Autolayout

Debugging and Size classes

Nadezhda Zenkova

Debugging

Types of Errors

Errors in Auto Layout can be divided into three main categories:

- Unsatisfiable Layouts. Your layout has no valid solution.
- Ambiguous Layouts. Your layout has two or more possible solutions.
- Logical Errors. There is a bug in your layout logic.

Unsatisfiable Layouts

Unsatisfiable Layouts

Unsatisfiable layouts occur when the system cannot find a valid solution for the current set of constraints.

Two or more required constraints conflict, because they cannot all be true at the same time.

Unsatisfiable Constraints

When the layout engine is unable to find a working layout that fits all your constraints it does two things:

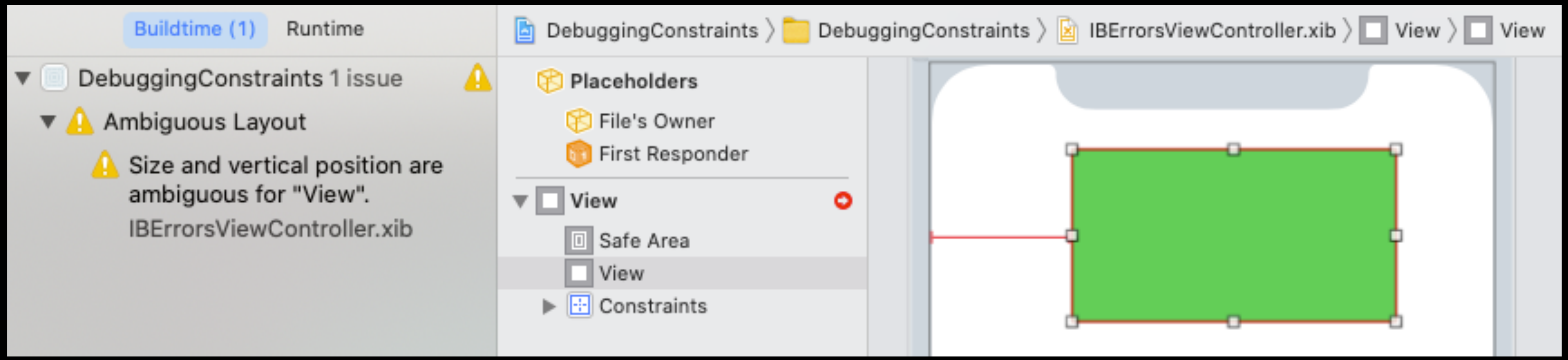
- Starts to break conflicting constraints until it can find a working solution.
- Logs the conflicting and broken constraints to the console.

Your app doesn’t crash, but views may be wrongly sized or positioned or even be offscreen.

The constraint the layout engine breaks is not necessarily the cause of the problem.

Identifying Unsatisfiable Constraints

Interface Builder



DebuggingConstraints 1 issue
Ambiguous Layout
Size and vertical position are ambiguous for "View".
IBErrorsViewController.xib

Often, Interface Builder can detect conflicts at design time. On these occasions, Interface Builder displays the error in a number of ways:

- All the conflicting constraints are drawn on the canvas in red.
- Xcode lists the conflicting constraints as warnings in the issue navigator.
- Interface Builder displays a red disclosure arrow in the upper right corner of the document outline. Here IB can recommend fixes for these issues.

Identifying Unsatisfiable Constraints

Interface Builder can catch the most obvious conflicts, but it cannot catch conflicts that happen at runtime or help when you create your layouts programmatically.

So, even though you should always fix all the issues that Interface Builder identifies, fixing the obvious errors is not sufficient. You still need to perform runtime testing across the full range of screen sizes, orientations, dynamic type sizes, and languages that you intend to support.

Identifying Unsatisfiable Constraints

When the system detects a unsatisfiable layout at runtime, it performs the following steps:

1. Auto Layout identifies the set of conflicting constraints.

2. It breaks one of the conflicting constraints and checks the layout. The system continues to break constraints until it finds a valid layout.

3. Auto Layout logs information about the conflict and the broken constraints to the console.

   This fallback system lets the app proceed, while still attempting to present something meaningful to the user.

Identifying Unsatisfiable Constraints

However, the effect of breaking constraints can vary greatly from layout to layout, or even from build to build.

In many cases, the missing constraints may not have any visible effect. The view hierarchy appears exactly as you expected. In other cases, the missing constraints can cause entire sections of the view hierarchy to be misplaced, missized, or to disappear entirely.

It is often tempting to ignore errors when they don’t have an obvious effect—after all, they don’t change the app’s behavior. However, any change to the view hierarchy or SDK could also alter the set of broken constraints, suddenly producing an obviously broken layout.

Therefore, always fix unsatisfiable constraint errors when you detect them. To help ensure that you catch nonobvious errors during testing, set a symbolic breakpoint for UIViewAlertForUnsatisfiableConstraints.

Preventing Unsatisfiable Constraints

Unsatisfiable constraints are relatively easy to fix.

The system tells you when unsatisfiable constraints occur and provides you with a list of conflicting constraints.

As soon as you know about the error, the solution is typically very straightforward. Either remove one of the constraints, or change it to an optional constraint.

There are, however, a few common issues worth examining in more detail.

Preventing Unsatisfiable Constraints

Unsatisfiable constraints often occur when programmatically adding views to the view hierarchy.

By default, new views have their translatesAutoresizingMaskIntoConstraints property set to YES.

Interface Builder automatically sets this property to NO when you begin drawing constraints to a view in the canvas.

However, when you’re programmatically creating and laying out your views, you need to set the property to NO before adding your own constraints.

Preventing Unsatisfiable Constraints

Unsatisfiable constraints often occur when a view hierarchy is presented in a space that is too small for it.

You can usually predict the minimum amount of space that your view has access to and design your layout appropriately.

However, both internationalization and dynamic type can cause the view’s content to be much bigger than expected.

You may want to build in failure points, so that your layout fails in a predictable, controlled manner.

Preventing Unsatisfiable Constraints

Consider converting some of your required constraints into high-priority optional constraints. These constraints lets you control where your layout will break when a conflict occurs.

For example, give your failure point a priority of 999 or less. Under most circumstances, this high-priority constraint acts as if it were required; however, when a conflict occurs, the high-priority constraint breaks, protecting the rest of your layout.

Similarly, avoid giving views with an intrinsic content size a required content-hugging or compression-resistance priority. Typically, a control's size acts as an ideal failure point. The control can be a little bigger or a little smaller without having any meaningful effect on the layout.

Yes, there are controls that should only be displayed at their intrinsic content size; however, even in these cases it is usually better to have a control that is a few points off rather than just letting your layout break in unpredictable ways.

Ambiguous Layouts

Ambiguous Layouts

Ambiguous layouts occur when the system of constraints has two or more valid solutions. There are two main causes:

- The layout needs additional constraints to uniquely specify the position and location of every view. Just add constraints to uniquely specify both the view's position and its size.

- The layout has conflicting optional constraints with the same priority, and the system does not know which constraint it should break.
  Here, you need to tell the system which constraint it should break, by changing the priorities so that they are no longer equal. The system breaks the constraint having the lowest priority first.

Detecting Ambiguous Layouts

As with unsatisfiable layouts, Interface Builder can often detect, and offer suggestions to fix, ambiguous layouts at design time.

Also Interface Builder cannot detect all possible ambiguities. Many errors can be found only through testing.

When an ambiguous layout occurs at runtime, Auto Layout chooses one of the possible solutions to use. This means the layout may or may not appear as you expect. Furthermore, there are no warnings written to the console, and there is no way to set a breakpoint for ambiguous layouts.

As a result, ambiguous layouts are often harder to detect and identify than unsatisfiable layouts. Even if the ambiguity does have an obvious, visible effect, it can be hard to determine whether the error is due to ambiguity or to an error in your layout logic.

Detecting Ambiguous Layouts

Methods

- hasAmbiguousLayout. Available for both iOS and OS X. Call this method on a misplaced view. It returns YES if the view’s frame is ambiguous. Otherwise, it returns NO.
- exerciseAmbiguityInLayout. Available for both iOS and OS X. Call this method on a view with ambiguous layout. This will toggle the system between the possible valid solutions.
- constraintsAffectingLayoutForAxis:. Available for iOS. Call this method on a view. It returns an array of all the constraints affecting that view along the specified axis.
- constraintsAffectingLayoutForOrientation:. Available for OS X. Call this method on a view. It returns an array of all the constraints affecting that view along the specified orientation.
- _autolayoutTrace. Available as a private method in iOS. Call this method on a view. It returns a string with diagnostic information about the entire view hierarchy containing that view. Ambiguous views are labeled, and so are views that have translatesAutoresizingMaskIntoConstraints set to YES.

Detecting Ambiguous Layouts

Console

You may run these commands in the console.

For example, after the breakpoint halts execution, type call

[self.myView exerciseAmbiguityInLayout] into the console window to call the exerciseAmbiguityInLayout method on the myView object.

Similarly, type po [self.myView autolayoutTrace] to print out diagnostic information about the view hierarchy containing myView.

Logical Errors

Logical errors are simply bugs. Somewhere, you have an assumption that is faulty. Perhaps it’s an assumption about how Auto Layout calculates the views’ frames. Perhaps it’s an assumption about the set of constraints that you’ve created, or the view properties you’ve set. Perhaps it’s an assumption about how the constraints interact to create complex behaviors. Regardless, something somewhere does not quite match your mental model.

There are no tools or step-by-step instructions here. Fixing logical errors typically involves experiments and iterative tests, both to identify the problem and to figure out how to fix it. There are, however, a few suggestions that may help.

Logical Errors

- Review the existing constraints. Make sure you haven’t missed any constraints or accidentally added unwanted constraints. Make sure all the constraints are attached to the correct items and attributes.
- Double-check the view frames. Make sure nothing is getting unexpectedly stretched or shrunk.
  This is particularly important for views with invisible backgrounds, like labels or buttons. It may not be obvious when these items are unexpectedly resized.
  One symptom of resizing is that baseline-aligned views no longer line up properly. This is because the baseline alignment works only when the view is displayed at its intrinsic content height. If you stretch or shrink the view vertically, the text mysteriously appears in the wrong location.
- If a control should always match its intrinsic content size, give it a very high content-hugging and compression-resistance priority (for example, 999).
- Look for any assumptions that you’re making about the layout, and add explicit constraints to make sure those assumptions are true.
  Remember, unsatisfiable layouts are generally the easiest problems to find and fix. Add additional constraints until you have a conflict, then examine and fix the conflict.
- Try to understand why the given constraints are producing the results that you see. If you understand it, you’re well on the way to fixing it.
- Experiment with alternative constraints. Auto Layout typically gives you a number of different solutions for the same problem. Trying an different approach may fix the problem or at least make it easier to spot the mistake.

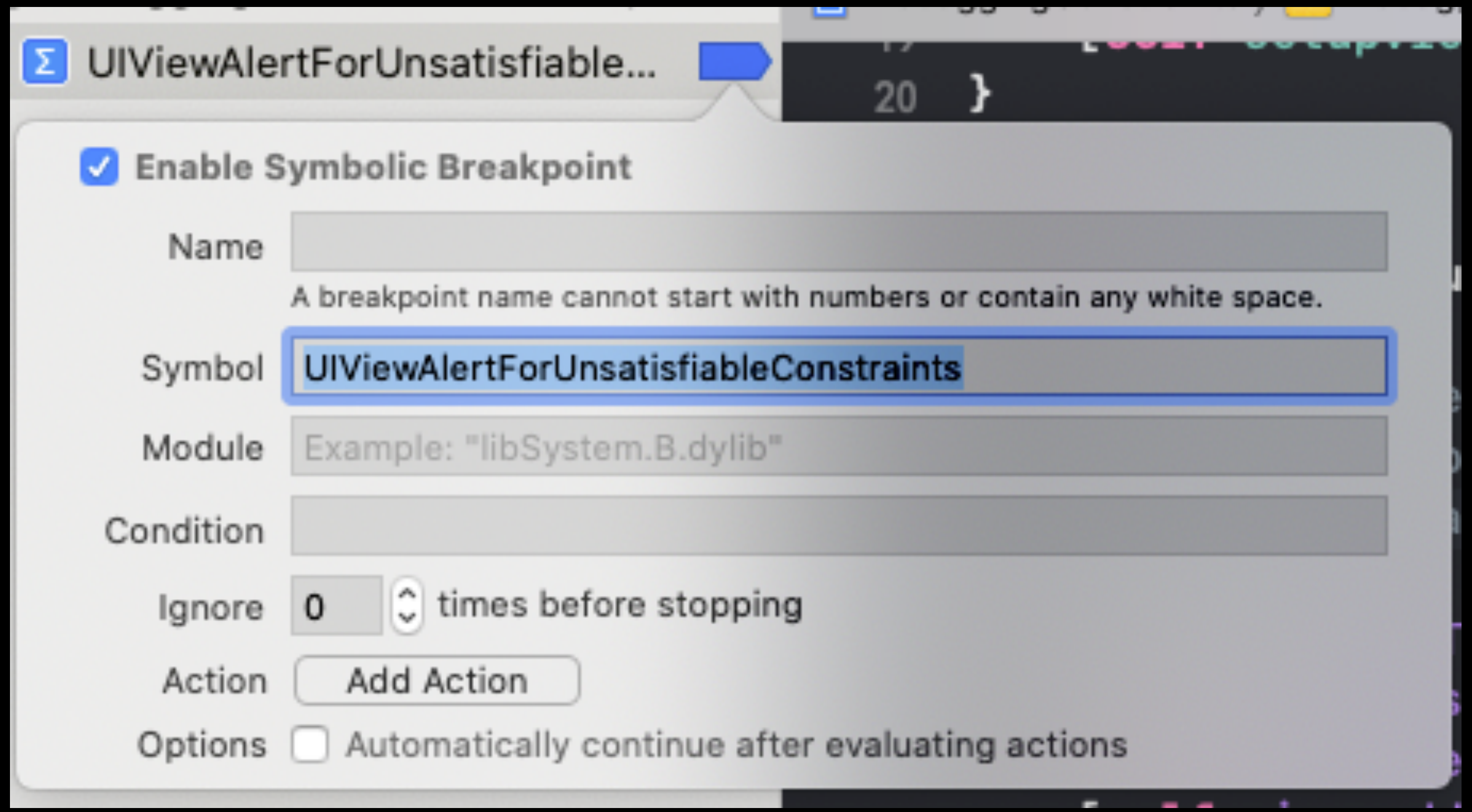
Debugging Tips And Tricks

Symbolic breakpoints

If you skip over the list of conflicting constraints and look at the hints at the end of the log, you will see a suggestion to add the symbolic breakpoint.

```
Make a symbolic breakpoint at UIViewAlertForUnsatisfiableConstraints to
    catch this in the debugger.
```

Use the breakpoint navigator or Debug › Breakpoints › Create Symbolic Breakpoint to add the breakpoint.



