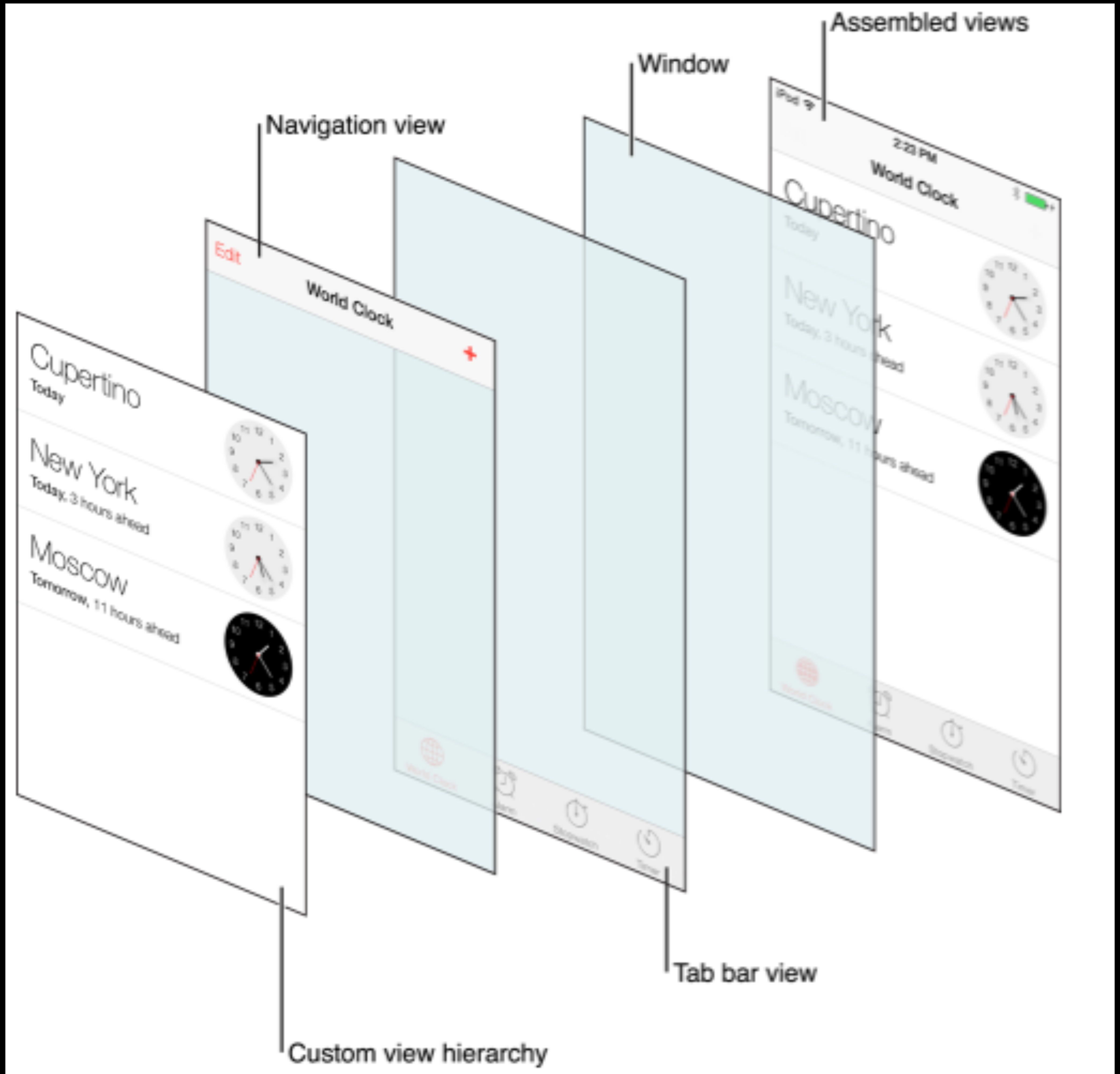
UIKit

About Views

Views are the building blocks for constructing your user interface.

Rather than using **one view** to present your content, you are more likely to **use several views**, ranging from simple buttons and text labels to more complex views such as table views, picker views and scroll views



About Views

Views allow users to:

- Experience app content
- Navigate within an app

Each **views** represents a particular portion of your user interface and is generally optimised for a specific type os content.

Views are implemented in the **UIView.** To add a subview to another view, you use the **addSubview(_:)** method.

UIView

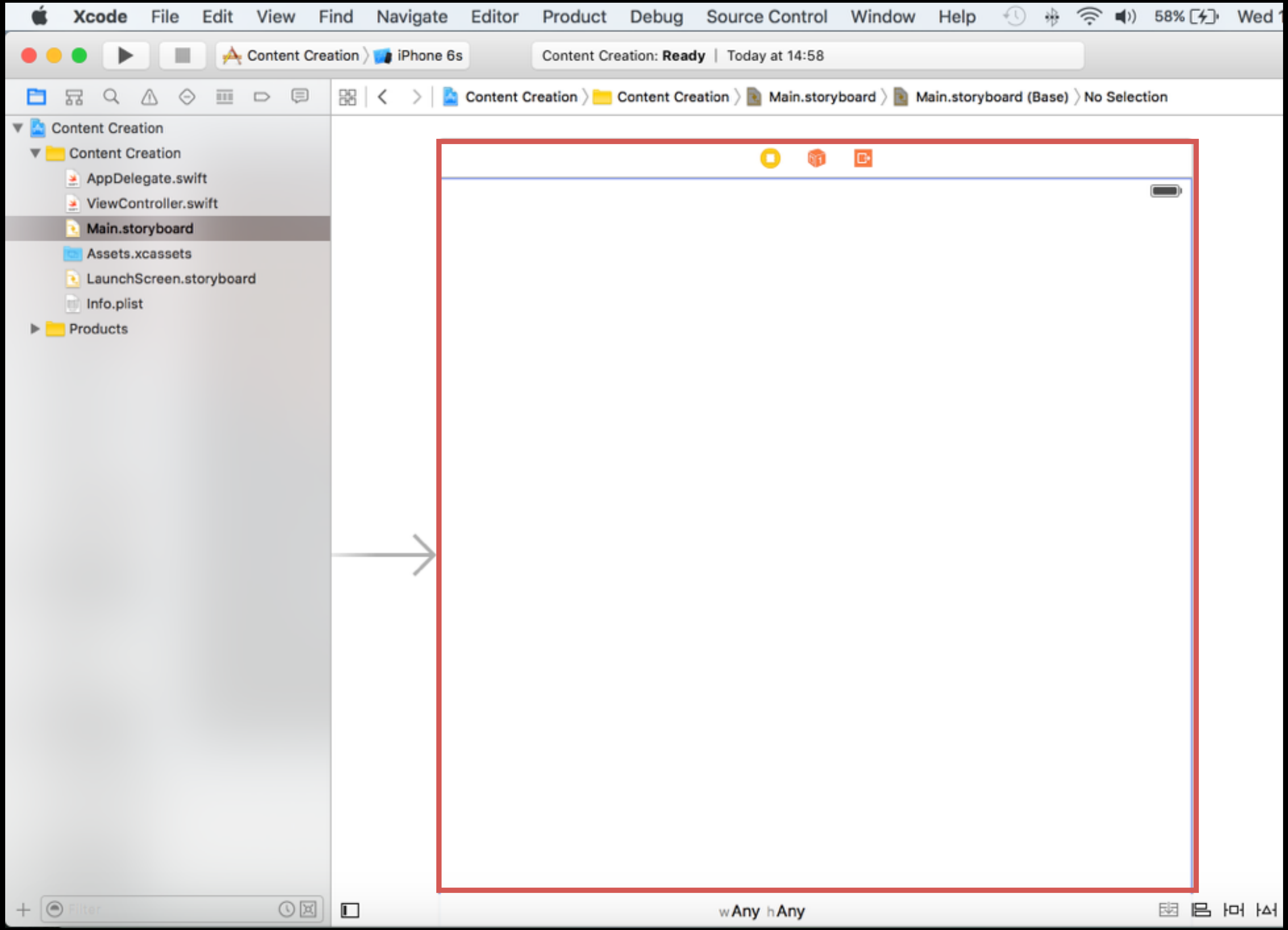
https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/UIKitUICatalog/index.html#//apple_ref/doc/uid/TP40012857-UIView-SW1

https://developer.apple.com/reference/uikit/uiview

UIView

The **UIView** class defines a rectangular area on the screen and the interfaces for managing the content in that area.

A **view** object handles the rendering of any content in its area and also handles any interaction with that content.



