call `setTranslatesAutoresizingMaskIntoConstraints:` is when you are not the person who specifies a view's relation to its container. For example, an `NSTableRowView` instance is placed by `NSTableView`. It might do this by allowing the autoresizing mask to be translated into constraints, or it might not. This is a private implementation detail. Other views on which you should not call `setTranslatesAutoresizingMaskIntoConstraints:` include an `NSTableCellView` object, a subview of `NSSplitView`, a view of `NSTabViewItem`, or the content view of an `NSPopover`, `NSWindow`, or `NSBox`

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

object. For those familiar with classic Cocoa layout: If you would not call `setAutoresizingMask:` on a view in classic style, you should not call `setTranslatesAutoresizingMaskIntoConstraints:` under Auto Layout.

If you have a view that does its own custom layout by calling `setFrame:`, your existing code should work. Just don't call `setTranslatesAutoresizingMaskIntoConstraints:` with the argument `NO` on views that you place manually.

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Visual Format Language

This appendix shows how to use the Auto Layout visual format language to specify common constraints, including standard spacing and dimensions, vertical layout, and constraints with different priorities. In addition, this appendix contains a complete language grammar.

Visual Format Syntax

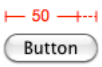
The following are examples of constraints you can specify using the visual format. Note how the text visually matches the image.

Standard Space

```
[button]-[textField]
```

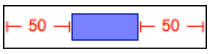
Width Constraint

```
[button(>=50)]
```



Connection to Superview

```
|-50-[purpleBox]-50-|
```



Vertical Layout

```
V:[topField]-10-[bottomField]
```

Flush Views

```
[maroonView][blueView]
```

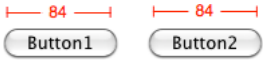
2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Priority

```
[button(100@20)]
```

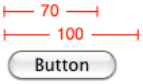
Equal Widths

```
[button1(==button2)]
```



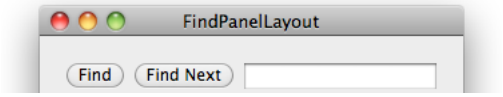
Multiple Predicates

```
[flexibleButton(>=70,<=100)]
```



A Complete Line of Layout

```
|-[find]-[findNext]-[findField(>=20)]-|
```



The notation prefers good visualization over completeness of expressibility. Most of the constraints that are useful in real user interfaces can be expressed using visual format syntax, but there are a few that cannot. One useful constraint that cannot be expressed is a fixed aspect ratio (for example, `imageView.width = 2 * imageView.height`). To create such a constraint, you must use `constraintWithItem:attribute:relatedBy:toItem:attribute:multiplier:constant:`.

Visual Format String Grammar

The visual format string grammar is defined as follows (literals are shown in `code font`; **e** denotes the empty string).

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Symbol	Replacement rule
<visualFormatString>	(<orientation>:)? (<superview><connection>)? <view>(<connection><view>)* (<connection><superview>)?
<orientation>	`H`|`V`
<superview>	`|`
<view>	`[`<viewName>(<predicateListWithParens>)?`]`
<connection>	**e**|–<predicateList>–|–
<predicateList>	<simplePredicate>|<predicateListWithParens>
<simplePredicate>	<metricName>|<positiveNumber>
<predicateListWithParens>	`(`<predicate>(`,`<predicate>)*`)`
<predicate>	(<relation>)?(<objectOfPredicate>)(`@`<priority>)?
<relation>	`==`|`<=`|`>=`
<objectOfPredicate>	<constant>|<viewName> *(see note)*
<priority>	<metricName>|<number>
<constant>	<metricName>|<number>
<viewName>	Parsed as a C identifier. This must be a key mapping to an instance of `NSView` in the passed views dictionary.
<metricName>	Parsed as a C identifier. This must be a key mapping to an instance of `NSNumber` in the passed metrics dictionary.
<number>	As parsed by `strtod_l`, with the C locale.

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

> **Note:** For the `objectOfPredicate` production, a `viewName` is only acceptable if the subject of the predicate is the width or height of a view. That is, you can use `[view1(==view2)]` to specify that `view1` and `view2` have the same width.

If you make a syntactic mistake, an exception is thrown with a diagnostic message. For example:

```
Expected ':' after 'V' to specify vertical arrangement
V|[backgroundBox]|
 ^


A predicate on a view's thickness must end with ')' and the view must end with ']'
|[whiteBox1][blackBox4(blackWidth][redBox]|
                                  ^


Unable to find view with name blackBox
|[whiteBox2][blackBox]
                     ^


Unknown relation. Must be ==, >=, or <=
V:|[blackBox4(>30)]|
               ^
```

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Document Revision History

This table describes the changes to *Auto Layout Guide*.

Date	Notes
2015-06-08	Updated to include information about using Auto Layout with iOS size classes. Added many Auto Layout examples.
2013-09-18	Updated to describe Auto Layout in Xcode 5.
2012-09-19	Added to iOS Library. Added links to WWDC videos.
2012-02-16	Corrected minor code error.
2011-07-06	New document that describes the constraint-based system for laying out user interface elements.

2015-06-08 | Copyright © 2015 Apple Inc. All Rights Reserved. Apple Confidential Information.

Apple Inc.
Copyright © 2015 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer or device for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-branded products.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, Aqua, Cocoa, Instruments, OS X, and Xcode are trademarks of Apple Inc., registered in the U.S. and other countries.

IOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT, ERROR OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

Some jurisdictions do not allow the exclusion of implied warranties or liability, so the above exclusion may not apply to you.