UIViewAlertForUnsatisfiable...

[✓] Enable Symbolic Breakpoint

Name:
A breakpoint name cannot start with numbers or contain any white space.

Symbol: UIViewAlertForUnsatisfiableConstraints

Module: Example: "libSystem.B.dylib"

Condition:

Ignore: 0 times before stopping

Action: Add Action

Options: [ ] Automatically continue after evaluating actions

The debugger now breaks when the layout engine detects an unsatisfiable constraint. Setting the breakpoint takes you quickly to the area of code that's adding the conflicting constraints.

Understanding the Logs

VFL

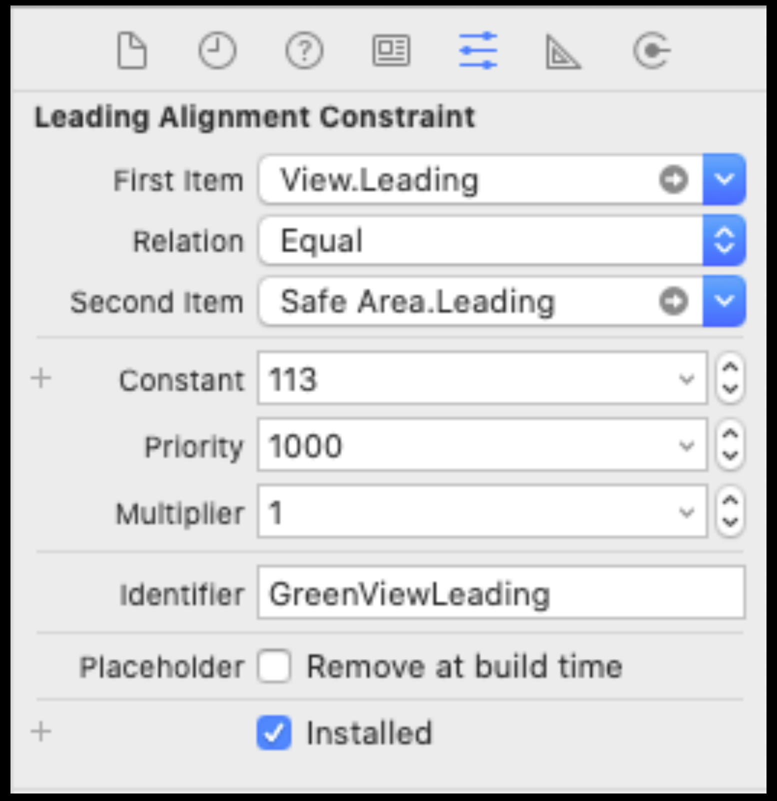
The first line of log tells us what sort of a layout problem we have: “Unable to simultaneously satisfy constraints.”

Constraints are written to the console using the Visual Format Language. Even if you never use the Visual Format Language to create your own constraints, you must be able to read and understand it to effectively debug Auto Layout issues.

The layout engine logs a considerable amount of detail when it has a problem, mixing our views and constraints in with those created by the system. It can help to add identifiers to your constraints and views, so they stand out in the log.

Understanding the Logs

Adding Identifiers to the Logs



Leading Alignment Constraint

First Item: View.Leading
Relation: Equal
Second Item: Safe Area.Leading
Constant: 113
Priority: 1000
Multiplier: 1
Identifier: GreenViewLeading
Placeholder: [ ] Remove at build time
[✓] Installed

All constraints have an identifier property that you can set in code or in Interface Builder.

When working with batches of constraints, you can set the same identifier for each logical group of constraints.

```
for (NSLayoutConstraint *constraint in centerConstraints) {
    constraint.identifier = @"BrokenButtonCenter";
}
```

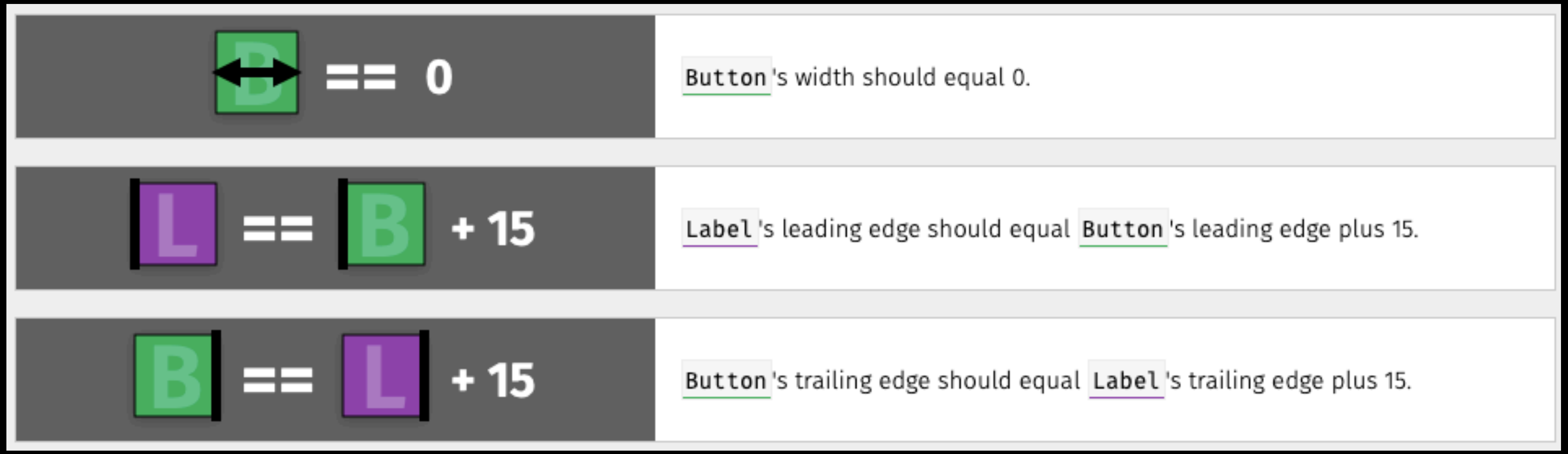
All views and layout guides have an accessibility identifier (don't confuse this with the accessibility label that VoiceOver reads).

If the view has an obvious textual component, Xcode uses that as an identifier. For example, Xcode uses a label's text, a button's title, or a text field's placeholder to identify these views.

```
self.view.accessibilityIdentifier = @"RootView"
```

Understanding the Logs

You can also get visual representation of broken constraints on wtfautolayout.com

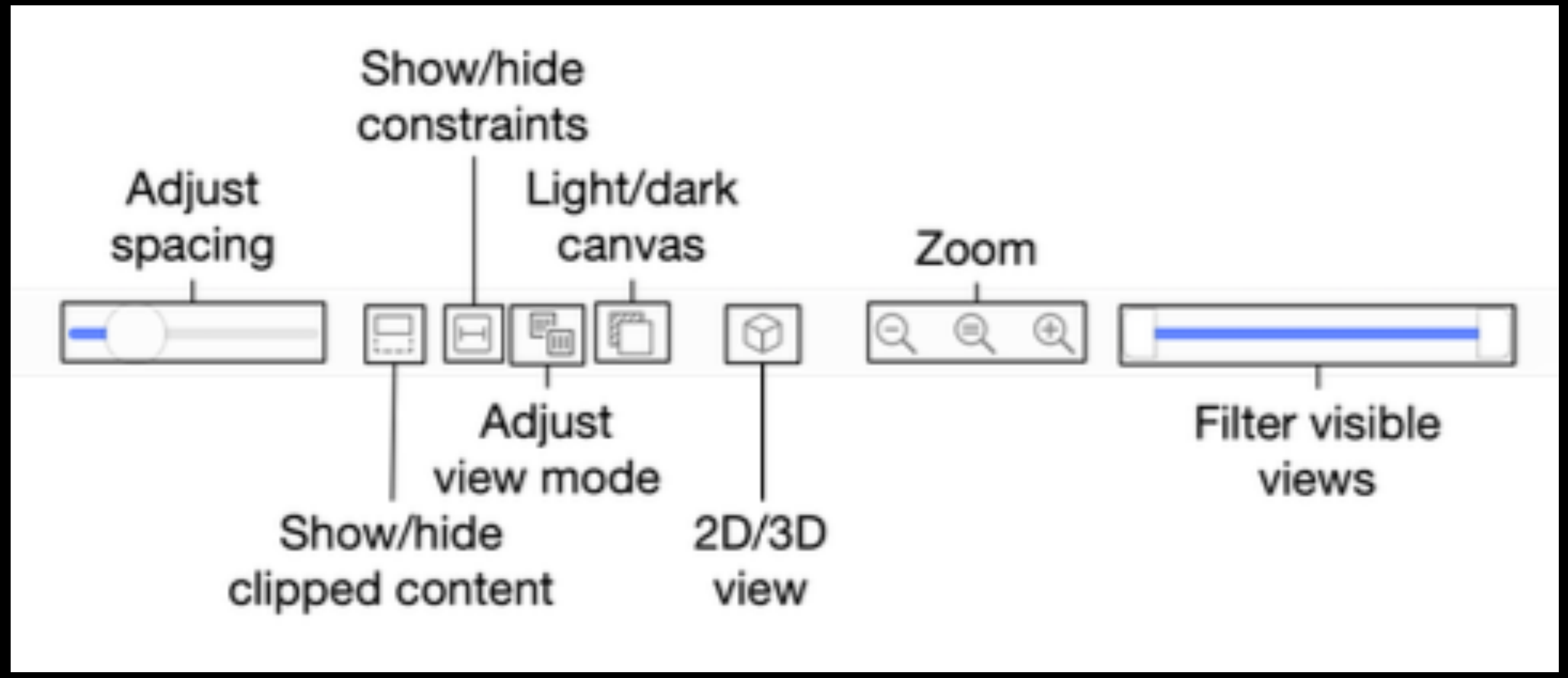


Button's width should equal 0.

Label's leading edge should equal Button's leading edge plus 15.

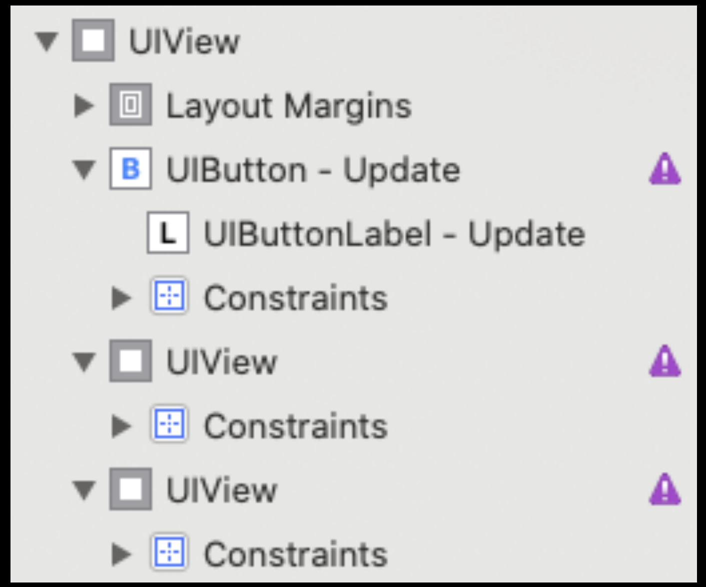
Button's trailing edge should equal Label's trailing edge plus 15.

View Debugger

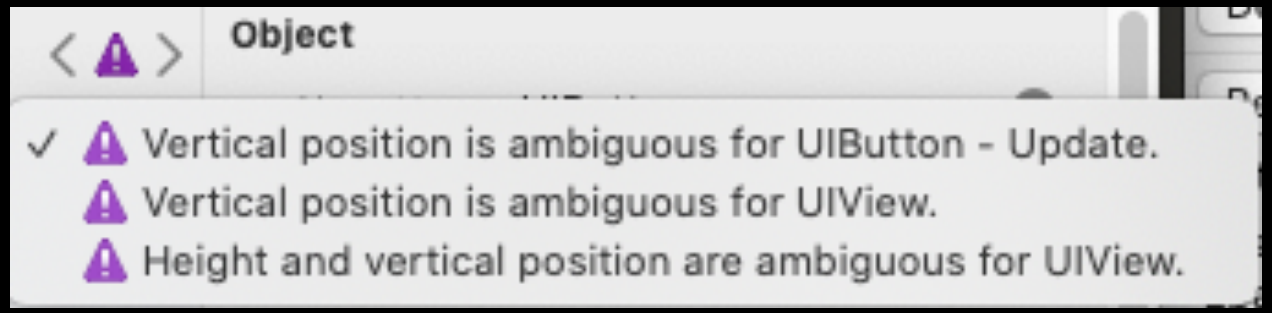


When you're having problems with layouts, the Xcode view debugger is usually a good place to start. You can rotate and zoom the canvas or use the toolbar below the canvas to filter what you see.