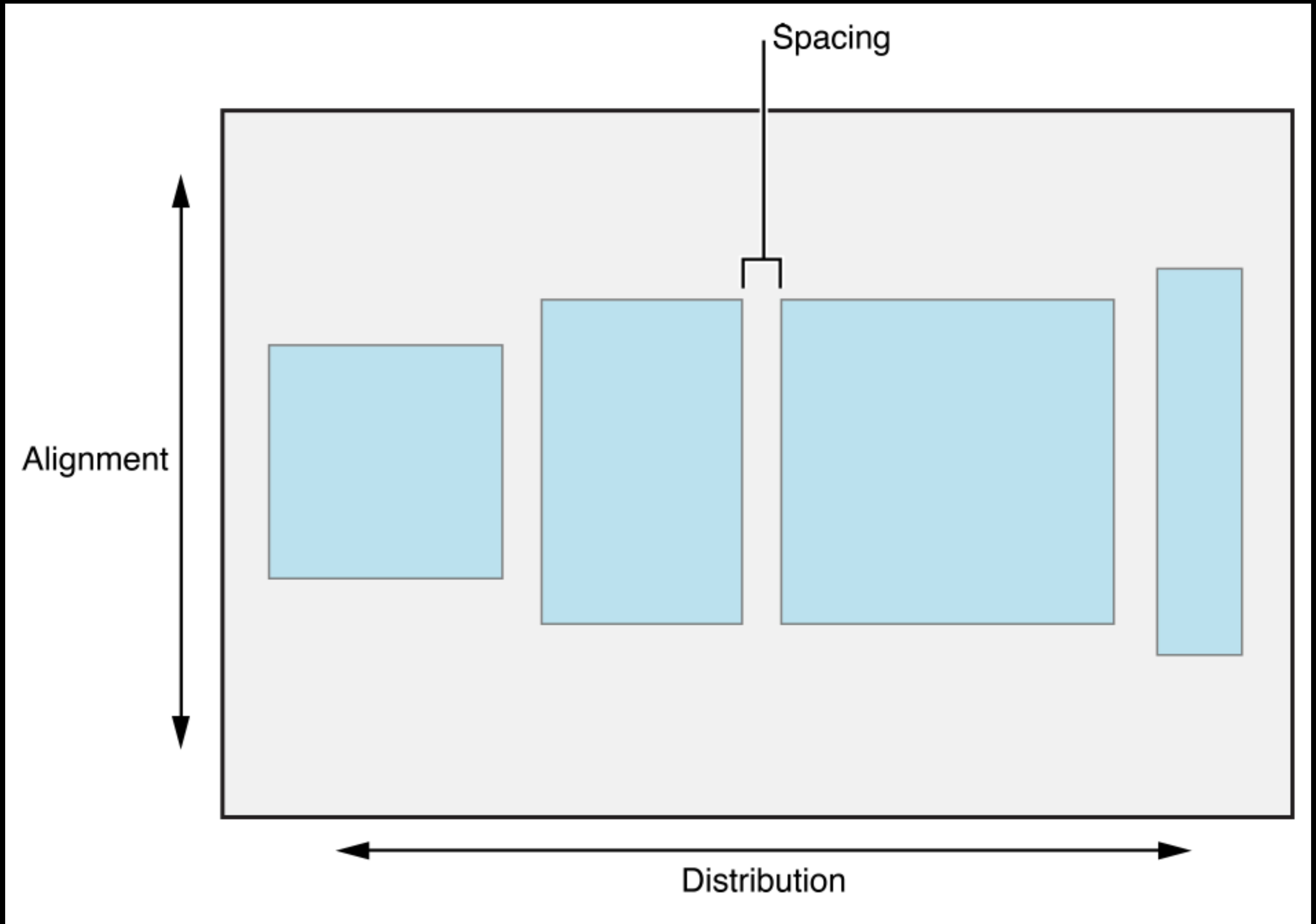
UIStackView

https://developer.apple.com/reference/uikit/uistackview

UIStackView

Provides a streamlined interface for laying out a collection of **views** in either a **column** or a **row**.

Stack views let you leverage the power of **Auto Layout,** creating user interfaces that can dynamically adapt to the device's orientation, screen size, and any changes in the available space.



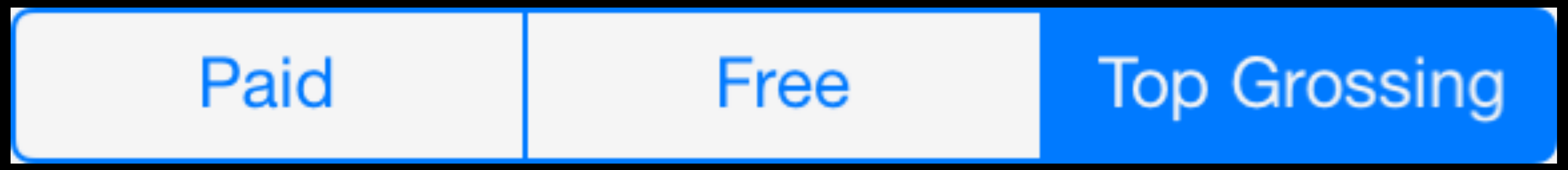
UISegmentedControl

https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/UIKitUICatalog/UISegmentedControl.html#//apple_ref/doc/uid/TP40012857-UISegmentedControl

https://developer.apple.com/reference/uikit/uisegmentedcontrol

UISegmentedControl

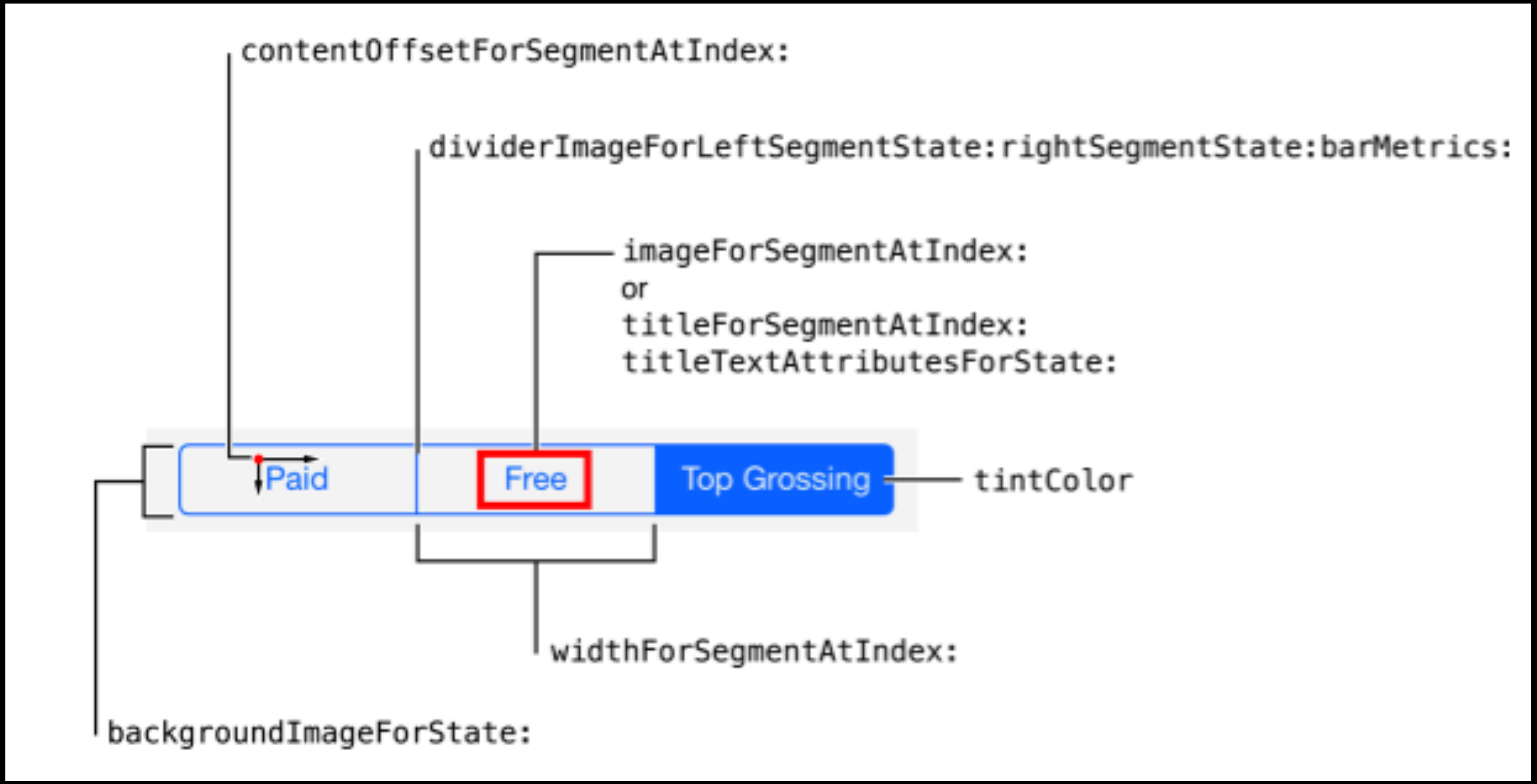
The **UISegmentedControl** object is a horizontal control made of multiple segmentes, each segment functioning as a discrete button.



A segmented controls allow **users** to **interact** with a compact group of a number of control.

UISegmentedControl

You can customize the appearance of a **segmented control** by setting the properties.

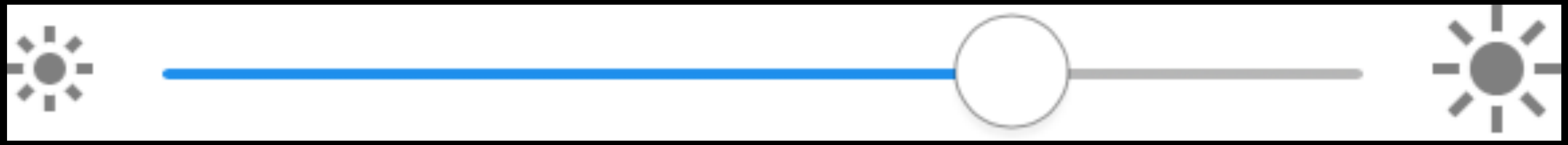


UISlider

https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/UIKitUICatalog/UISlider.html#//apple_ref/doc/uid/TP40012857-UISlider-SW1

Sliders

Sliders enable users to interactively modify some adjustable value in an app, such as speaker volume or screen brightness.

