Programmatically Creating Constraints

On This Page

Whenever possible, use Interface Builder to set your constraints. Interface Builder provides a wide range of tools to visualize, edit, manage, and debug your constraints. By analyzing your constraints, it also reveals many common errors at design time, letting you find and fix problems before your app even runs.

Interface Builder can manage an ever-growing number of tasks. You can build almost any type of constraint directly in interface builder (see Working with Constraints in Interface Builder). You can also specify size-class-specific constraints (see Debugging Auto Layout), and with new tools like stack views, you can even dynamically add or remove views at runtime (see Dynamic Stack View). However, some dynamic changes to your view hierarchy can still be managed only in code.

You have three choices when it comes to programmatically creating constraints: You can use layout anchors, you can use the `NSLayoutConstraint` class, or you can use the Visual Format Language.

Layout Anchors

The `NSLayoutAnchor` class provides a fluent interface for creating constraints. To use this API, access the anchor properties on the items you want to constrain. For example, the view controller's top and bottom layout guides have `topAnchor`, `bottomAnchor`, and `heightAnchor` properties. Views, on the other hand, expose anchors for their edges, centers, size, and baselines.

> NOTE
>
> In iOS, views also have `layoutMarginsGuide` and `readableContentGuide` properties. These properties expose `UILayoutGuide` objects that represent the view's margins and readable content guides, respectively. These guides, in turn, expose anchors for their edges, centers, and size.
>
> Use these guides when programmatically creating constraints to the margins or to readable content guides.

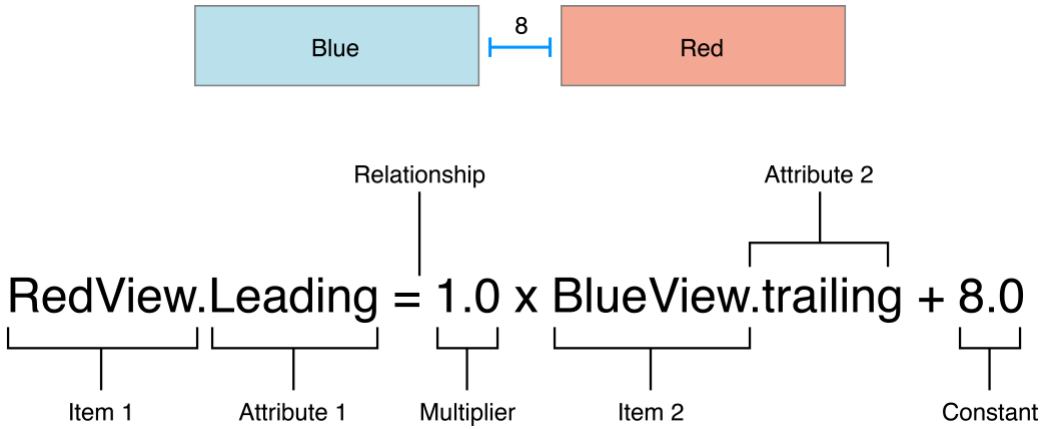
Layout anchors let you create constraints in an easy-to-read, compact format. They expose a number of methods for creating different types of constraints, as shown in Listing 13-1.

Listing 13-1 Creating layout anchors

```
1  // Get the superview's layout
2  let margins = view.layoutMarginsGuide
3
4  // Pin
5  myView.leadingAnchor.constraintEqualToAnchor(margins.leadingAnchor).active = true
6
7  // Pin the trailing edge of myView to the margin's trailing edge
8  myView.trailingAnchor.constraintEqualToAnchor(margins.trailingAnchor).active = true
9
10 // Give myView a 1:2 aspect ratio
11 myView.heightAnchor.constraintEqualToAnchor(myView.widthAnchor, multiplier: 2.0)
```

On This Page

As described in Anatomy of a Constraint, a constraint is simply a linear equation.



The layout anchors have several different methods for creating constraints. Each method includes parameters only for the elements of the equation that affect the results. So in the following line of code:

```
myView.leadingAnchor.constraintEqualToAnchor(margins.leadingAnchor).active = true
```

the symbols correspond to the following parts of the equation:

Equation	Symbol

Item 1	myView
Attribute 1	leadingAnchor
Relationship	constraintEqualToAnchor
Multiplier	None (defaults to 1.0)
Item 2	margins
Attribute 2	leadingAnchor
Constant	None (defaults to 0.0)

The layout anchors also provides additional type safety. The `NSLayoutAnchor` class has a number of subclasses that add type information and subclass-specific methods for creating constraints. This helps prevent the accidental creation of invalid constraints. For example, you can constrain horizontal anchors (`leadingAnchor` or `trailingAnchor`) only with other horizontal anchors. Similarly, you can provide multipliers only for size constraints.