Also view debugger tells you if any views in the view hierarchy have an ambiguous layout. In this case, it has marked both buttons with a purple indicator.



```
▼ UIView
  ▶ Layout Margins
  ▼ UIButton - Update          ⚠
      UIButtonLabel - Update
    ▶ Constraints
  ▼ UIView                     ⚠
    ▶ Constraints
  ▼ UIView                     ⚠
    ▶ Constraints
```



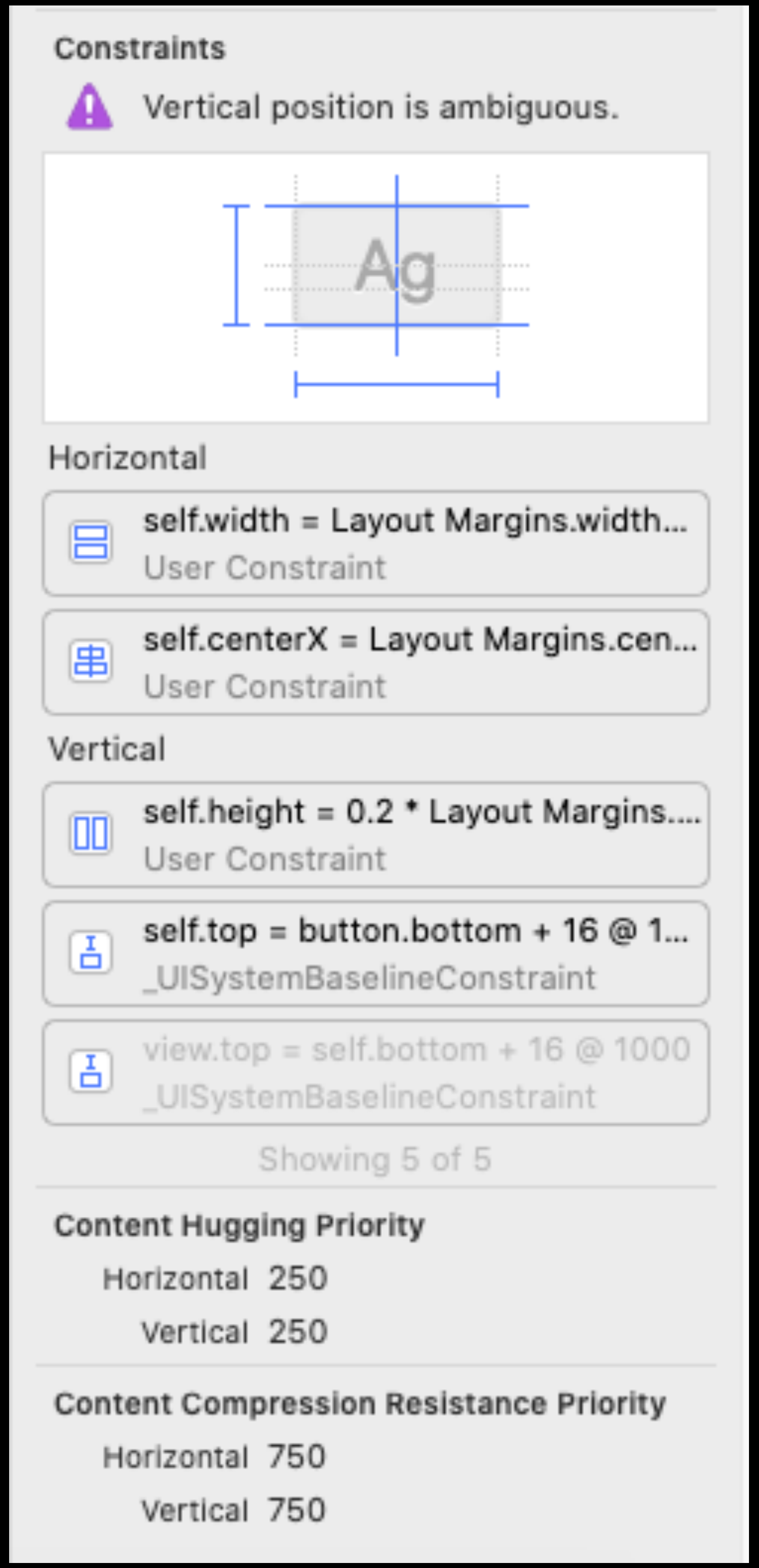
```
Object
✓ ⚠ Vertical position is ambiguous for UIButton - Update.
  ⚠ Vertical position is ambiguous for UIView.
  ⚠ Height and vertical position are ambiguous for UIView.
```

View Debugger

If you click on the cancel button the Size Inspector shows the view frame, bounds, the constraints involving the view and the content priorities.

The Size Inspector makes it clear what is ambiguous and lists the horizontal and vertical constraints involving the view. Selecting any of the constraints highlights it in the canvas.

Only the constraints in bold are affecting the layout of the button. A lot easier to understand than the console log.



Constraints

Vertical position is ambiguous.

Horizontal

self.width = Layout Margins.width...
User Constraint

self.centerX = Layout Margins.cen...
User Constraint

Vertical

self.height = 0.2 * Layout Margins....
User Constraint

self.top = button.bottom + 16 @ 1...
_UISystemBaselineConstraint

view.top = self.bottom + 16 @ 1000
_UISystemBaselineConstraint

Showing 5 of 5

Content Hugging Priority

Horizontal	250
Vertical	250

Content Compression Resistance Priority

Horizontal	750
Vertical	750

Private Debug Methods

Private Debug Methods

The view debugger is generally the best way to investigate layout problems. Sometimes though you may prefer to work from the command-line in the debugger or you may be prototyping a layout in a playground where the debugger is not available.

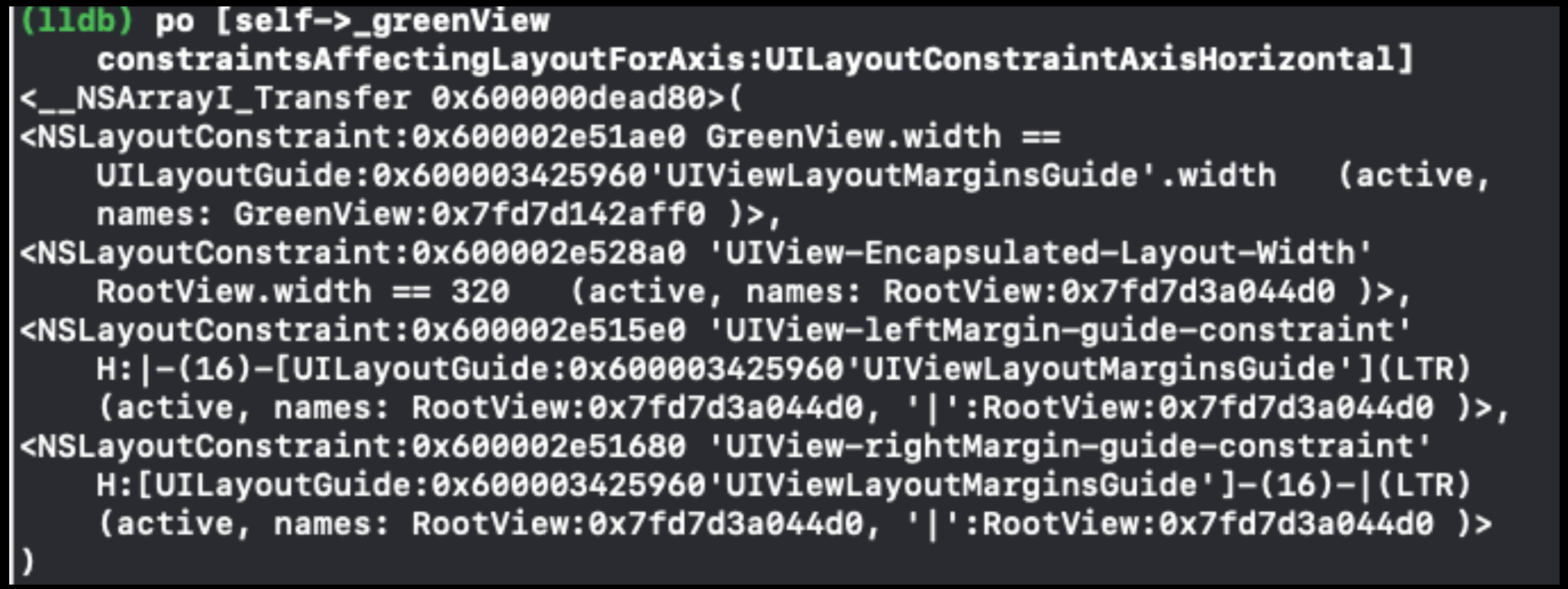
There are some private methods and properties you can use in your code or from the debugger that show much the same information as the view debugger.

These are private methods and properties intended for debugging.

! Don’t use any of them in a released version of your application.

Private Debug Methods

Viewing Constraints That Affect The Layout



```
(lldb) po [self->_greenView
    constraintsAffectingLayoutForAxis:UILayoutConstraintAxisHorizontal]
<__NSArrayI_Transfer 0x600000dead80>(
<NSLayoutConstraint:0x600002e51ae0 GreenView.width ==
    UILayoutGuide:0x600003425960'UIViewLayoutMarginsGuide'.width   (active,
    names: GreenView:0x7fd7d142aff0 )>,
<NSLayoutConstraint:0x600002e528a0 'UIView-Encapsulated-Layout-Width'
    RootView.width == 320   (active, names: RootView:0x7fd7d3a044d0 )>,
<NSLayoutConstraint:0x600002e515e0 'UIView-leftMargin-guide-constraint'
    H:|-(16)-[UILayoutGuide:0x600003425960'UIViewLayoutMarginsGuide'](LTR)
    (active, names: RootView:0x7fd7d3a044d0, '|':RootView:0x7fd7d3a044d0 )>,
<NSLayoutConstraint:0x600002e51680 'UIView-rightMargin-guide-constraint'
    H:[UILayoutGuide:0x600003425960'UIViewLayoutMarginsGuide']-(16)-|(LTR)
    (active, names: RootView:0x7fd7d3a044d0, '|':RootView:0x7fd7d3a044d0 )>
)
```

When debugging a problem, you’re often not interested in seeing every constraint involving a view. Also unless you’re using an aspect ratio constraint the horizontal and vertical constraints don’t interact.

So you can reduce the clutter by looking at just those constraints impacting the layout along one axis.

The constraintsAffectingLayoutForAxis: method returns the constraints impacting the layout of a view or layout guide for the specified axis.

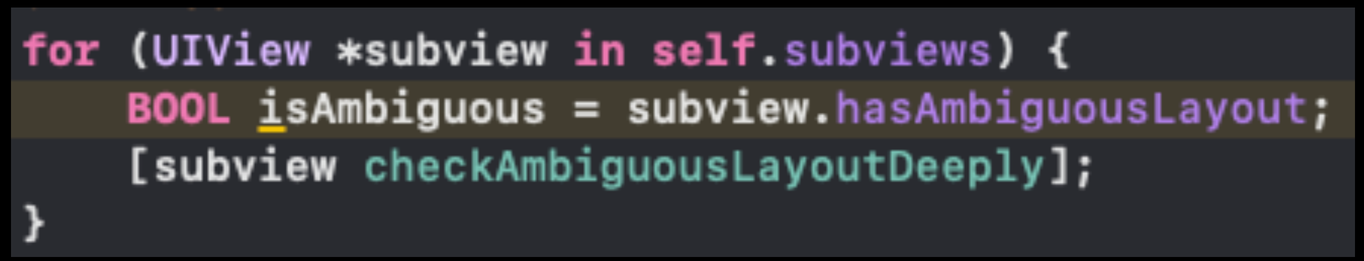
Use it with caution: while this provides a good starting point for debugging, there is no guarantee that the returned set of constraints will include all of the constraints that have an impact on the view’s layout in the given orientation.

Private Debug Methods

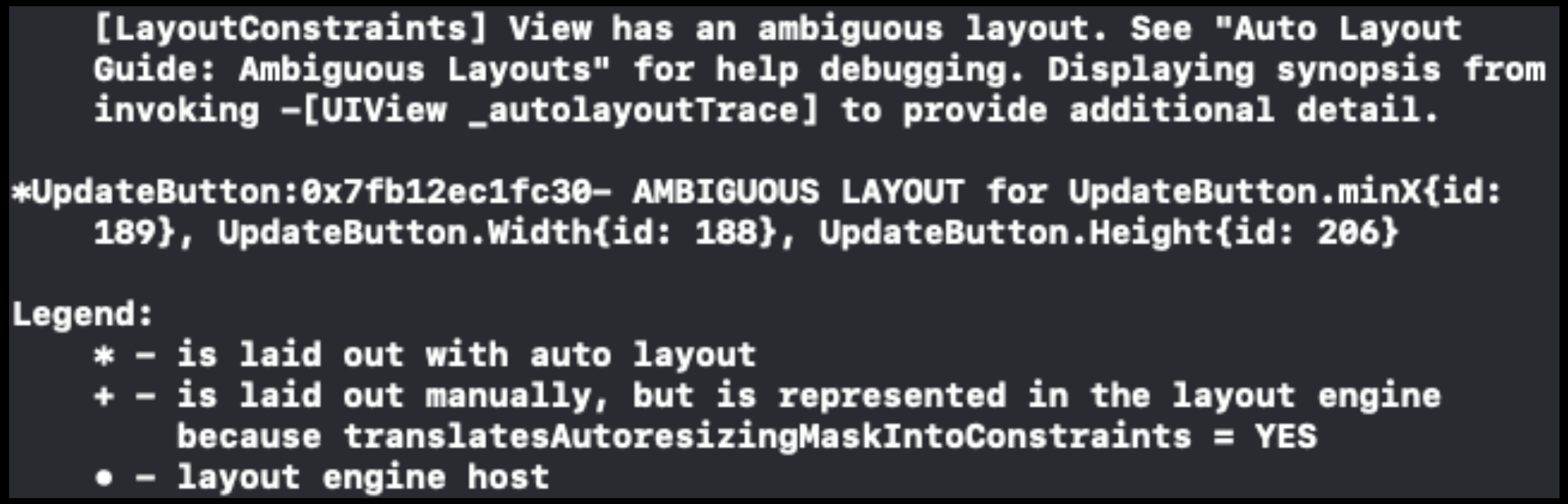
Checking For Ambiguous Layout

Both UIView and UILayoutGuide have a boolean instance property hasAmbiguousLayout that returns true if their layout is ambiguous.