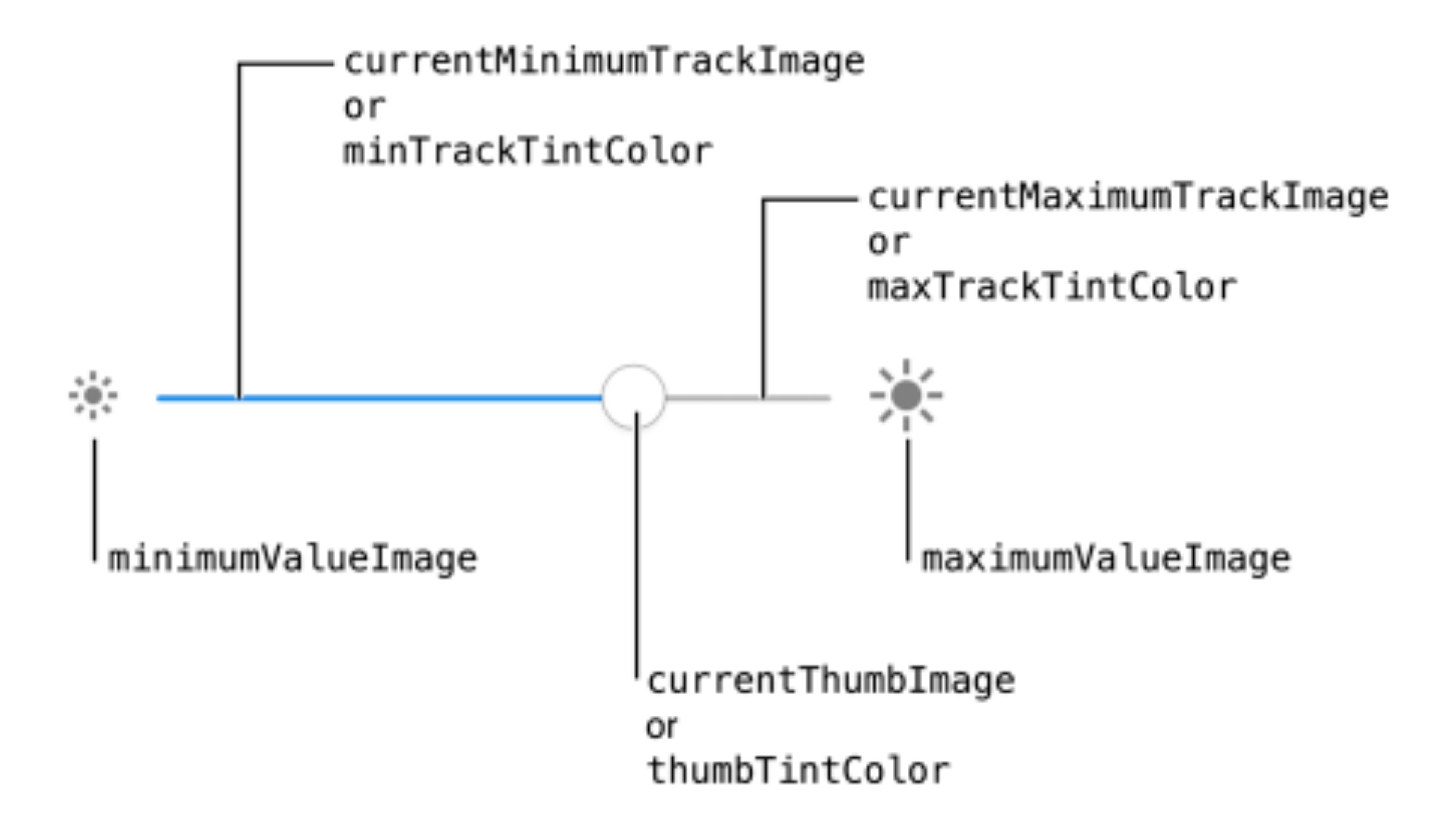


Sliders allow users to:

- Make smooth and continuous adjustments to a value
- Have relative control over a value within a range
- Set a value with a single simple gesture

Sliders

You can **customize** the appearance of a slider by setting the properties as showed by picture.



UIImage

UIImage

A **UIImage** object manages image data in your app. You use image objects to represent image data of all image formats supported by the underlying platform.

Image objects are immutable, so you always create them from existing image data, such as an image file on disk or programmatically created image data

UIImage

The **UIImage** object support all platform-native image formats

Is recommended the use of **PNG** or **JPEG** files

Use **image** objects to:

- Assign an **image** to a display the image in your interface
- Customize system controls such as buttons, sliders, and segmented controls
- Draw an image directly into a view or other graphics context
- Pass an image to other APIs that might require image data

UIImageView

https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/UIKitUICatalog/UIImageView.html#//apple_ref/doc/uid/TP40012857-UIImageView-SW1

https://developer.apple.com/reference/uikit/uiimageview

UIImageView

A **UIImageView** object displays a single image or a sequence of animated images in your interface.

Image views allow users to view images within an app

Image views let you efficiently draw any image (JPEG,TIFF, PNG, bmp, ico, cur and xbm formats) that can be specified using a **UIImage** object.

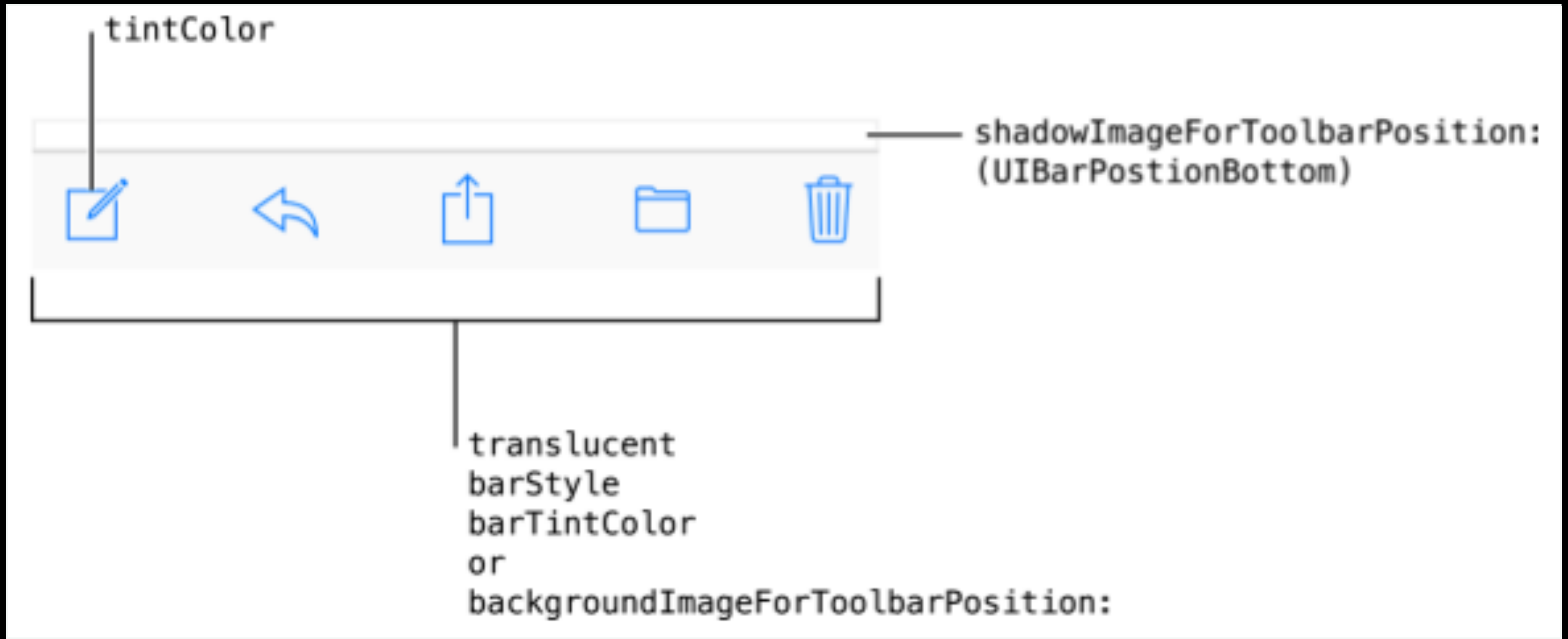
UIToolbar

https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/UIKitUICatalog/UIToolbar.html

UIToolbar

A **toolbar** is a control that displays one or more buttons, called **toolbar items**.

Usually appears at the bottom of a screen. Is often used in conjunction with a **navigation controller**, which manages both the **navigation bar** and the **toolbar**


tintColor
shadowImageForToolbarPosition:
(UIBarPostionBottom)
translucent
barStyle
barTintColor
or
backgroundImageForToolbarPosition:

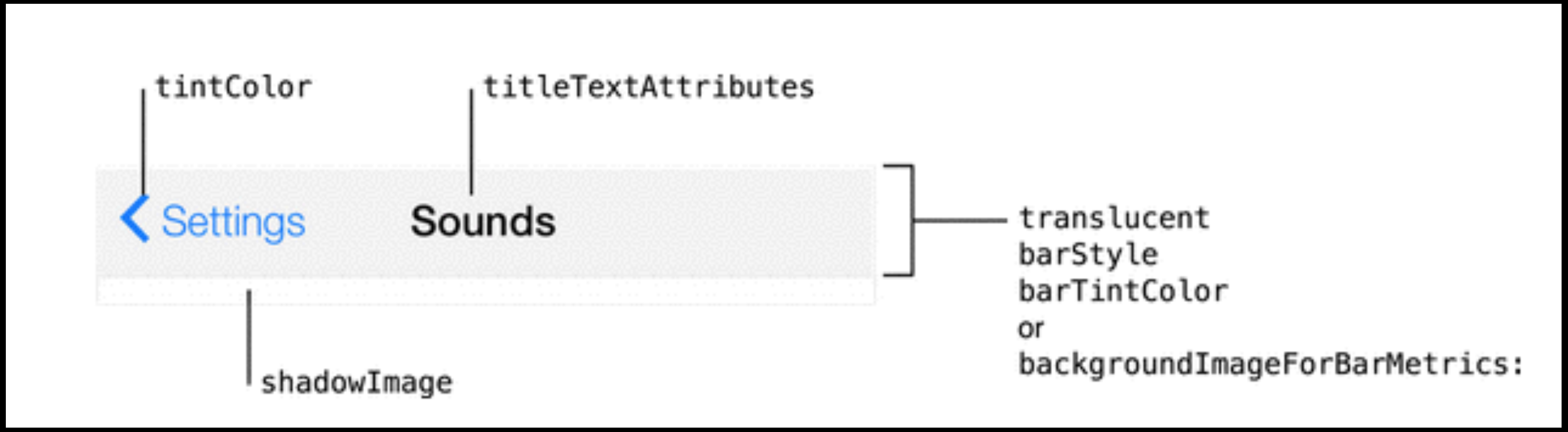
UINavigationBars

UINavigationBars

Navigation bars allow you to present your app's content in an organised and intuitive way.

Is displayed at the top of the screen, and contains buttons for navigation through a hierarchy of screens.

Generally has a **back button**, a **title**, and a **right button**.