> NOTE
>
> These rules are not enforced by the `NSLayoutConstraint` API. Instead, if you create an invalid constraint, that constraint throws an exception at runtime. Layout anchors, therefore, help convert runtime errors into compile time errors.

For more information, see *NSLayoutAnchor Class Reference*.

NSLayoutConstraint Class

You can also create constraints directly using the `NSLayoutConstraint` class's `constraintWithItem:attribute:relatedBy:toItem:attribute:multiplier:constant:` convenience method. This method explicitly converts the constraint equation into code. Each parameter corresponds to a part of the equation (see The constraint equation).

Unlike the approach taken by the layout anchor API, you must specify a value for each parameter, even if it doesn't affect the layout. The end result is a considerable amount of boilerplate code, which is usually harder to read. For example, the code in Listing 13-2 code is functionally identical to the code in Listing 13-1.

Listing 13-2 Directly Instantiating Constraints

```
NSLayoutConstraint(item: myView, attribute: .Leading, relatedBy: .Equal, toItem:
  view, attribute: .LeadingMargin, multiplier: 1.0, constant: 0.0).active = true

NSLayoutConstraint(item: myView, attribute: .Trailing, relatedBy: .Equal, toItem:
  view, attribute: .TrailingMargin, multiplier: 1.0, constant: 0.0).active = true

NSLayoutConstraint(item: myView, attribute: .Height, relatedBy: .Equal, toItem:
  myView, attribute:.Width, multiplier: 2.0, constant:0.0).active = true
```

On This Page

> NOTE
>
> In iOS, the `NSLayoutAttribute` enumeration contains values for the view's margins. This means that you can create constraints to the margins without going through the `layoutMarginsGuide` property. However, you still need to use the `readableContentGuide` property for constraints to the readable content guides.

Unlike the layout anchor API, the convenience method does not highlight the important features of a particular constraint. As a result, it is easier to miss important details when scanning over the code. Additionally, the compiler does not perform any static analysis of the constraint. You can freely create invalid constraints. These constraints then throw an exception at runtime. Therefore, unless you need to support iOS 8 or OS X v10.10 or earlier, consider migrating your code to the newer layout anchor API.

For more information, see *NSLayoutConstraint Class Reference*.

Visual Format Language

The Visual Format Language lets you use ASCII-art like strings to define your constraints. This provides a visually descriptive representation of the constraints. The Visual Formatting Language has the following advantages and disadvantages:

- Auto Layout prints constraints to the console using the visual format language; for this reason, the debugging messages look very similar to the code used to create the constraints.
- The visual format language lets you create multiple constraints at once, using a very compact expression.

- The visual format language lets you create only valid constraints.
- The notation emphasizes good visualization over completeness. Therefore some constraints (for example, aspect ratios) cannot be created using the visual format language.
- The comp
  testing onl.

On This Page

In Listing 13-1, the example has been rewritten using the Visual Format Language:

Listing 13-3 Creating constraints with the Visual Format Language

```
let views = ["myView" : myView]
let formatString = "|-[myView]-|"

let constraints = NSLayoutConstraint.constraintsWithVisualFormat(formatString,
  options:.AlignAllTop , metrics: nil, views: views)

NSLayoutConstraint.activateConstraints(constraints)
```

This example creates and activates both the leading and trailing constraints. The visual format language always creates 0-point constraints to the superview's margins when using the default spacing, so these constraints are identical to the earlier examples. However, Listing 13-3 cannot create the aspect ratio constraint.

If you create a more complex view with multiple items on a line, the Visual Format Language specifies both the vertical alignment and the horizontal spacing. As written, the "Align All Top" option does not affect the layout, because the example has only one view (not counting the superview).

To create constraints using the visual format language:

1. Create the `views` dictionary. This dictionary must have strings for the keys and view objects (or other items that can be constrained in Auto Layout, like layout guides) as the values. Use the keys to identify the views in the format string.

> NOTE
>
> When using Objective-C, use the `NSDictionaryOfVariableBindings` macro to create the views dictionary. In Swift, you must create the dictionary yourself.

2. (Optional) Create the `metrics` dictionary. This dictionary must have strings for the keys, and `NSNumber` objects for the values. Use the keys to represent the constant values in the format string.
3. Create the format string by laying out a single row or column of items.
4. Call the `N`
   This method returns an array containing all the constraints.
5. Activate the constraints by calling the `NSLayoutConstraint` class's `activateConstraints:` method.

For more information, see the Visual Format Language appendix.

On This Page