This doesn't tell you if a subview of the view has an ambiguous layout. You have to check the full view hierarchy yourself to be sure your layout is without problems.



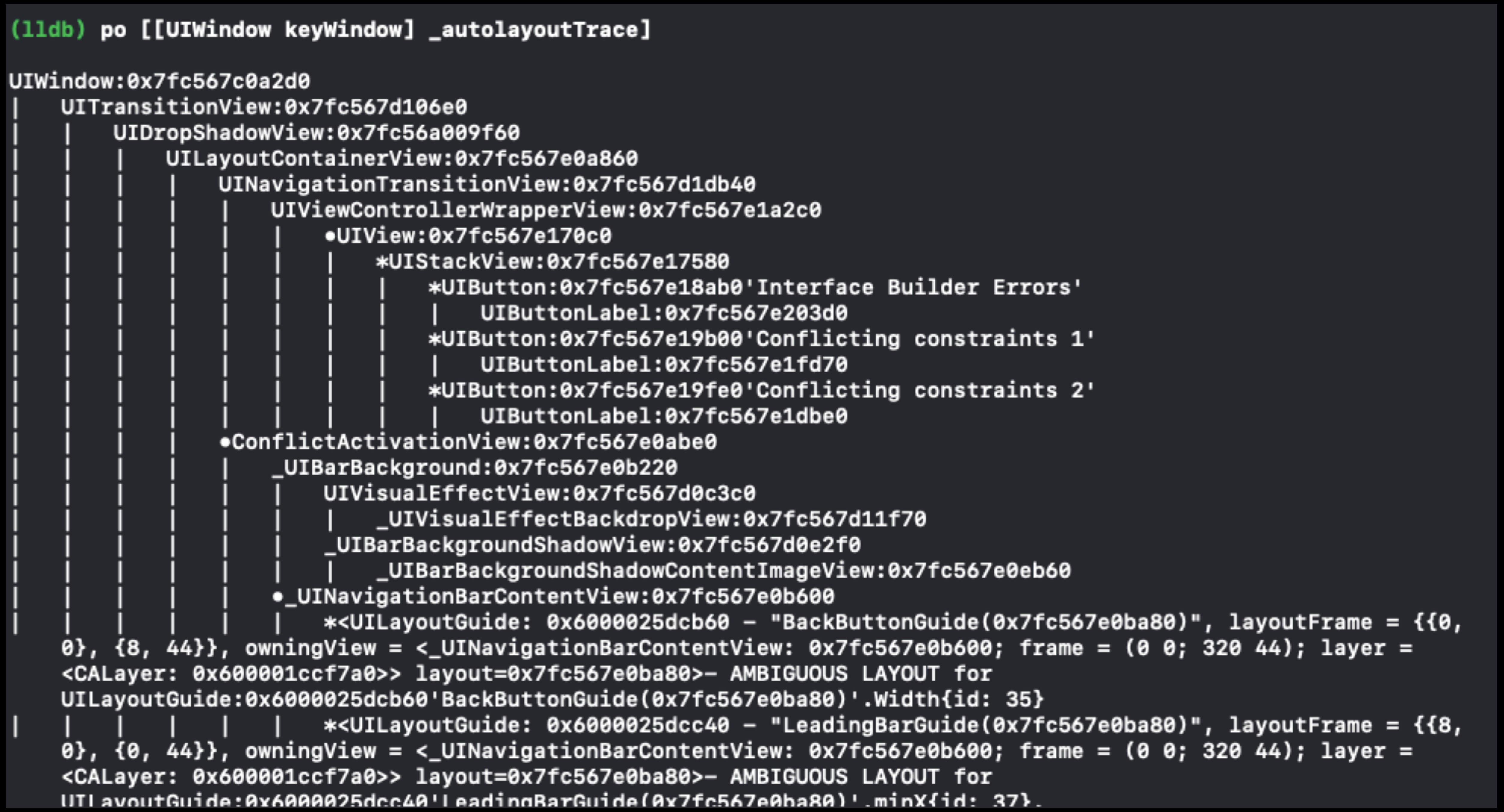
```
for (UIView *subview in self.subviews) {
    BOOL isAmbiguous = subview.hasAmbiguousLayout;
    [subview checkAmbiguousLayoutDeeply];
}
```



```
    [LayoutConstraints] View has an ambiguous layout. See "Auto Layout
    Guide: Ambiguous Layouts" for help debugging. Displaying synopsis from
    invoking -[UIView _autolayoutTrace] to provide additional detail.

*UpdateButton:0x7fb12ec1fc30- AMBIGUOUS LAYOUT for UpdateButton.minX{id:
    189}, UpdateButton.Width{id: 188}, UpdateButton.Height{id: 206}

Legend:
    * - is laid out with auto layout
    + - is laid out manually, but is represented in the layout engine
        because translatesAutoresizingMaskIntoConstraints = YES
    • - layout engine host
```

Private Debug Methods

Tracing The View Hierarchy



```
(lldb) po [[UIWindow keyWindow] _autolayoutTrace]

UIWindow:0x7fc567c0a2d0
|   UITransitionView:0x7fc567d106e0
|   |   UIDropShadowView:0x7fc56a009f60
|   |   |   UILayoutContainerView:0x7fc567e0a860
|   |   |   |   UINavigationTransitionView:0x7fc567d1db40
|   |   |   |   |   UIViewControllerWrapperView:0x7fc567e1a2c0
|   |   |   |   |   |   •UIView:0x7fc567e170c0
|   |   |   |   |   |   |   *UIStackView:0x7fc567e17580
|   |   |   |   |   |   |   |   *UIButton:0x7fc567e18ab0'Interface Builder Errors'
|   |   |   |   |   |   |   |   |   UIButtonLabel:0x7fc567e203d0
|   |   |   |   |   |   |   |   *UIButton:0x7fc567e19b00'Conflicting constraints 1'
|   |   |   |   |   |   |   |   |   UIButtonLabel:0x7fc567e1fd70
|   |   |   |   |   |   |   |   *UIButton:0x7fc567e19fe0'Conflicting constraints 2'
|   |   |   |   |   |   |   |   |   UIButtonLabel:0x7fc567e1dbe0
|   |   |   |   •ConflictActivationView:0x7fc567e0abe0
|   |   |   |   |   _UIBarBackground:0x7fc567e0b220
|   |   |   |   |   |   UIVisualEffectView:0x7fc567d0c3c0
|   |   |   |   |   |   |   _UIVisualEffectBackdropView:0x7fc567d11f70
|   |   |   |   |   |   _UIBarBackgroundShadowView:0x7fc567d0e2f0
|   |   |   |   |   |   |   _UIBarBackgroundShadowContentImageView:0x7fc567e0eb60
|   |   |   |   |   •_UINavigationBarContentView:0x7fc567e0b600
|   |   |   |   |   |   *<UILayoutGuide: 0x6000025dcb60 - "BackButtonGuide(0x7fc567e0ba80)", layoutFrame = {{0,
    0}, {8, 44}}, owningView = <_UINavigationBarContentView: 0x7fc567e0b600; frame = (0 0; 320 44); layer =
    <CALayer: 0x600001ccf7a0>> layout=0x7fc567e0ba80>- AMBIGUOUS LAYOUT for
    UILayoutGuide:0x6000025dcb60'BackButtonGuide(0x7fc567e0ba80)'.Width{id: 35}
|   |   |   |   |   |   *<UILayoutGuide: 0x6000025dcc40 - "LeadingBarGuide(0x7fc567e0ba80)", layoutFrame = {{8,
    0}, {0, 44}}, owningView = <_UINavigationBarContentView: 0x7fc567e0b600; frame = (0 0; 320 44); layer =
    <CALayer: 0x600001ccf7a0>> layout=0x7fc567e0ba80>- AMBIGUOUS LAYOUT for
    UILayoutGuide:0x6000025dcc40'LeadingBarGuide(0x7fc567e0ba80)'.minX{id: 37}.
```

Instead of calling hasAmbiguousLayout on each subview you can use _autolayoutTrace.

This dumps the view hierarchy starting from the top window and marks any views with an ambiguous layout in the output. As with the unsatisfiable constraints error message, we still have to figure out which view belongs to the memory address of the printout.

Private Debug Methods

Tracing The View Hierarchy - Swift Version

Note: it’s not directly available from Swift. As a workaround you can use value(forKey:).

```
print(view.value(forKey: "_autolayoutTrace")!)
```

In the debugger, you can also fall back to using the Objective-C method. Since it doesn’t matter where in the view hierarchy we call the method we can start from the key window of the application.

```
(lldb) expr -l objc -o -- [[[UIApplication sharedApplication] keyWindow] _autolayoutTrace]
```

The -o prints the Objective-C description of the object (like po).

You can shorten this even further if you use the private keyWindow class method of UIWindow.

```
(lldb) expr -l objc -o -- [[UIWindow keyWindow] _autolayoutTrace]
```

LLDB Quick Start Guide

Private Debug Methods

Exercising Ambiguity

Another more visual way to spot ambiguous layouts is to use exerciseAmbiguityInLayout. This will randomly change the view's frame between valid values.

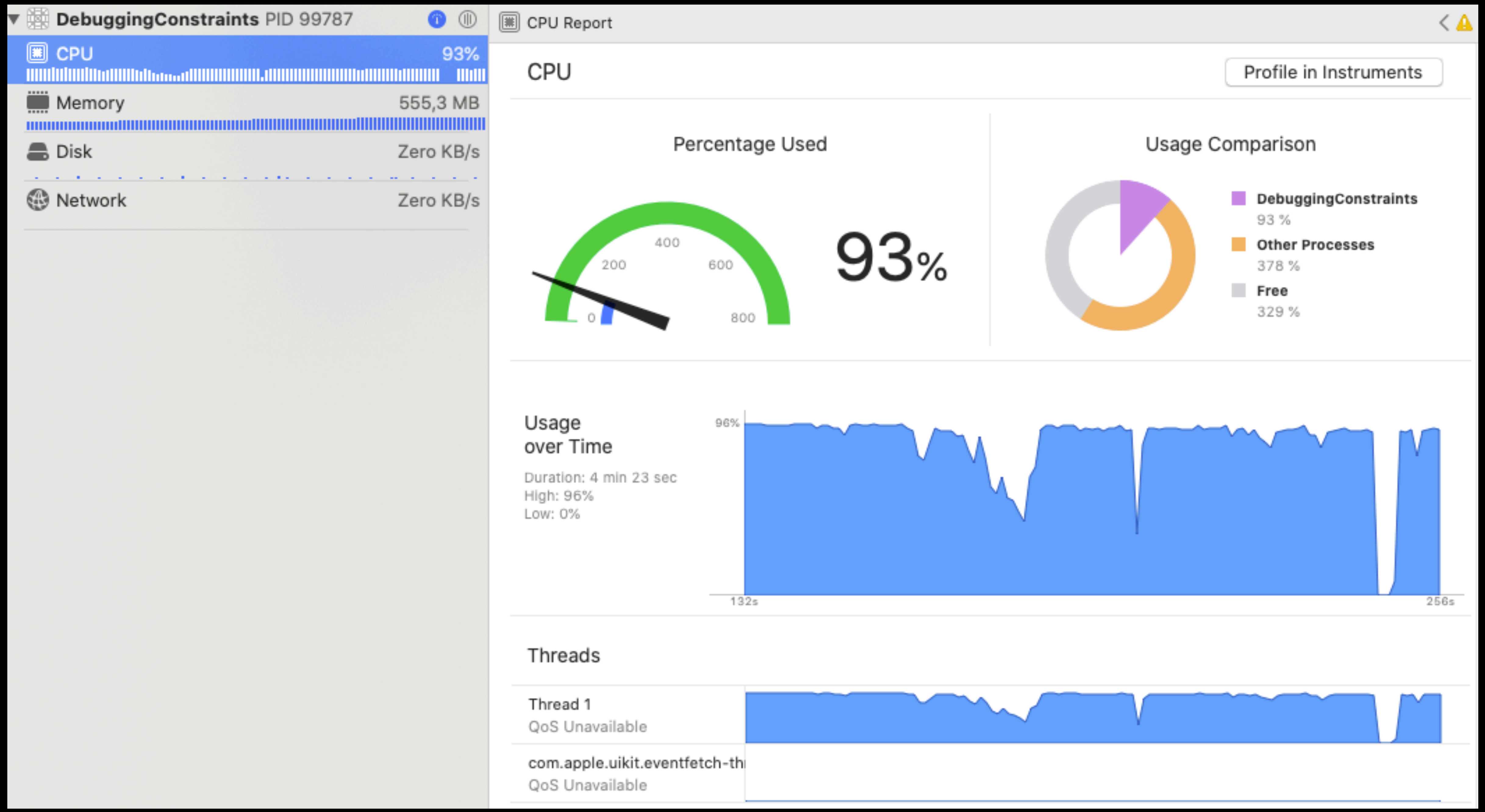
However, calling this method once will also just change the frame once. So chances are that you will not see this change at all when you start your app.

It's a good idea to create a helper method with timer which traverses through the whole view hierarchy and makes all views that have an ambiguous layout “jiggle."

We might call this from viewDidAppear after the layout engine has done its work.