UITextView

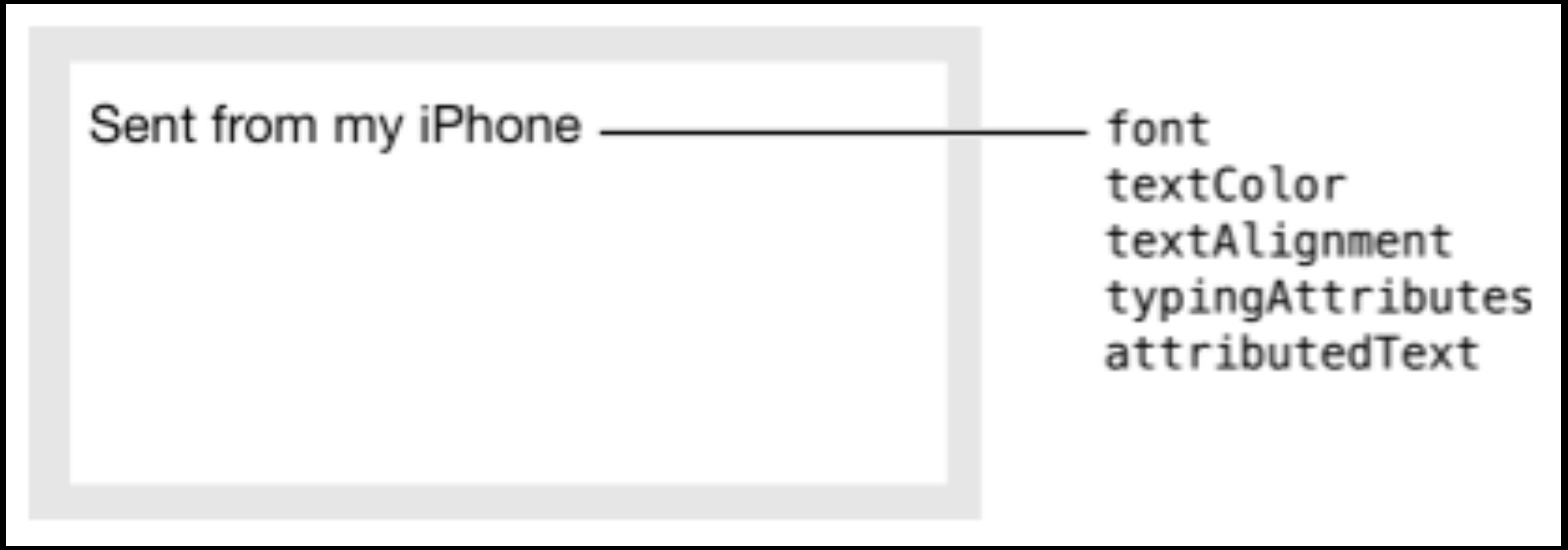
https://developer.apple.com/reference/uikit/uitextview

https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/UIKitUICatalog/UITextView.html#//apple_ref/doc/uid/TP40012857-UITextView-SW1

UITextView

A **UITextView** accepts and displays multiple lines of text. Support scrolling and text editing., multiline text region.

Typically is used to display a large amount of text, such as the body of an email message.



UIBarButtonItem

UIBarButtonItem

A bar button item is a button specialized for placement on a **UIToolbar** or **UINavigationBar** object.

It inherits basic button behavior from its abstract superclass, **UIBarItem.**

Defines additional initialisation methods and properties for use on **toolbars** and **navigation bars**.

Auto Layout

https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/AutolayoutPG/

Understanding Auto Layout

Auto Layout dynamically calculates the size and position of all views in your view hierarchy, based on constraints placed on those views.

The constraint-based approach to design allows you to build user interfaces that dynamically respond to both internal and external changes.

Understanding Auto Layout

Example:

You can constrain a button so that it is horizontally centered with an **Image view** and so that the button's top edge always remains 8 points below the image's bottom.

If the image view's size or position **changes**, the button's position **automatically adjusts** to match.

Understanding Auto Layout

External Changes

Occurs when the size of shape of your superview changes. With each changes, you must update the layout of your view hierarchy to best use the available space.

Understanding Auto Layout

External Changes - Common Sources

- The user resizes the window (OS X)
- The user enters or leaves Split View on an iPad (iOS)
- The devices rotates (iOS)
- The active call and audio recording bars appers or disappear (iOS)
- Support different size classes
- Support different screen sizes

Most of these changes can occur at **runtime**, and they require a **dynamic response** from your app. Others, like support for different screen sizes, represent the app adapting to different environments.

Understanding Auto Layout

Internal Changes

Occurs when the size of the view or controls in your user interface change.

Understanding Auto Layout

Internal Changes - Common Sources

- The content displayed by the app changes
- The app supports internalization
- The app support Dynamic Type (iOS)

When your app's **content changes**, the new content may require a **different layout** than the old. This commonly occurs in apps that display text or images.

Auto Layout *versus* Frame-Based Layout

There are **three** main approaches to laying out a user interface:

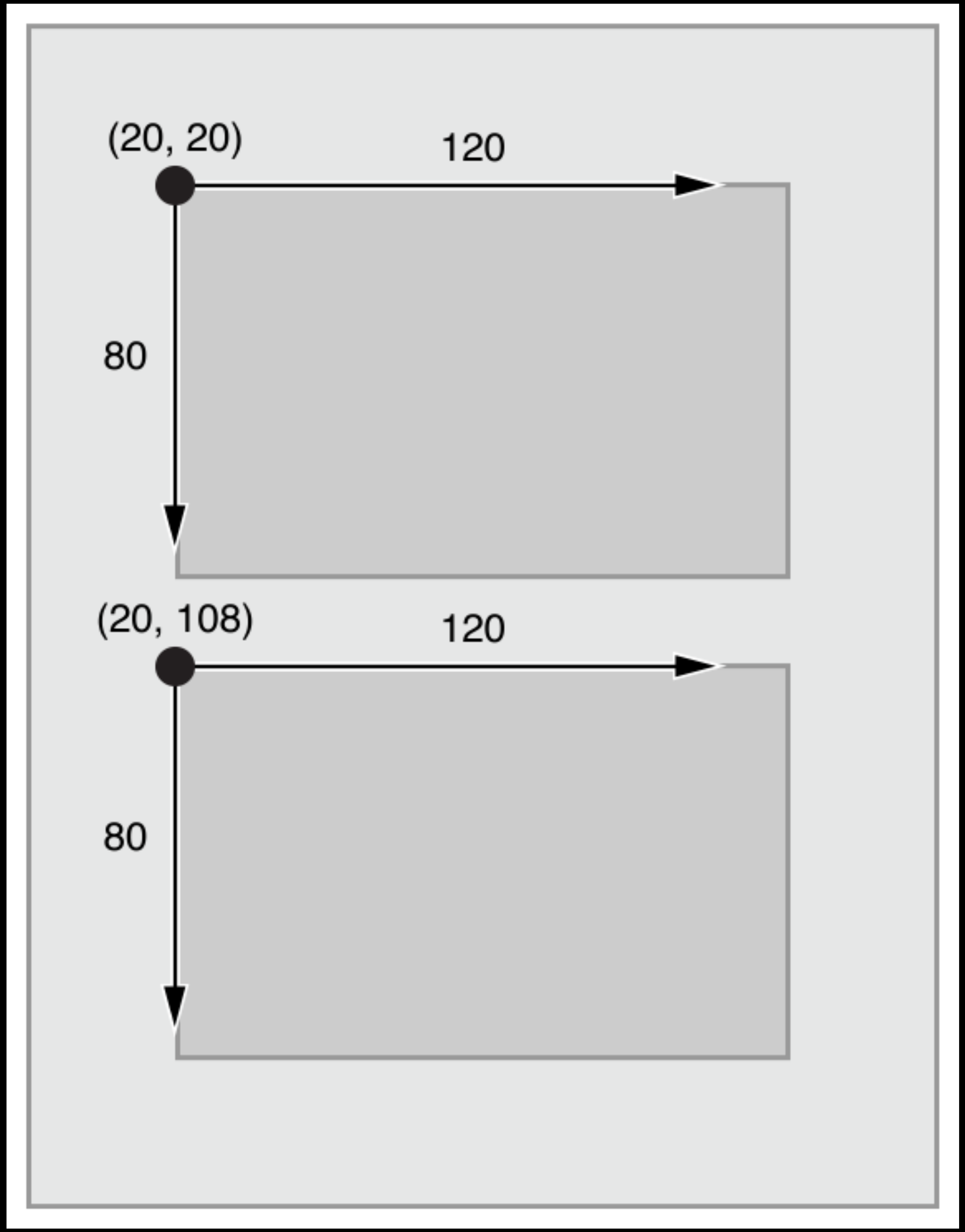
- Programmatically
- Autoresizing Masks
- Auto Layout

Auto Layout *versus* Frame-Based Layout

Programmatically

Setting the frame for each view in a view hierarchy.

The frame defined the view's origin, height, and width in the superview's coordinate system.



Auto Layout *versus* Frame-Based Layout

Autoresizing Masks

Support a relatively small subset of possible layouts

Define how a view’s frame changes when its superview’s frame changes.

For complex user interfaces, you typically need to augment the autoresizing masks with your own programmatic changes.

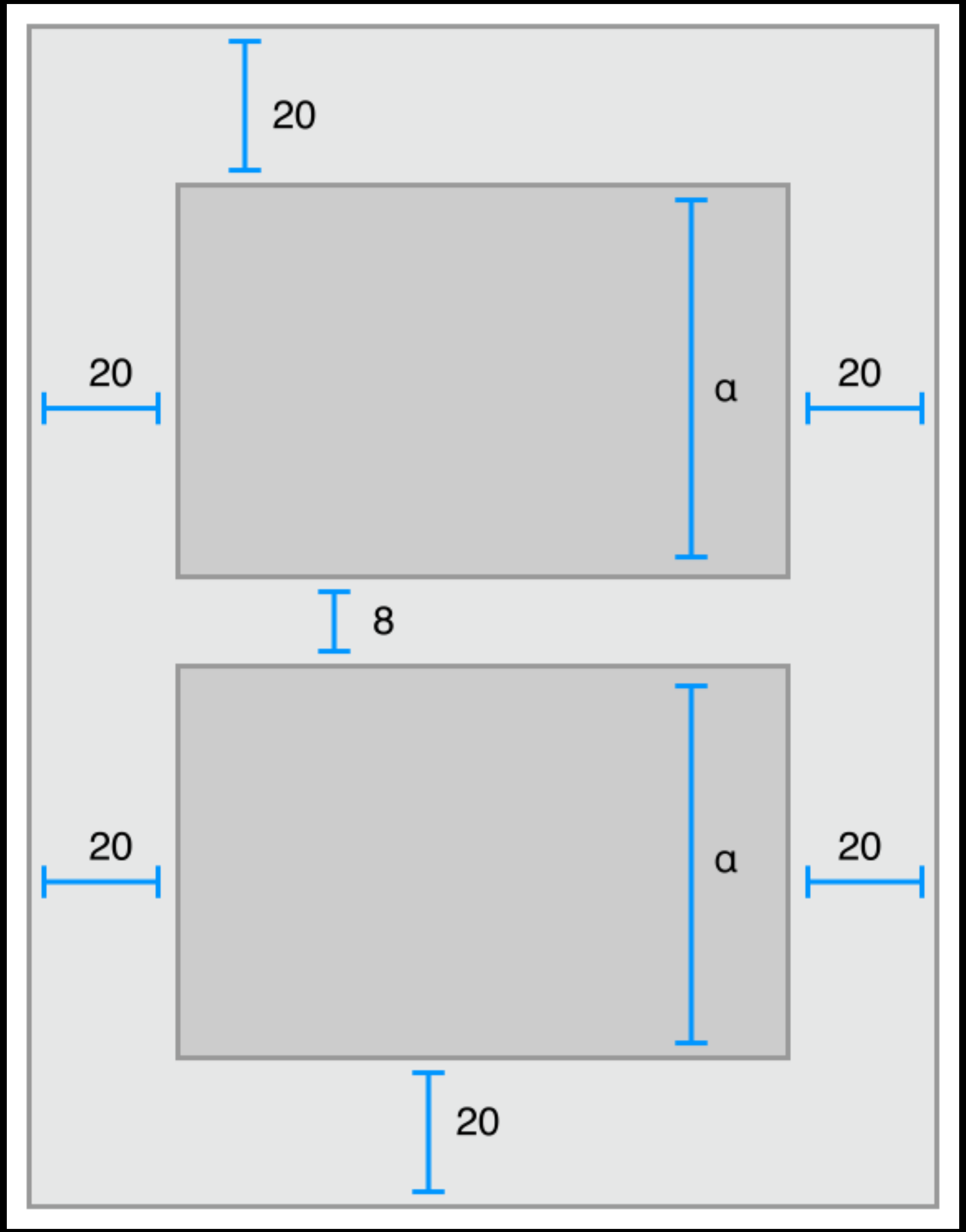
Adapt **only to external changes**. They do not support **internal changes**.

Auto Layout *versus* Frame-Based Layout

Auto Layout

Defines your user interface using a series of **Contraints**, that typically represent a relationship between two views.

Auto Layout calculates the size and location os each view based on these constraints.

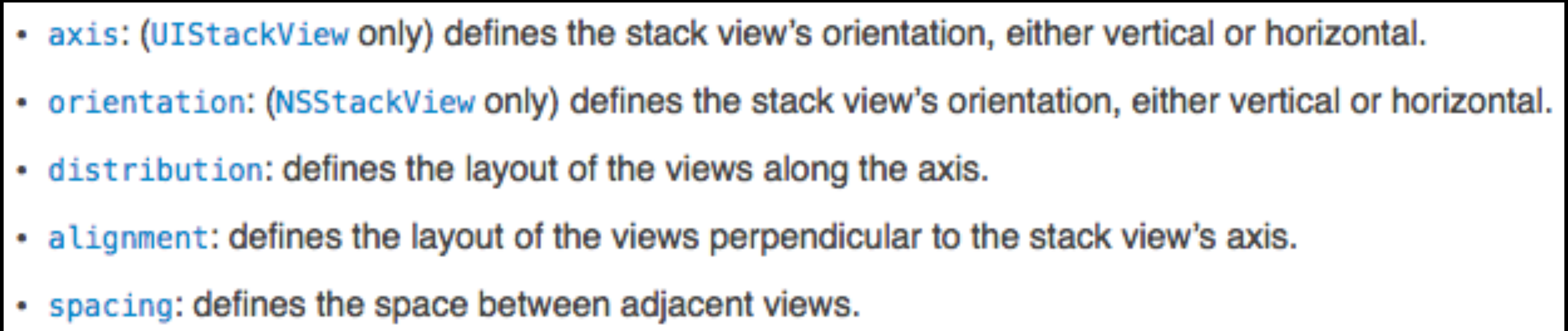


This produce **layouts** that **dynamically** respond to both **internal** and **external** changes

Auto Layout *without* Constraints

Stack views provide an easy way to leverage the power of Auto Layout without introducing the complexity of constraints and defines a row or column of user interface elements.

Stack views stack view arranges these elements based on its properties:

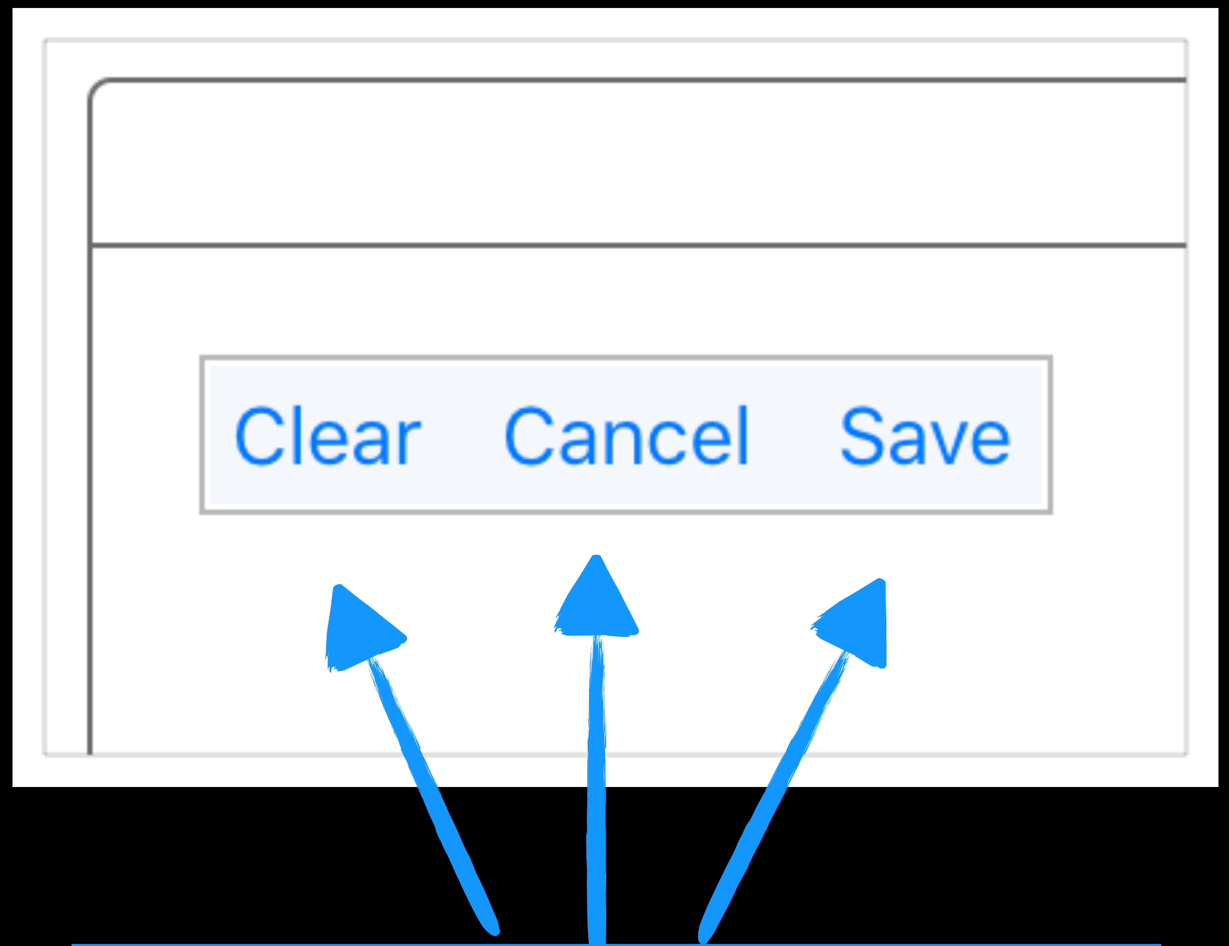


- `axis`: (`UIStackView` only) defines the stack view's orientation, either vertical or horizontal.
- `orientation`: (`NSStackView` only) defines the stack view's orientation, either vertical or horizontal.
- `distribution`: defines the layout of the views along the axis.
- `alignment`: defines the layout of the views perpendicular to the stack view's axis.
- `spacing`: defines the space between adjacent views.

Auto Layout *without* Constraints

To use a **Stack views** in Interface Builder, drag either a vertical or horizontal stack view onto the canvas. Then drag out the content and drop it into the stack.

To further fine-tune the layout, you can modify the stack view's properties using the Attributes Inspector.