Layout Loops

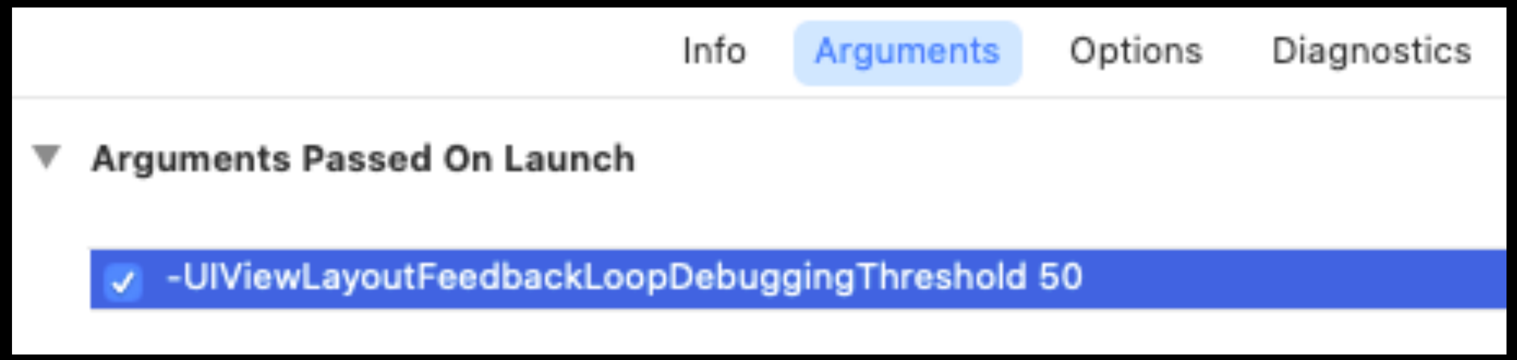
Layout Loops



While much less common than the other layout problems it’s possible to get the layout engine into a loop.

It’s hard to miss when it happens as your user interface locks up, the device CPU utilization goes to 100%, and the memory consumption starts to grow.

Layout Loops



Arguments Passed On Launch

✓ -UIViewLayoutFeedbackLoopDebuggingThreshold 50

There’s a launch argument you can add that turns on layout feedback loop debugging:

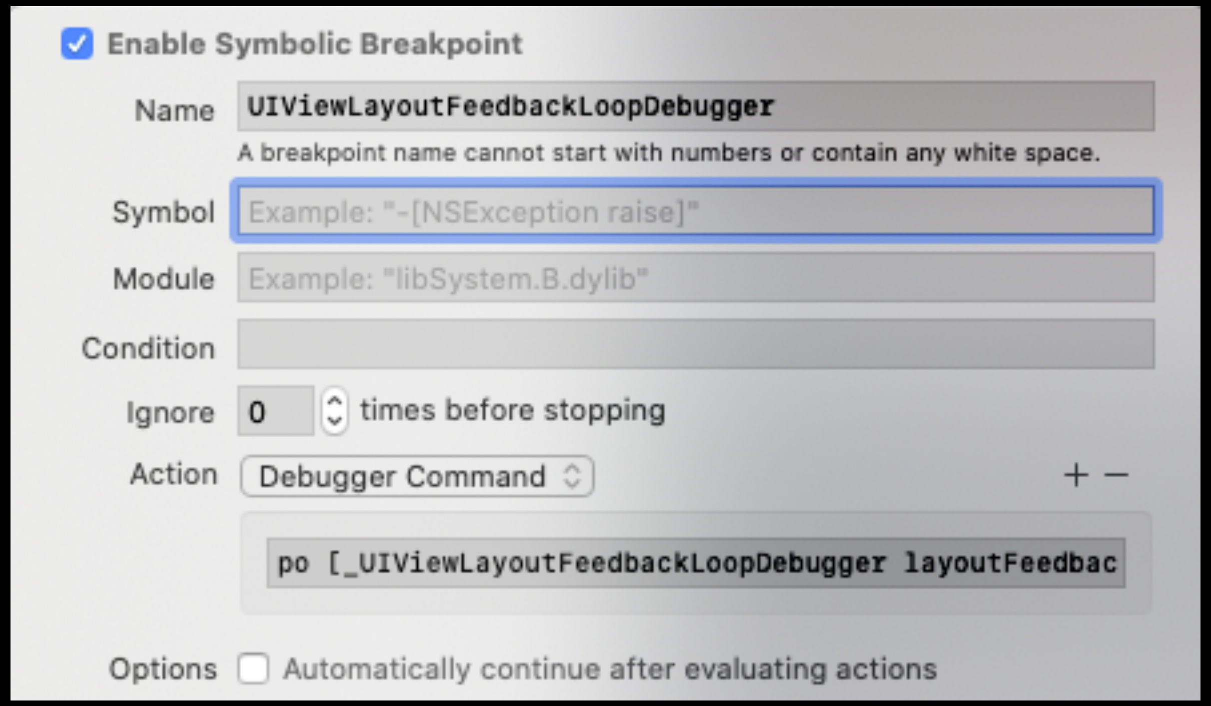
-UIViewLayoutFeedbackLoopDebuggingThreshold 50

The layout engine keeps a count of the calls to layoutSubviews during a layout pass. When it exceeds the threshold, it aborts that App and logs the results. The threshold can be in the range 50 to 1000. Choose a value that’s higher than the number of subviews in your layout, but 50 is a good starting point. From the Xcode scheme editor (Product › Scheme › Edit Scheme...) add the launch argument to the run action.

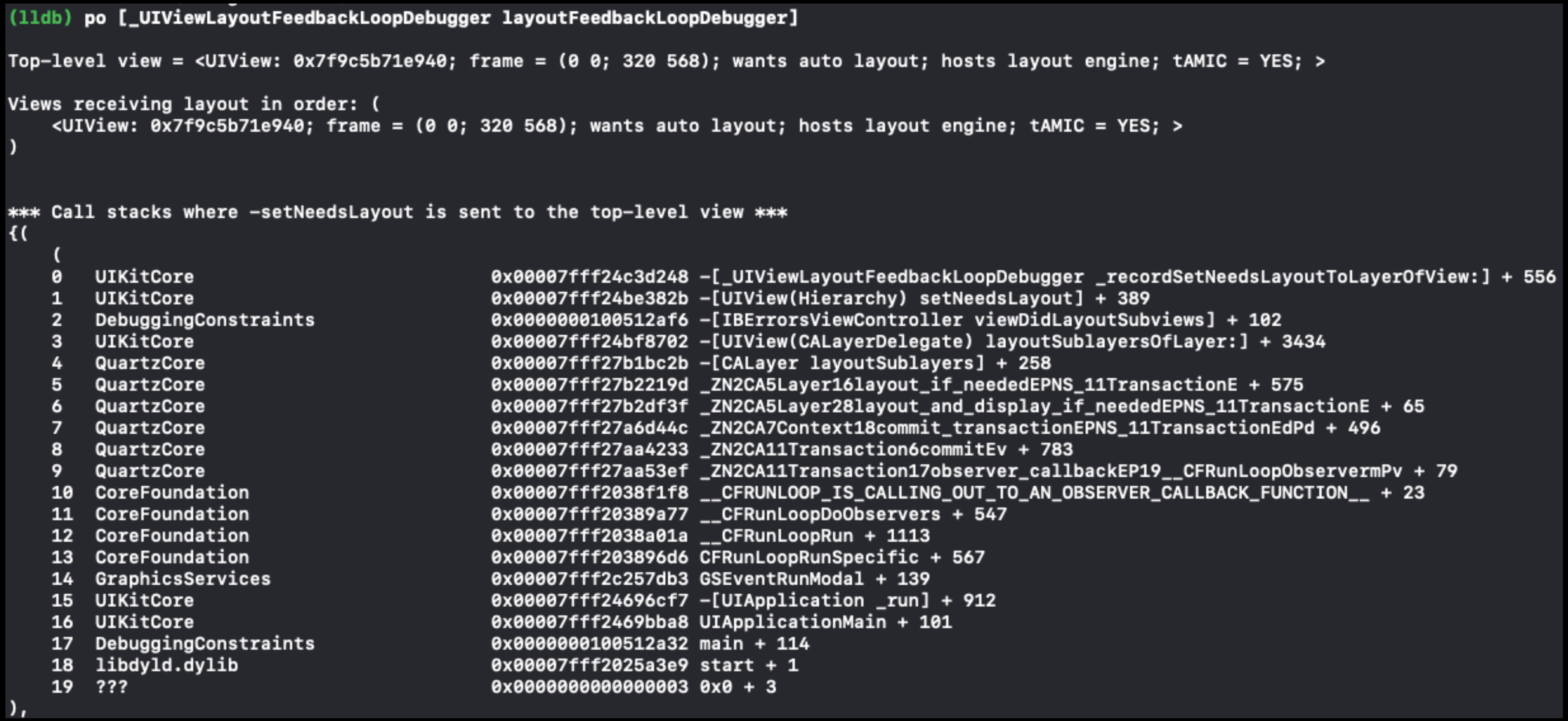
Layout Loops

To print the debug data in the debugger: `po [_UIViewLayoutFeedbackLoopDebugger layoutFeedbackLoopDebugger]`

You can also set an exception breakpoint and have it create the layout feedback debug log.



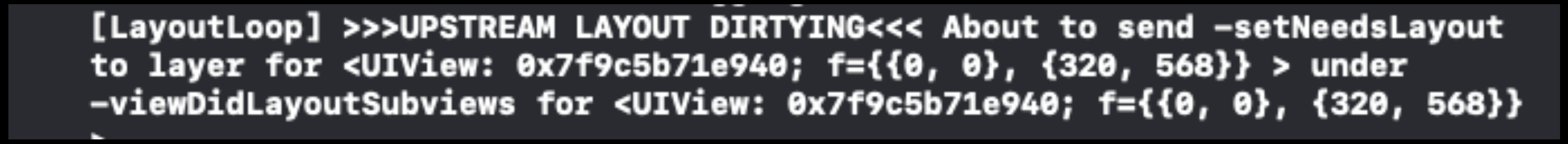
☑ Enable Symbolic Breakpoint	
Name	UIViewLayoutFeedbackLoopDebugger
	A breakpoint name cannot start with numbers or contain any white space.
Symbol	Example: "-[NSException raise]"
Module	Example: "libSystem.B.dylib"
Condition	
Ignore	0 times before stopping
Action	Debugger Command
	po [_UIViewLayoutFeedbackLoopDebugger layoutFeedbac
Options	☐ Automatically continue after evaluating actions



```
(lldb) po [_UIViewLayoutFeedbackLoopDebugger layoutFeedbackLoopDebugger]

Top-level view = <UIView: 0x7f9c5b71e940; frame = (0 0; 320 568); wants auto layout; hosts layout engine; tAMIC = YES; >

Views receiving layout in order: (
    <UIView: 0x7f9c5b71e940; frame = (0 0; 320 568); wants auto layout; hosts layout engine; tAMIC = YES; >
)


*** Call stacks where -setNeedsLayout is sent to the top-level view ***
{(
    (
    0   UIKitCore                           0x00007fff24c3d248 -[_UIViewLayoutFeedbackLoopDebugger _recordSetNeedsLayoutToLayerOfView:] + 556
    1   UIKitCore                           0x00007fff24be382b -[UIView(Hierarchy) setNeedsLayout] + 389
    2   DebuggingConstraints                0x0000000100512af6 -[IBErrorsViewController viewDidLayoutSubviews] + 102
    3   UIKitCore                           0x00007fff24bf8702 -[UIView(CALayerDelegate) layoutSublayersOfLayer:] + 3434
    4   QuartzCore                          0x00007fff27b1bc2b -[CALayer layoutSublayers] + 258
    5   QuartzCore                          0x00007fff27b2219d _ZN2CA5Layer16layout_if_neededEPNS_11TransactionE + 575
    6   QuartzCore                          0x00007fff27b2df3f _ZN2CA5Layer28layout_and_display_if_neededEPNS_11TransactionE + 65
    7   QuartzCore                          0x00007fff27a6d44c _ZN2CA7Context18commit_transactionEPNS_11TransactionEdPd + 496
    8   QuartzCore                          0x00007fff27aa4233 _ZN2CA11Transaction6commitEv + 783
    9   QuartzCore                          0x00007fff27aa53ef _ZN2CA11Transaction17observer_callbackEP19__CFRunLoopObservermPv + 79
    10  CoreFoundation                      0x00007fff2038f1f8 __CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION__ + 23
    11  CoreFoundation                      0x00007fff20389a77 __CFRunLoopDoObservers + 547
    12  CoreFoundation                      0x00007fff2038a01a __CFRunLoopRun + 1113
    13  CoreFoundation                      0x00007fff203896d6 CFRunLoopRunSpecific + 567
    14  GraphicsServices                    0x00007fff2c257db3 GSEventRunModal + 139
    15  UIKitCore                           0x00007fff24696cf7 -[UIApplication _run] + 912
    16  UIKitCore                           0x00007fff2469bba8 UIApplicationMain + 101
    17  DebuggingConstraints                0x0000000100512a32 main + 114
    18  libdyld.dylib                       0x00007fff2025a3e9 start + 1
    19  ???                                 0x0000000000000003 0x0 + 3
),
```

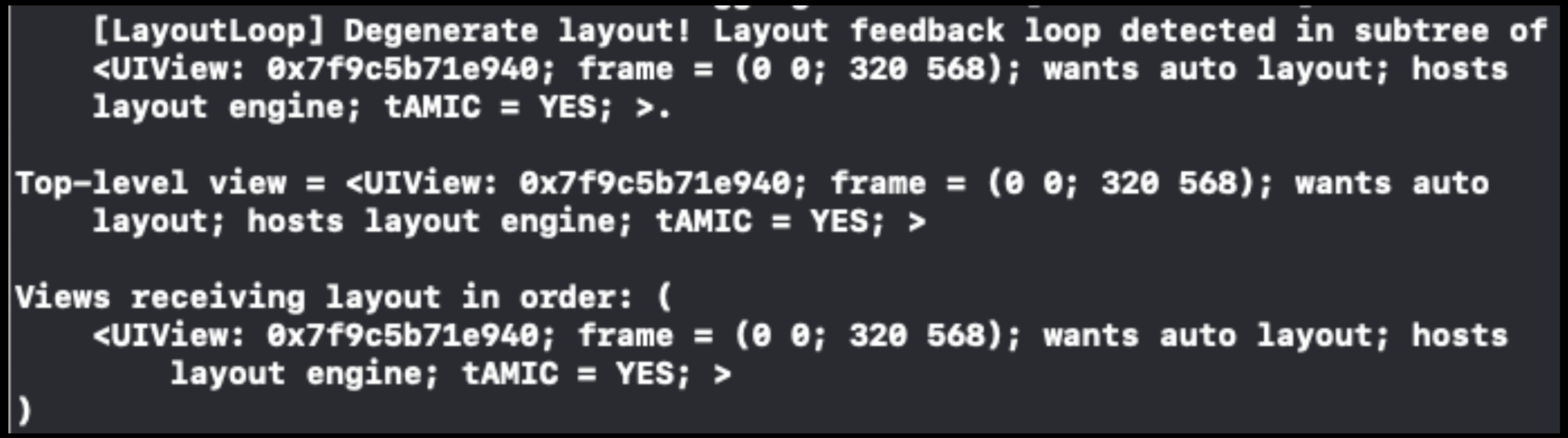
Layout Loops

The logs are verbose but here are the key excerpts that show us what's happening. First a warning that we have a viewDidLayoutSubviews method that's calling setNeedsLayout on a parent view dirtying the layout.