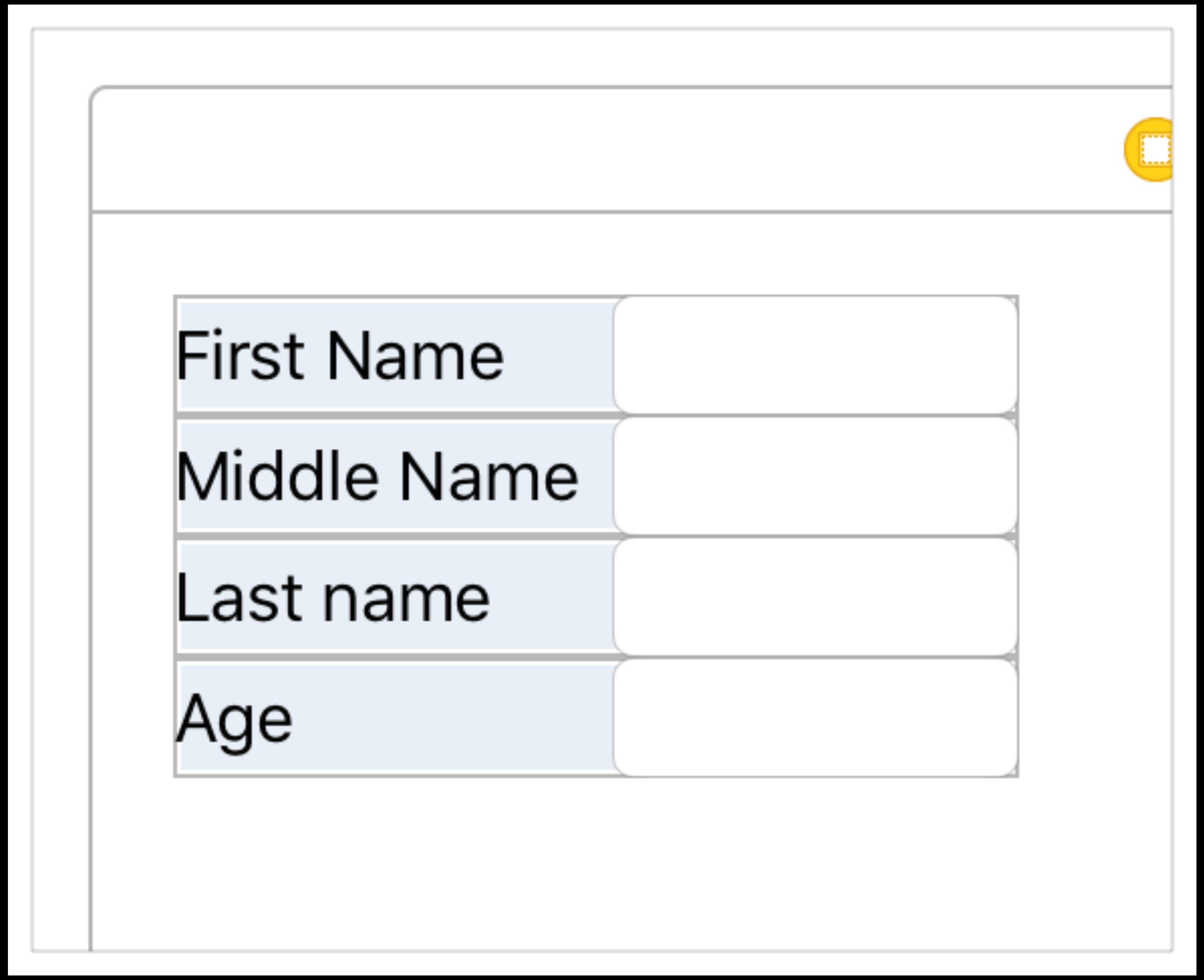


8-point spacing and a Fills Equally distribution

Auto Layout *without* Constraints

The **Stack views** also bases its layout on the arranged view’s content-hugging and compression-resistance priorities.

You can modify these using the Size Inspector and you can nest stack views inside other stack view's to build more complex layouts.



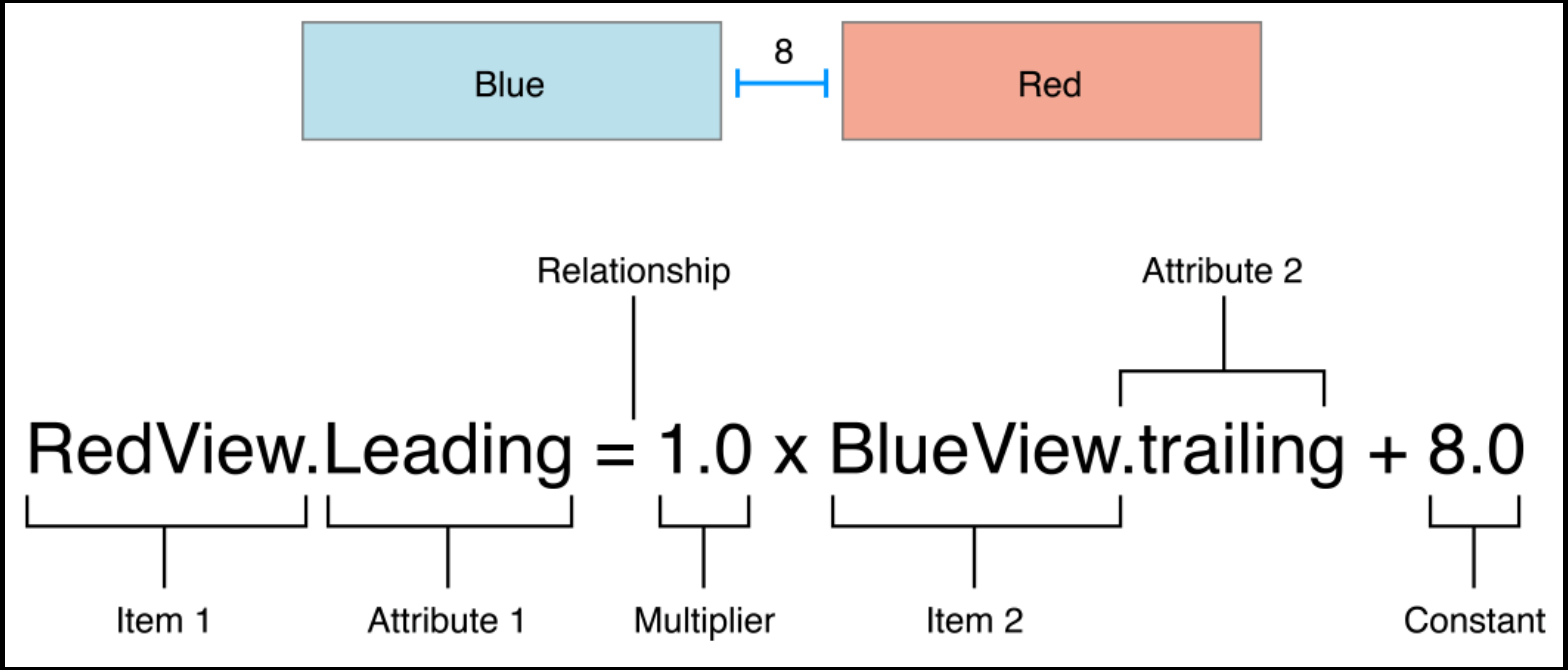
Anatomy of a Constant

The **layout** of your view is defined as a series of linear equations.

Each **constraints** represents a single equation.

Your **goal** is to declare a series of **equations** that has one and only one **possible solution**.

Anatomy of a Constant



A sample equation

Auto Layout Attributes

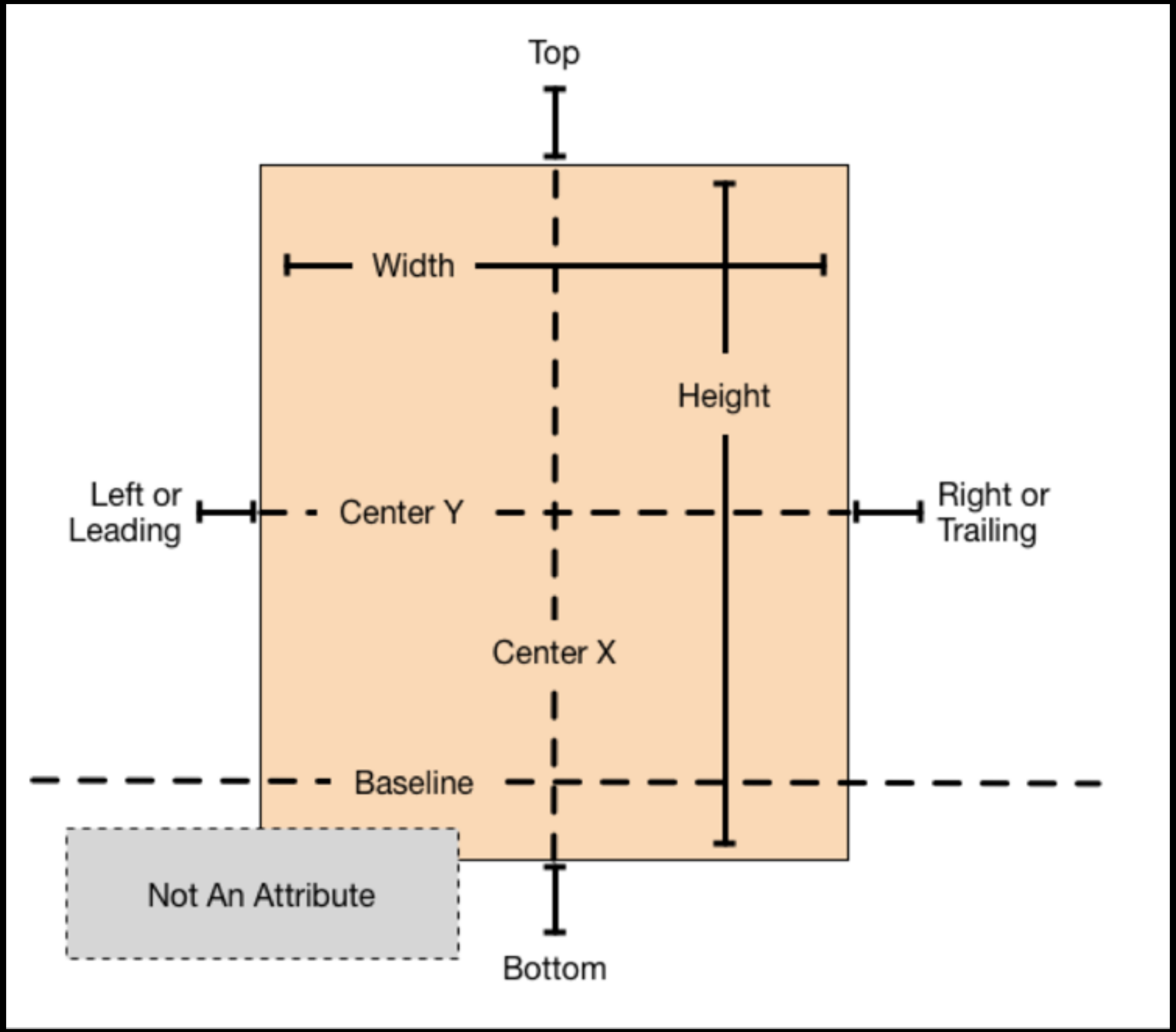
Defines a features that can be constrained.

Includes the four edges:

- **leading**
- **trailing**
- **top**
- **bottom**

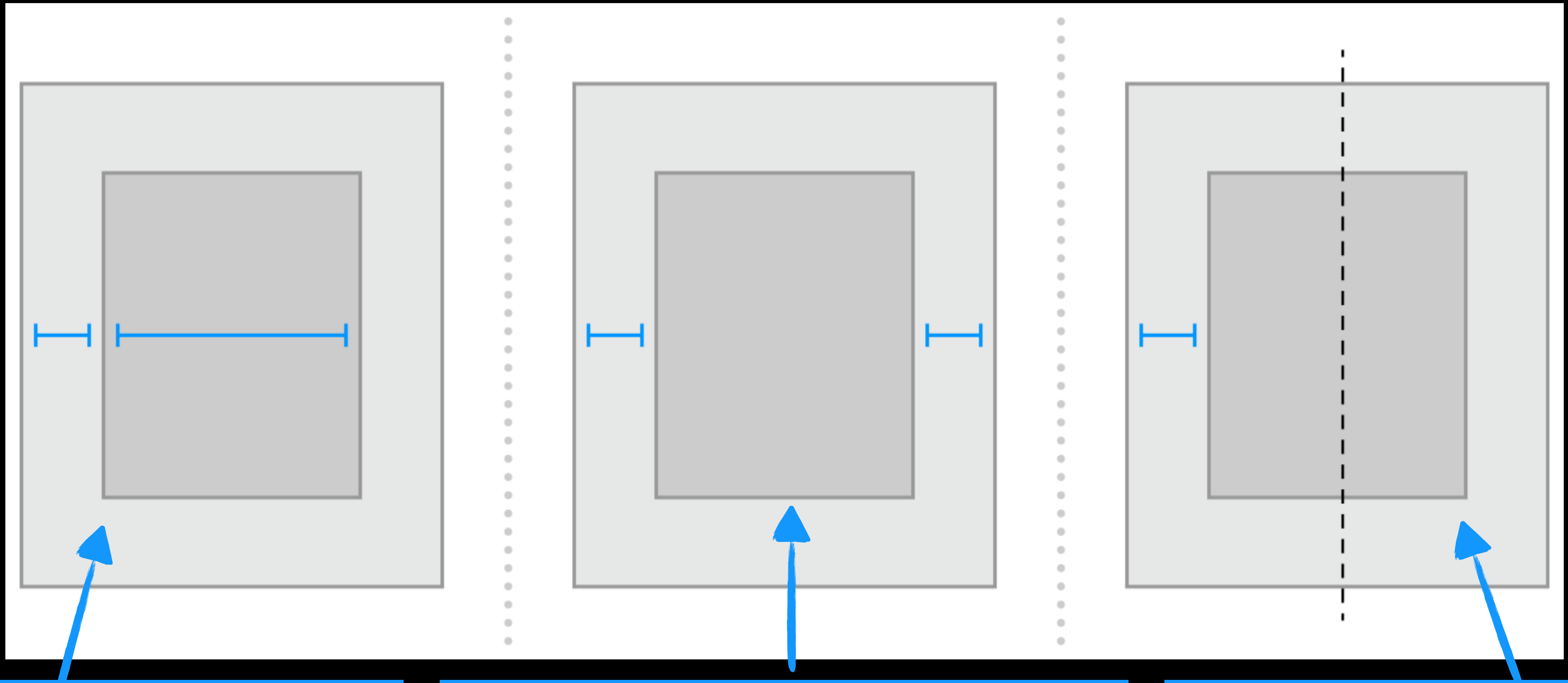
as well as…

- height
- width
- vertical and horizontal centers



Creating Non ambiguous, Satisfiable Layouts

The constraints must define both the size and the position of each view.

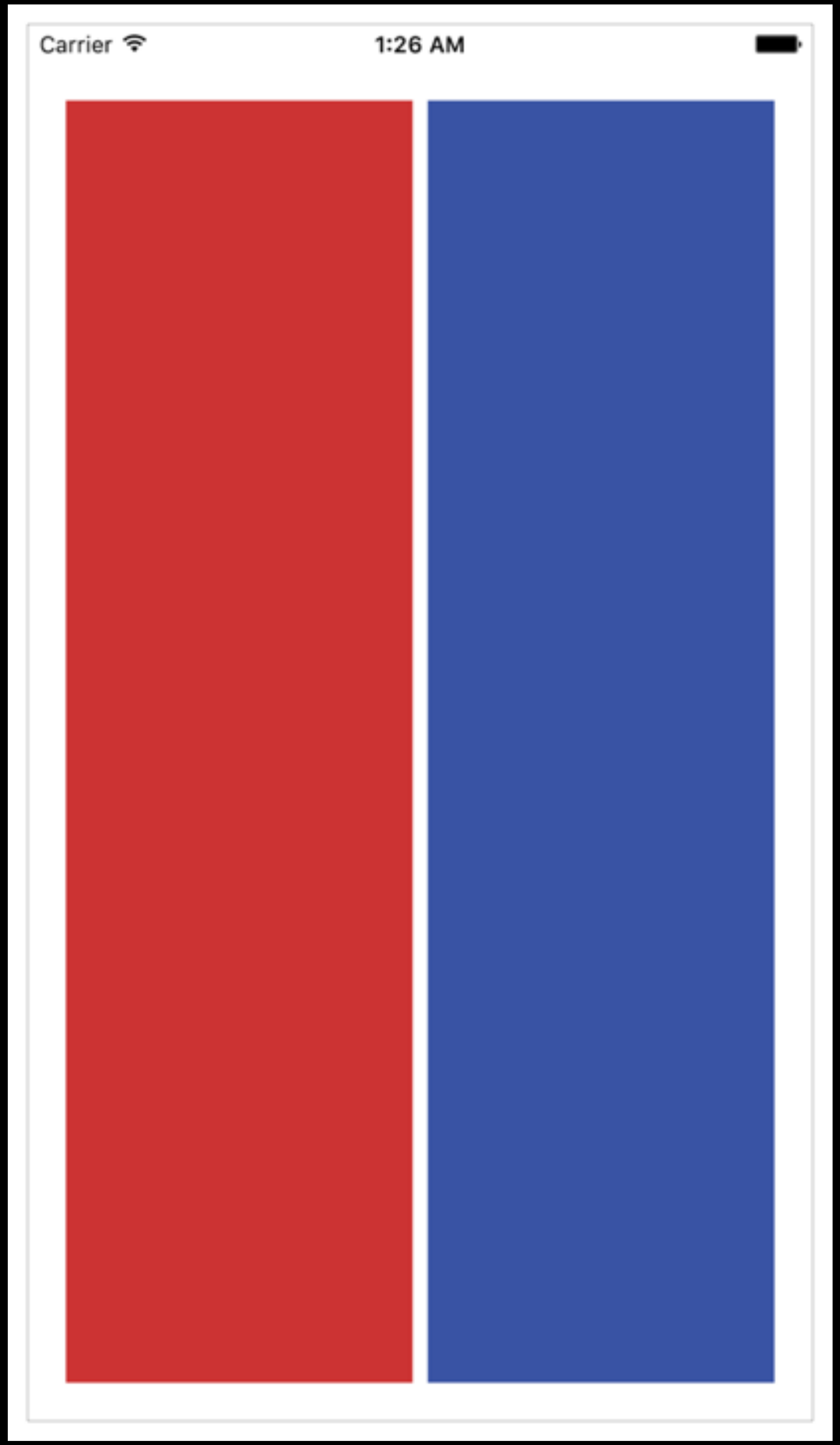


Gives the view a fixed width. The position of the trailing edge can then be calculated based on the superview's size and the other constraints.