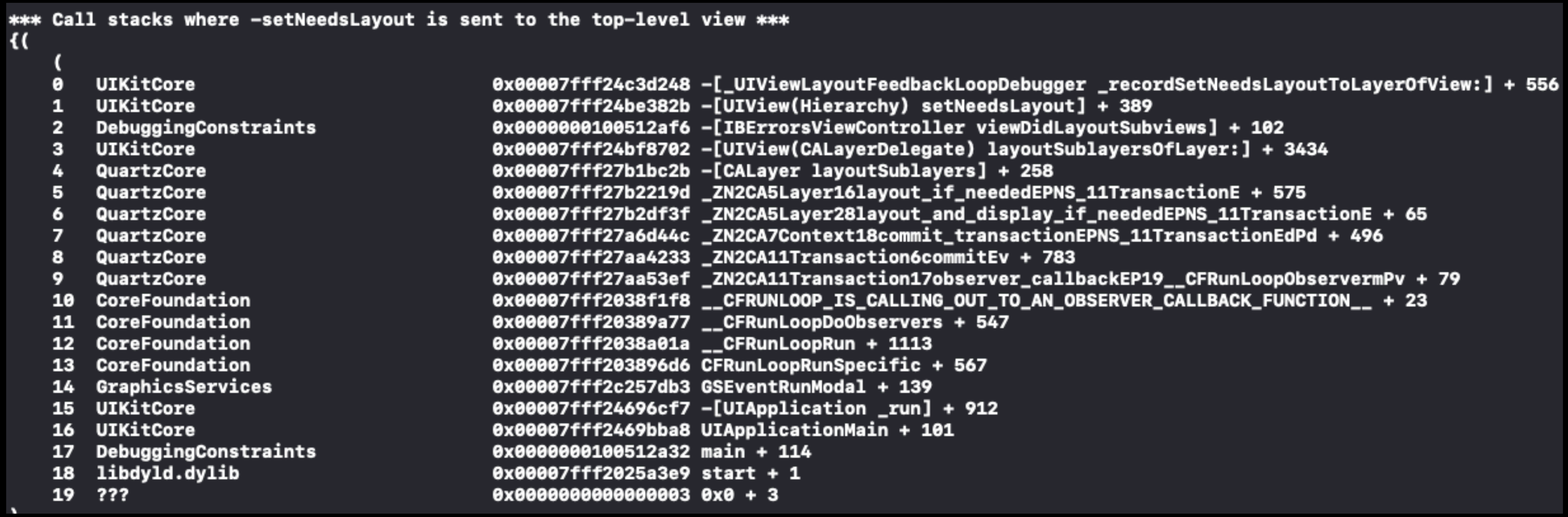
```
[LayoutLoop] >>>UPSTREAM LAYOUT DIRTYING<<< About to send -setNeedsLayout
to layer for <UIView: 0x7f9c5b71e940; f={{0, 0}, {320, 568}} > under
-viewDidLayoutSubviews for <UIView: 0x7f9c5b71e940; f={{0, 0}, {320, 568}}
```

Warnings about a layout loop in what turns out to be the root view of view controller.



```
    [LayoutLoop] Degenerate layout! Layout feedback loop detected in subtree of
    <UIView: 0x7f9c5b71e940; frame = (0 0; 320 568); wants auto layout; hosts
    layout engine; tAMIC = YES; >.

Top-level view = <UIView: 0x7f9c5b71e940; frame = (0 0; 320 568); wants auto
    layout; hosts layout engine; tAMIC = YES; >

Views receiving layout in order: (
    <UIView: 0x7f9c5b71e940; frame = (0 0; 320 568); wants auto layout; hosts
        layout engine; tAMIC = YES; >
)
```

The call stacks show that the culprit is viewDidLayoutSubviews in view controller. Calling setNeedsLayout after the view has finished laying out its subviews causes it to start again in an endless loop. Removing setNeedsLayout removes the layout loop.



```
*** Call stacks where -setNeedsLayout is sent to the top-level view ***
{(
    (
    0   UIKitCore                           0x00007fff24c3d248 -[_UIViewLayoutFeedbackLoopDebugger _recordSetNeedsLayoutToLayerOfView:] + 556
    1   UIKitCore                           0x00007fff24be382b -[UIView(Hierarchy) setNeedsLayout] + 389
    2   DebuggingConstraints                0x0000000100512af6 -[IBErrorsViewController viewDidLayoutSubviews] + 102
    3   UIKitCore                           0x00007fff24bf8702 -[UIView(CALayerDelegate) layoutSublayersOfLayer:] + 3434
    4   QuartzCore                          0x00007fff27b1bc2b -[CALayer layoutSublayers] + 258
    5   QuartzCore                          0x00007fff27b2219d _ZN2CA5Layer16layout_if_neededEPNS_11TransactionE + 575
    6   QuartzCore                          0x00007fff27b2df3f _ZN2CA5Layer28layout_and_display_if_neededEPNS_11TransactionE + 65
    7   QuartzCore                          0x00007fff27a6d44c _ZN2CA7Context18commit_transactionEPNS_11TransactionEdPd + 496
    8   QuartzCore                          0x00007fff27aa4233 _ZN2CA11Transaction6commitEv + 783
    9   QuartzCore                          0x00007fff27aa53ef _ZN2CA11Transaction17observer_callbackEP19__CFRunLoopObservermPv + 79
    10  CoreFoundation                      0x00007fff2038f1f8 __CFRUNLOOP_IS_CALLING_OUT_TO_AN_OBSERVER_CALLBACK_FUNCTION__ + 23
    11  CoreFoundation                      0x00007fff20389a77 __CFRunLoopDoObservers + 547
    12  CoreFoundation                      0x00007fff2038a01a __CFRunLoopRun + 1113
    13  CoreFoundation                      0x00007fff203896d6 CFRunLoopRunSpecific + 567
    14  GraphicsServices                    0x00007fff2c257db3 GSEventRunModal + 139
    15  UIKitCore                           0x00007fff24696cf7 -[UIApplication _run] + 912
    16  UIKitCore                           0x00007fff2469bba8 UIApplicationMain + 101
    17  DebuggingConstraints                0x0000000100512a32 main + 114
    18  libdyld.dylib                       0x00007fff2025a3e9 start + 1
    19  ???                                 0x0000000000000003 0x0 + 3
```

Key Points To Remember

Forgetting to disable translation of the autoresizing mask into constraints

is a frequent reason for unsatisfiable constraint problems:

translatesAutoresizingMaskIntoConstraints = false

Interface Builder can help find a lot of common problems, but it cannot warn you about runtime problems.

Test your App on as many devices configurations as you can. Some problems only show up when you squeeze a view into the smallest iPhone screen or stretch it to fill the largest iPad Pro.

Ambiguous layouts happen when you're missing constraints or have conflicting priorities. Use the view debugger to find which views are ambiguous.

Add identifiers to your constraints and accessibility identifiers to your views to make them stand out in the logs.

When you can't use the view debugger the private layout debug methods (_autolayoutTrace, hasAmbiguousLayout) can help.

If your layout hangs and your App is consuming high CPU you may have created a layout loop. Add a launch argument to set a loop debugging threshold to help find the problem: -UIViewLayoutFeedbackLoopDebuggingThreshold 50

Size classes

Size classes
Adapting For Size

Before the size classes appeared developers needed to create at least 2 storyboards if they were developing for both iPhone and iPad.

In cases if an interface was particularly complicated developers could even create a storyboard for a separate orientation or device.

Now you can create an interface in one storyboard and adapt it to all screen sizes and orientations.

Size classes

Adapting For Size

Apple introduced the concept of adaptive user interfaces in iOS 8 using a combination of Auto Layout, size classes, and adaptive view controller presentations. The aim is to make it easier to develop universal interfaces that scale from the smallest iPhone to the largest iPad.

Building user interfaces that adjust to changes in screen size became even more critical when Apple added the multi-tasking slide over and split-screen modes to the iPad in iOS 9.

Size classes

Trait Collections

Before iOS 8 if you wanted to have a different user interface for the iPad version of an App you could check the userInterfaceIdiom of the device running your App:

```
if (UIDevice.currentDevice.userInterfaceIdiom == UIUserInterfaceIdiomPad)
```

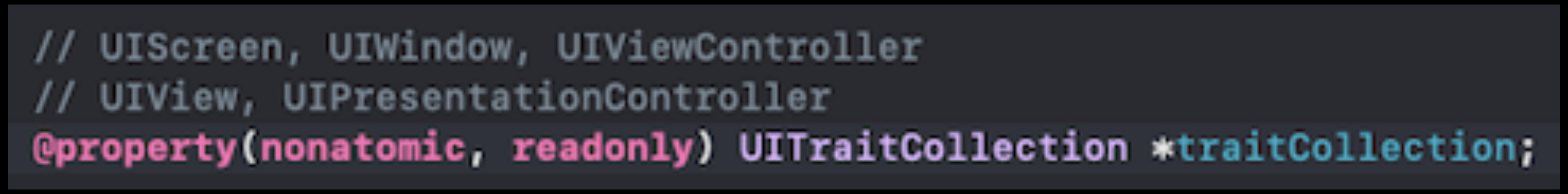
That was fine when Apps always ran full-screen on either an iPhone or iPad. We now have more devices, different screen resolutions and color gamuts and properties like dynamic type and split-screen modes that can change at runtime.

Size classes
Trait Collections

Starting in iOS 8 Apple introduced the idea of a trait collection to describe the environmental properties (traits) of a user interface.

The traitCollection is freighted with a considerable number of properties describing the environment.

You can check the trait collection of views, view controllers, and other classes that adopt the trait environment protocol (UITraitEnvironment) using their traitCollection property:



```
// UIScreen, UIWindow, UIViewController
// UIView, UIPresentationController
@property(nonatomic, readonly) UITraitCollection *traitCollection;
```

Size classes

Trait Collections

The traitCollection property is of type UITraitCollection and has the following trait properties:

horizontalSizeClass: UIUserInterfaceSizeClass. Possible values are unspecified, compact, regular. Default is unspecified.

verticalSizeClass: UIUserInterfaceSizeClass. Possible values are unspecified, compact, regular. Default is unspecified.

displayScale: CGFloat indicating the display scale where 0.0 is unspecified, 1.0 is non-Retina and 2.0 is retina.

displayGamut: UIDisplayGamut. Possible values are unspecified, SRGB and P3. Available since iOS 10.

userInterfaceStyle: UIUserInterfaceStyle. Possible values are unspecified, light, dark. Available since iOS 12.

userInterfaceIdiom: UIUserInterfaceIdiom. Possible values are unspecified, phone, pad, tv, carPlay, mac (iOS 14). Default is unspecified.

userInterfaceLevel: UIUserInterfaceLevel. Possible values are unspecified, base, elevated. Available since iOS 13.

Size classes
Trait Collections

layoutDirection: UITraitEnvironmentLayoutDirection. Possible values are unspecified, leftToRight and rightToLeft. Avail- able since iOS 10.

accessibilityContrast: UIAccessibilityContrast. Possible values are unspecified, normal, high. Available since iOS 13.

legibilityWeight: UILegibilityWeight. Possible values are unspecified, regular, bold. Available since iOS 13.

activeAppearance: UIUserInterfaceActiveAppearance. Possible values are unspecified, active, inactive. Available since iOS 14.

forceTouchCapability: UIForceTouchCapability. Possible values are unknown, available, unavailable. Note a user can disable force touch in the accessibility settings.

preferredContentSizeCategory: UIContentSizeCategory. Available since iOS 10.

Many traits have a default value of unspecified when the system has not set a value because the object is not yet in the view hierarchy.

Size classes

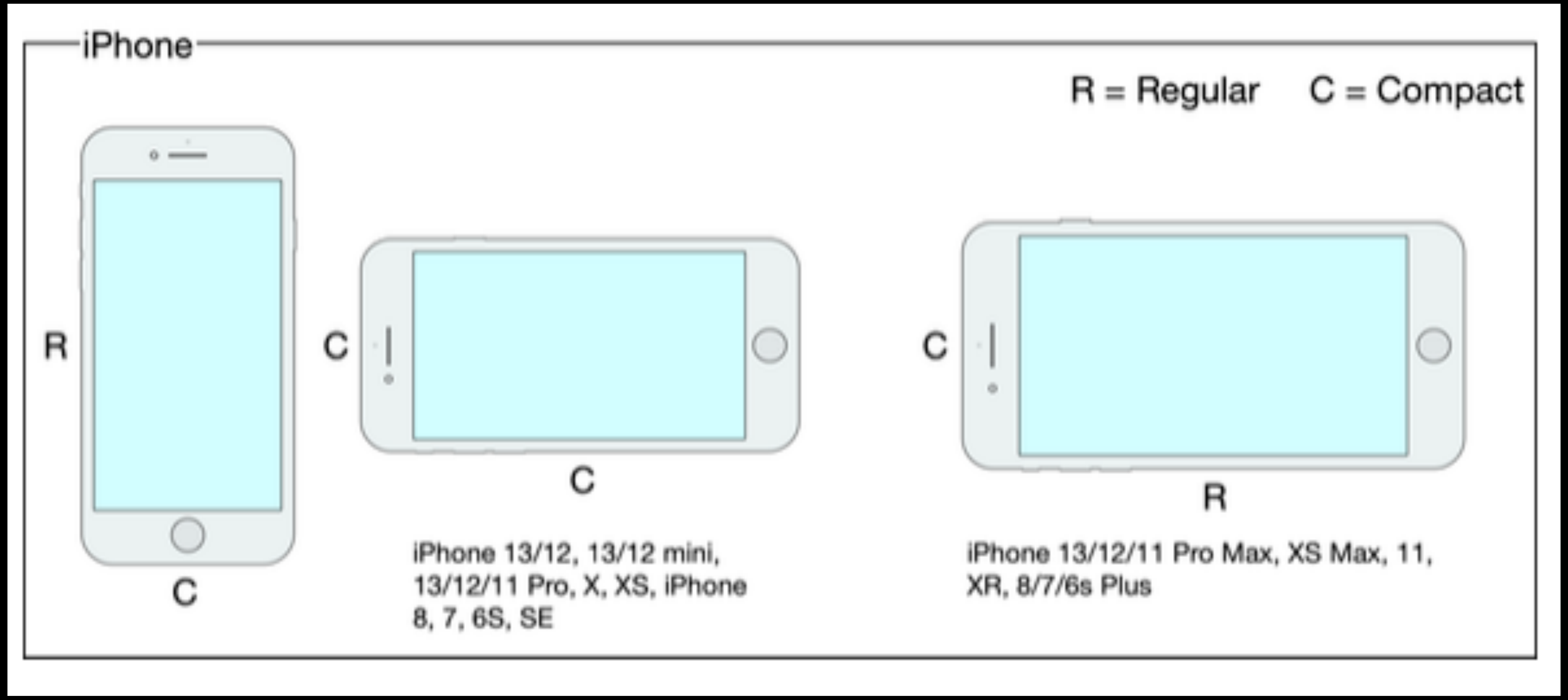
The two user interface traits you probably use the most when creating adaptive layouts are the horizontal and vertical size classes. There are two size classes that can apply to either the horizontal (width) or vertical (height) dimension of a user interface:

- regular: when the interface has lots of space.
- compact: when the interface has limited space.

Size classes

The regular iPhones all have a regular height and compact width in portrait but in landscape both dimensions are compact.

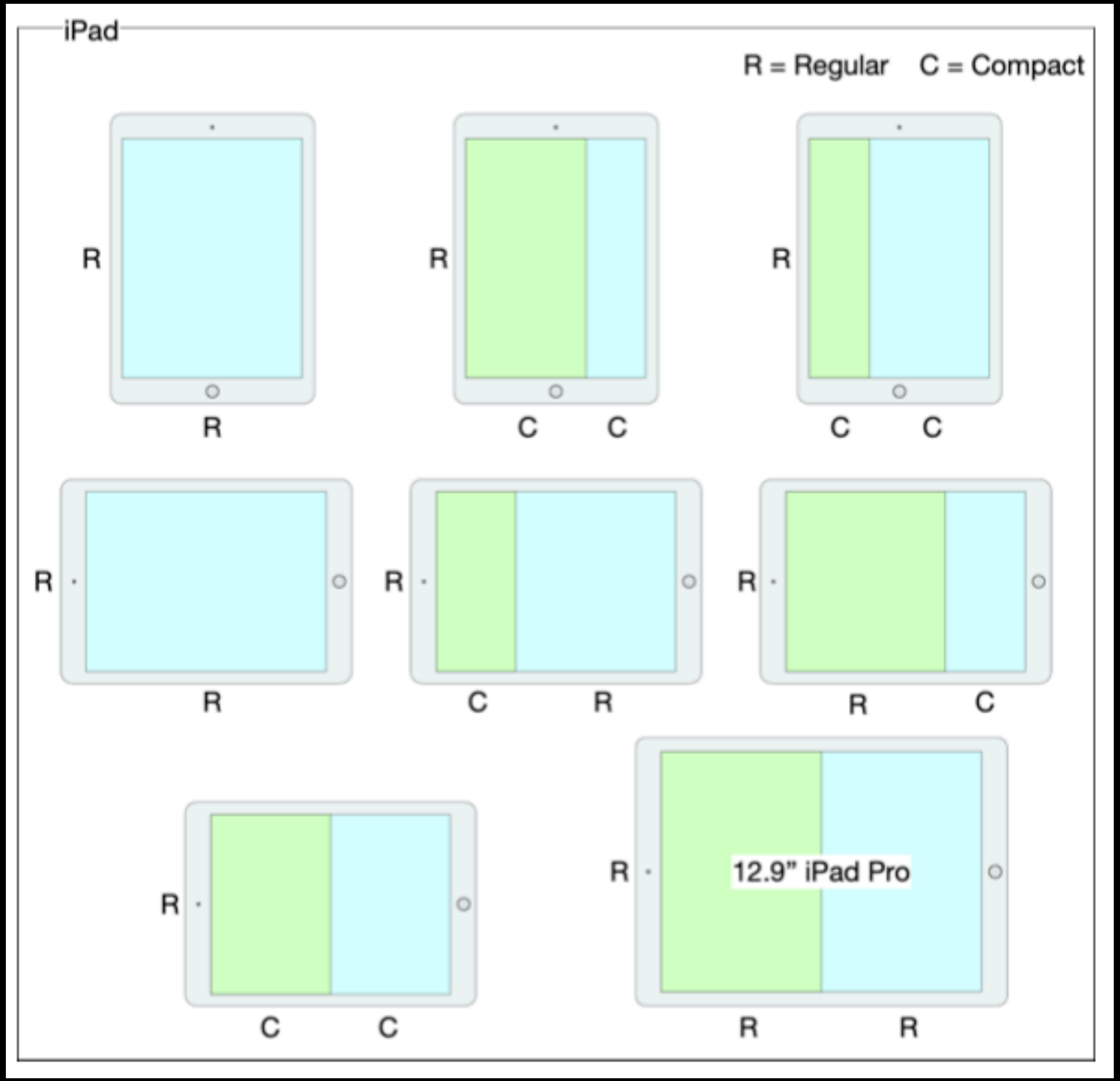
The various iPhone Plus and Max models, the XR and iPhone 11 differ in that they also have a regular width in landscape.


iPhone
R = Regular
C = Compact
R
C
C
C
iPhone 13/12, 13/12 mini, 13/12/11 Pro, X, XS, iPhone 8, 7, 6S, SE
C
R
iPhone 13/12/11 Pro Max, XS Max, 11, XR, 8/7/6s Plus

Size classes

The iPad has more possibilities due to the multitasking split views introduced in iOS 9 which split the screen to show two apps side-by-side each having 1/3, 1/2 or 2/3 of the screen width.

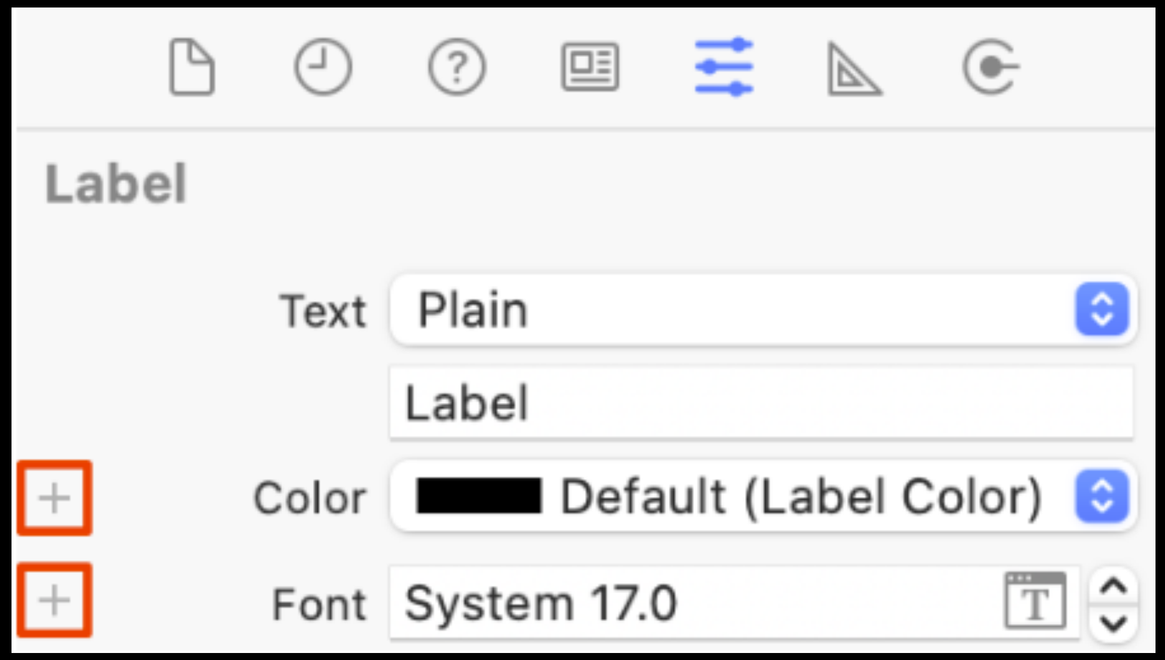
- A full-screen iPad application always has regular height and regular width size classes regardless of orientation or device.
- The height/vertical size class is always regular even with a split view.
- In portrait the apps have different widths (1/3 and 2/3), but both have a compact horizontal size class.
- In landscape the 2/3 width gives a regular horizontal size class.
- In landscape the 50:50 split view gives both apps a compact horizontal size class except for the larger 12.9” iPad Pro where both apps have a regular horizontal size class.



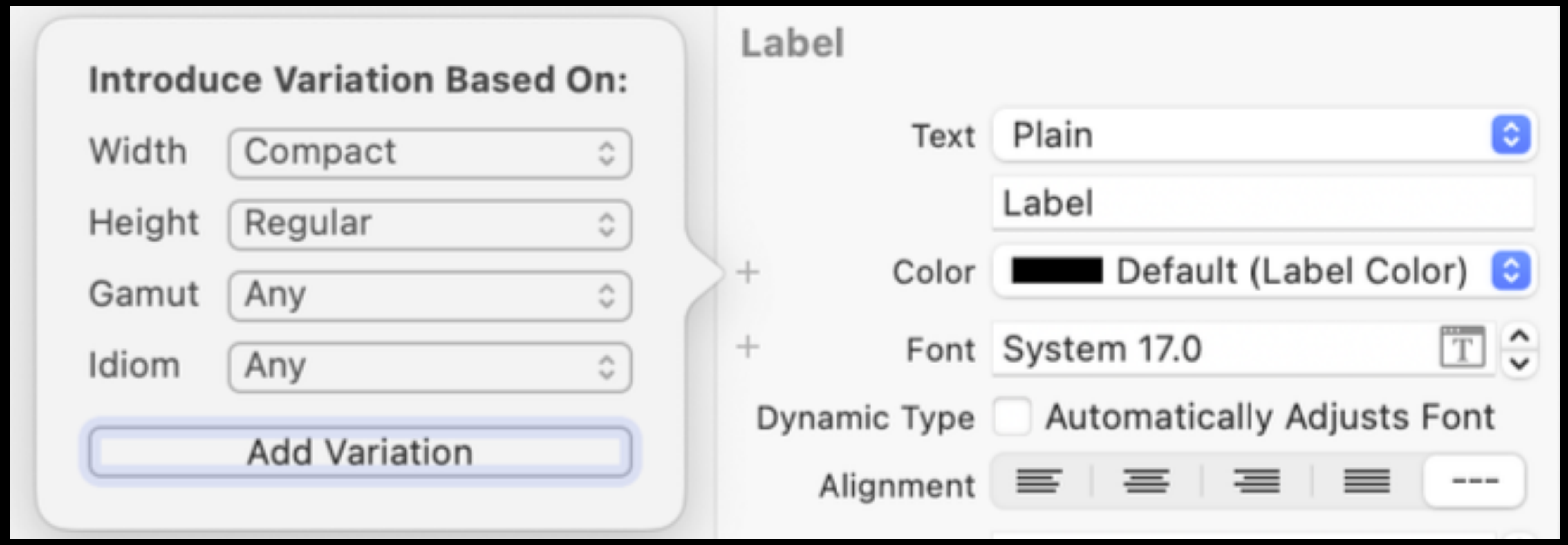
Using Size Classes With Interface Builder

Interface Builder has several ways to build layouts that adapt to the traits of the user interface. Use the attributes and size inspectors to create a trait variation for a property of a view or constraint. The properties you can adapt this way have a [+] symbol on the left in the inspector.

Furthermore, you can vary the property for a combination of width, height, color gamut or idiom (for Mac Catalyst apps).


Label

Text: Plain
Label
+ Color: Default (Label Color)
+ Font: System 17.0


Introduce Variation Based On:

Width: Compact
Height: Regular
Gamut: Any
Idiom: Any

Add Variation

Label

Text: Plain
Label
+ Color: Default (Label Color)
+ Font: System 17.0
Dynamic Type: Automatically Adjusts Font
Alignment

Using Size Classes With Interface Builder

Properties that you can vary based on traits with Interface Builder include:

- Whether to install a view or constraint.
- Whether to show or hide a view.
- The font, color, tint or background.
- Layout margins.
- The image file used by an image view.
- Stack view configuration (axis, alignment, distribution, spacing, baseline).

Adapting Properties For Size Classes

If you’ve tried building adaptive layouts with Xcode 12 you may remember the Vary for Traits button that allowed you to control which constraints are active for certain size classes.

Unfortunately, Apple has removed that approach in Xcode 13. Instead we need to inspect constraints and create variations on their installed property.

Using Traits In Code

You don’t have to use Interface Builder to build adaptive layouts. The UITraitEnvironment and UIContentContainer protocols give us two ways to react to trait changes in code.

Using Traits In Code

UITraitEnvironment Protocol

The UITraitEnvironment protocol is adopted by UIScreen, UIWindow, UIViewController, UIView and UIPresentationController. It defines a property you can use to get the trait collection of the object and a method you can override to be informed when the trait collection changes.

```
- (void)traitCollectionDidChange:(UITraitCollection *)previousTraitCollection;
```

If you override the method, you must first call super. In iOS 12 and earlier, UIKit calls the traitCollectionDidChange method after adding a view to the view hierarchy. The previousTraitCollection parameter is nil this first time and the current value is available in the traitCollection property.

Starting with iOS 13, UIKit sets the traits of a view when you create it, before you add it to the view hierarchy. UIKit guesses the likely traits for the view based on the context. When you add the view to the view hierarchy the actual traits are inherited from the parent. If UIKit has guessed correctly you don't get a call to traitCollectionDidChange.

Using Traits In Code

UITraitEnvironment Protocol

If you have relied on size changes to build adaptive layouts in iOS 12, you should review your code for iOS 13. UIKit now predicts the initial traits for a view so you cannot assume traitCollectionDidChange will be called when a view is first added to the view hierarchy.

As iOS 13 doesn’t guarantee traitCollectionDidChange will be called when our view is first loaded. We need to make sure our initial views state is correct. One way to do that is to configure it from viewDidLoad.

In iOS 12 and earlier, there is no promise that UIKit has correctly set the traits for the view at this point. However, we will get a call to traitCollectionDidChange when the view is added to the hierarchy so we can correct things.

In iOS 13, this configures the stack view based on the predicted traits. If UIKit guesses wrong, traitCollectionDidChange is called with the final size class. Either way we end up with a correctly configured layout.

Using Traits In Code

UITraitEnvironment Protocol

Where to perform any work involving traits:

- (void) viewDidLoad

Or in one of the layout methods:

- (void) viewWillLayoutSubviews (UIViewController)

- (void) layoutSubviews (UIView)

- (void) viewDidLayoutSubviews (UIViewController)

The trait collections are updated before layout occurs, so are current in any of the above methods. The only caveat with this approach is that the layout methods may be called multiple times during the life of the view controller or view so you should take steps to avoid repeating work. We set the layout state the first time layout is performed and then rely on traitCollectionDidChange for any future size class changes.

Using Traits In Code

UIContentContainer Protocol

The UIContentContainer protocol provides a second way to react not only to trait changes but also to size changes. It’s adopted by view controllers but not by views and provides two methods for responding to changes:

```
- (void)viewWillTransitionToSize:(CGSize)size
withTransitionCoordinator: id<UIViewControllerTransitionCoordinator>)coordinator;
```

```
- (void)willTransitionToTraitCollection:(UITraitCollection *)newCollection
withTransitionCoordinator: id<UIViewControllerTransitionCoordinator>)coordinator;
```

The first of these methods is called before changing the size of a view controller’s view. Its first parameter is the new size of the view. The view has not yet changed size so you can still get its old size if needed.

The second method is called before changing the trait collection. Its first parameter is the new trait collection. The old trait collection is available in the traitCollection property of the view controller. If you override either method, you should call super in your implementation.

Using Traits In Code

UIContentContainer Protocol

How do these methods compare to the UITraitEnvironment method traitCollectionDidChange?

• Neither of the UIContentContainer methods are called when a view controller first adds its view to the view hierarchy. Similar to the way traitCollectionDidChange works in iOS 13 you cannot rely n these methods to do initial view and constraint setup.

• The UIContentContainer methods are only available in the view controller whereas the traitCollectionDidChange method can be used both in the view controller and in any custom subclass of UIView.

• You can use the coordinator object of the UIContentContainer methods when you want to run animations alongside the view controller transition animation.

Using Traits In Code

UIContentContainer Protocol

When a view controller is about to transition to a new size or trait collection the sequence of events is as follows:

1. During the setup of the transition the methods are called in the following order:

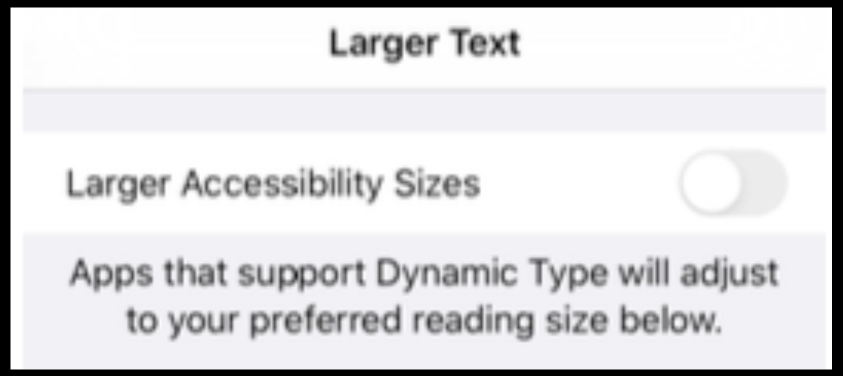
```
// UIContentContainer - trait will change
- (void)willTransitionToTraitCollection:(UITraitCollection *)newCollection
withTransitionCoordinator: id<UIViewControllerTransitionCoordinator>)coordinator;
// UIContentContainer - size will change
- (void)viewWillTransitionToSize:(CGSize)size
withTransitionCoordinator: id<UIViewControllerTransitionCoordinator>)coordinator;


// UITraitEnvironment - trait changed

- (void)traitCollectionDidChange:(UITraitCollection *)previousTraitCollection;
```

2. During the animation of the transition, you use the coordinator object to run your animations alongside the system animations.

3. Once the animation completes, use the completion handler of your animations to perform any cleanup.

Adapting For Dynamic Type Size

We can do more than adapt to the size class or size of the view.


Larger Text

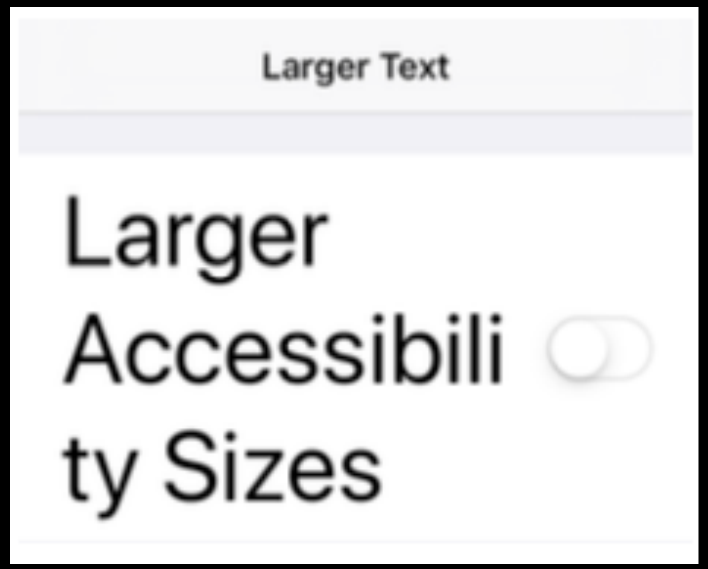
Larger Accessibility Sizes

Apps that support Dynamic Type will adjust to your preferred reading size below.

Here's how a text looks on an iPhone 8 at the default text size and

at the larger accessibility sizes keeping the label and switch arranged

horizontally becomes cramped especially on small devices like the iPhone SE.


Larger Text

Larger Accessibili ty Sizes

Adapting For Dynamic Type Size

To give the text as much space as possible we can move the switch down below the label when the user selects one of the larger accessibility sizes.

We could observe UIContentSizeCategoryDidChangeNotification, but there’s a better way using trait collections.

Since iOS 10 the UITraitCollection class has a property for the preferred content size category.

Using Traits With The Asset Catalog

The Asset Catalog allows you to create asset variations for a variety of different device traits.

Device Idiom Device types such as iPhone, iPad, Watch, Mac.

Appearance Light or dark interface style. Normal or high contrast.

Display Scale Pixel density of the screen.

Color Gamut: Use different images for sRGB and P3 color spaces.

Text Direction: Image variations for left-to-right and right-to-left text directions.

Width and Height Class: Image variations for different combinations of horizontal and vertical size class.

Memory: Variations based on the memory capacity of the device.

Graphics: Variations based on the graphics capability of the device.

Screen Width: Apple Watch screen width variations (38mm and 42mm).

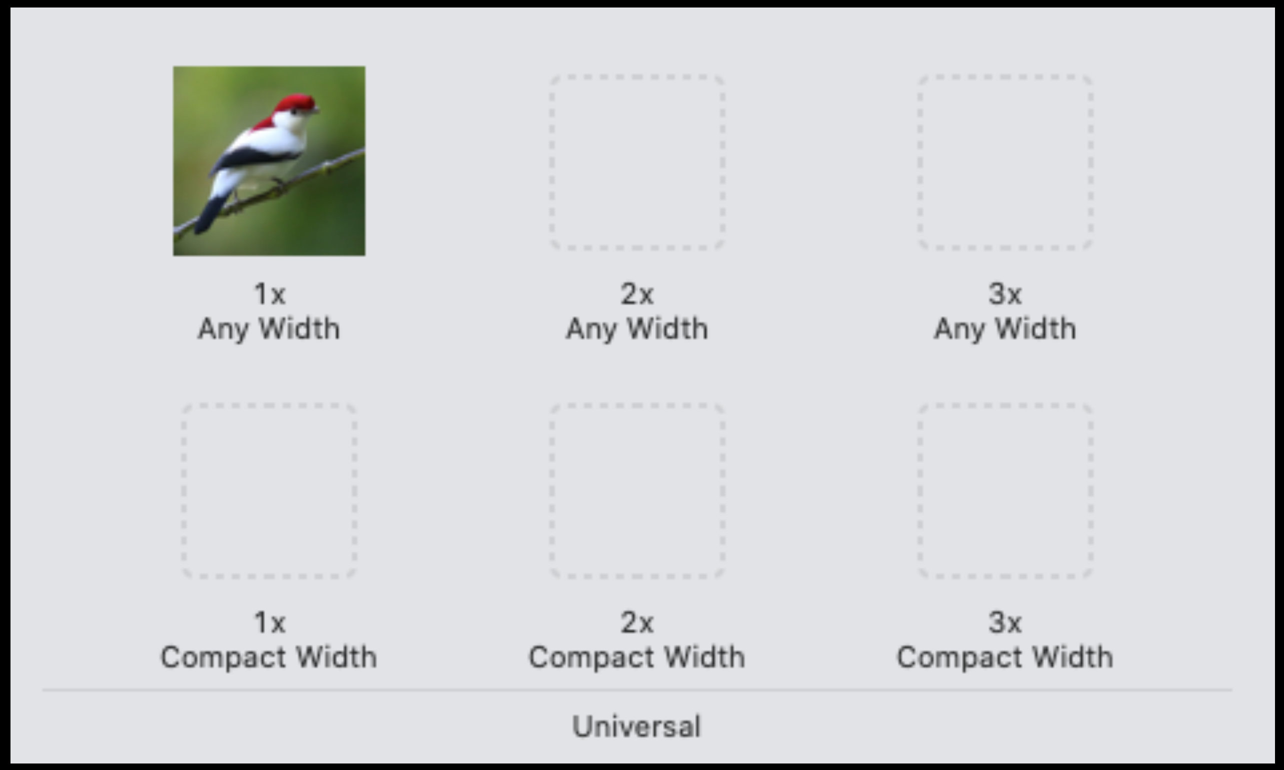
Using Traits With The Asset Catalog

As you create new variations, additional placeholders appear in the asset catalog.

Drag the asset you want to apply for a variation onto the placeholder.

For example, selecting “Any & Compact” for the height class adds a “Compact Height” variation to the catalog that you can use to add an image that better fits a compact height user interface.

You don’t need any code changes to use the image variations. The image view uses the “Compact Height” image when the vertical size class is compact.



Variable Width Strings

You can use size classes to adapt the font size of text but how about changing the text based on the available space? This can sometimes help when you’re using a fixed size font (I don’t recommend doing this with dynamic type). When you have limited space, you can use a shorter version of the text rather than shrinking the font size.

The localization system allows us to build a dictionary of variable width strings for this purpose. Both UILabel and UIButton are aware of variable width strings and automatically select the best string variation from the dictionary for the available screen width.

Key Points To Remember

1. You get the horizontal and vertical size classes of a view or view controller from its traitCollection property. Prefer designing your layout for the size class over the device idiom (iPhone or iPad) or orientation (portrait or landscape).

2. Don’t forget about managing the keyboard in iPad split-view modes. The keyboard can cover your App even if you don’t accept text input.

3. Adapt your layouts to size class changes using Interface Builder or by overriding traitCollectionDidChange in your view or view controller.

4. A stack view saves you work when you want to switch between horizontal and vertical layouts or when you want to show or hide views selectively.

5. To animate changes to your layouts when the size class or view size changes use the UIContentContainer methods in your view controller.

6. Don’t forget that you can add size class specific images to the asset catalog.

7. If you’re struggling to squeeze text with a fixed font into the available space try using localization strings dictionaries to create variations for different widths.

8. Horizontal and vertical size classes give you a crude measure of how much screen space is available. When you need finer control create layout variations based on the available space. Use the content container methods to react to size changes in your view controller.

References

1. Apple Auto Layout Guide

2. Advanced Auto Layout Toolbox