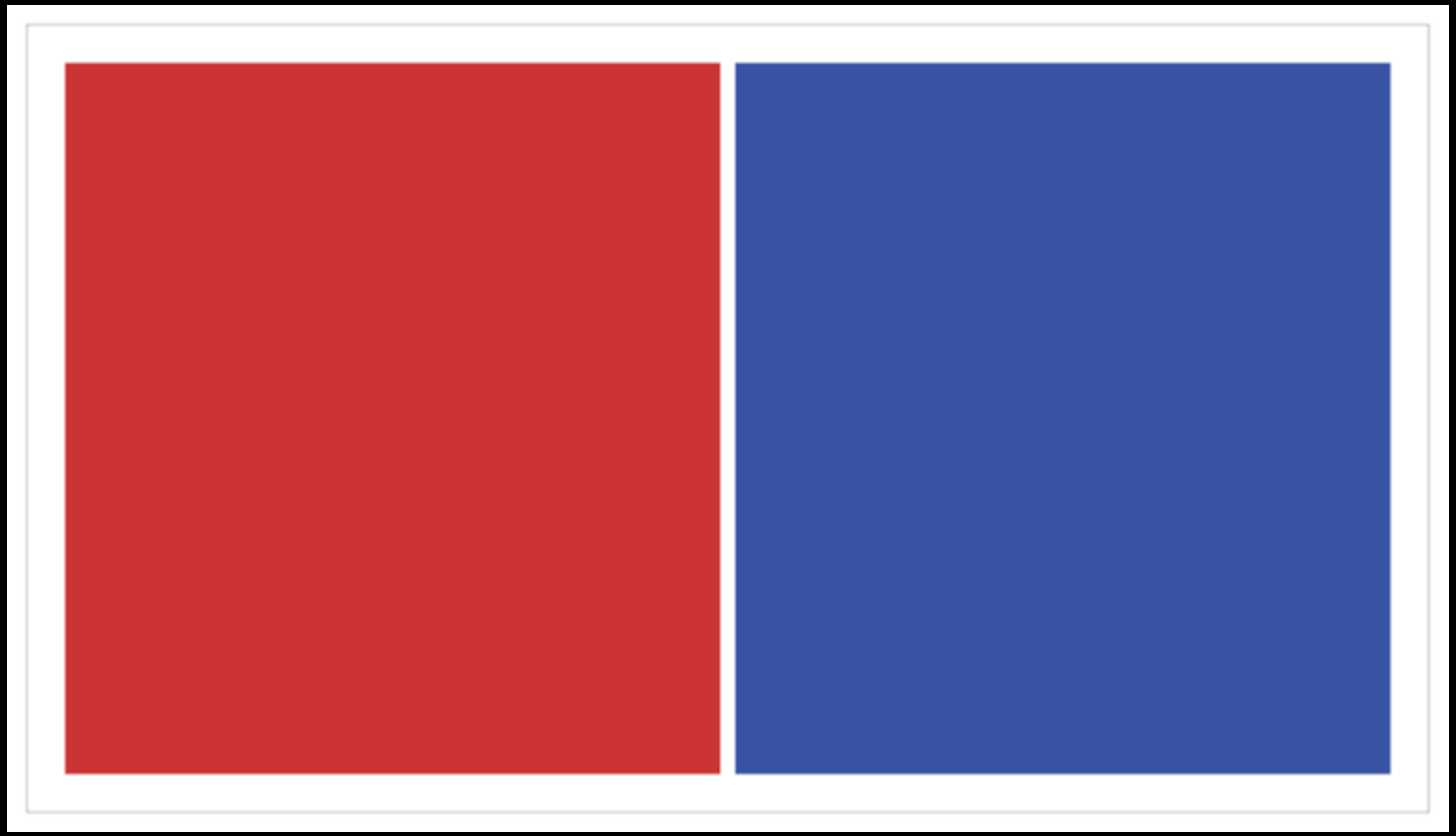
Constraints the view trailing edge relative to the superview's trailing edge. The view's width can then be calculated based on the superview's size and the other constraints.

Center aligns the view and superview. Both the width and trailing edge's position can then be calculates based on the superview's size and the other constraints.

Creating Non ambiguous, Satisfiable Layouts



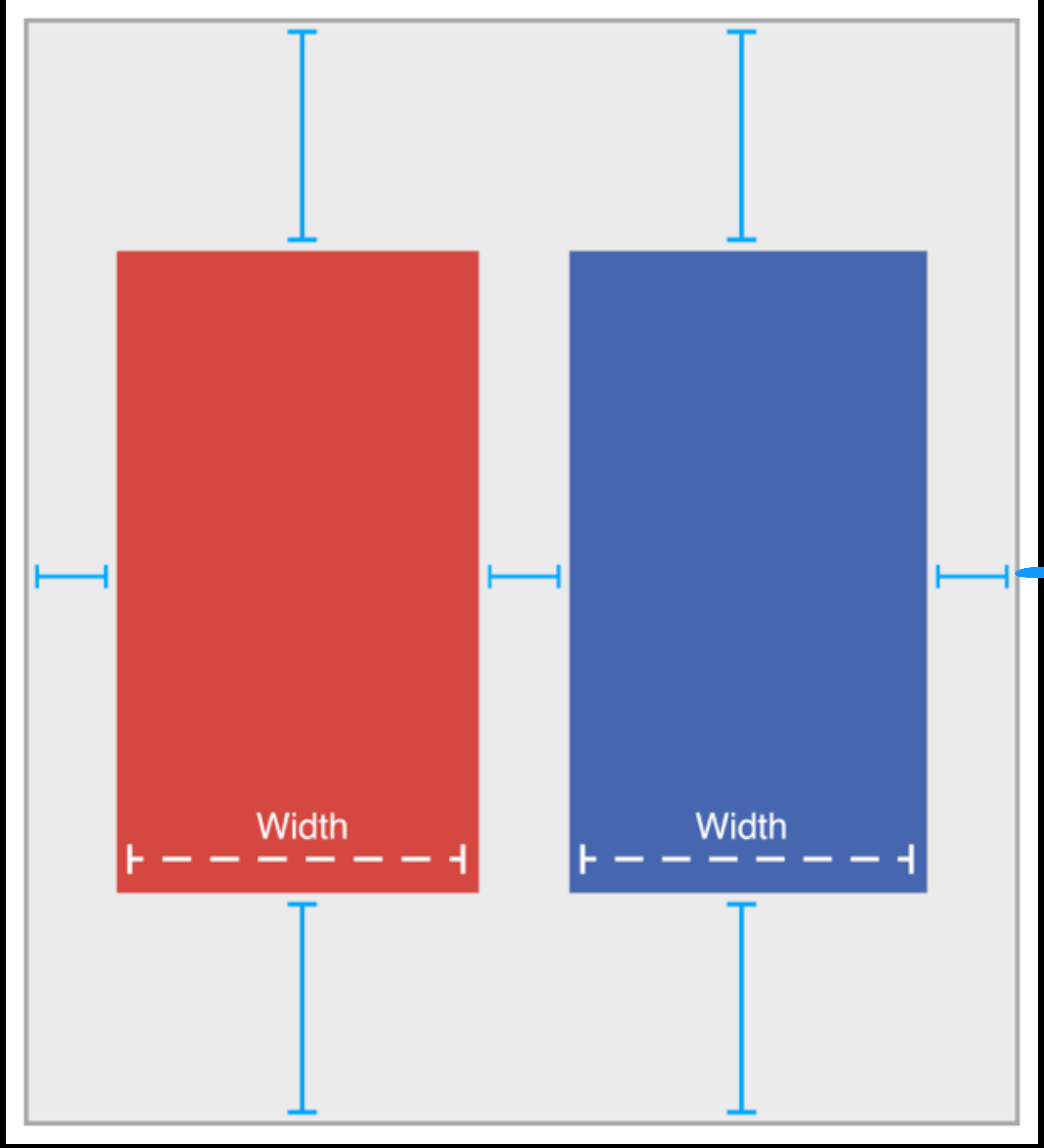
Portrait orientation



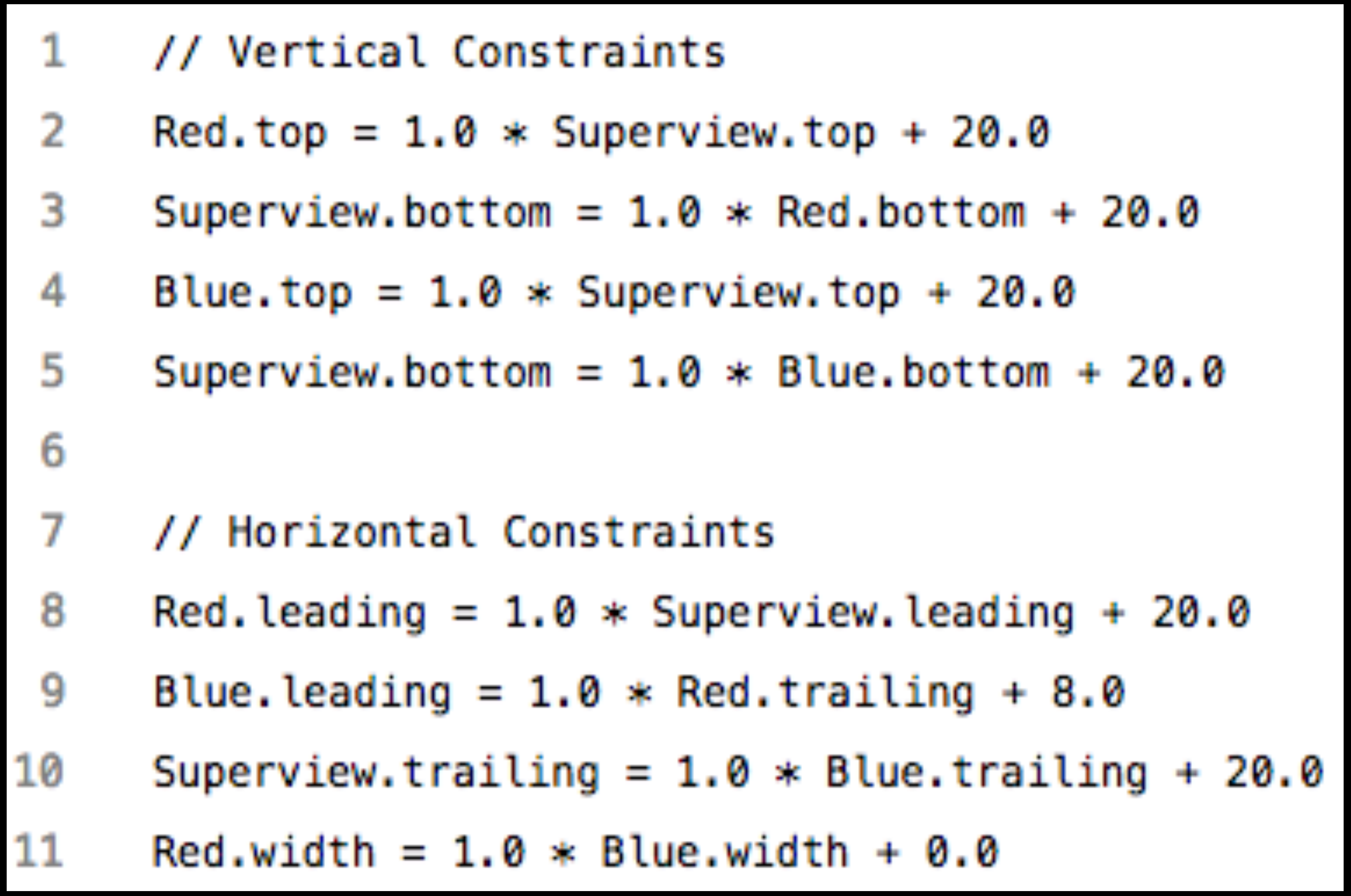
Landscape orientation

Creating Non ambiguous, Satisfiable Layouts

So what should these constraints look like?



Constraints:



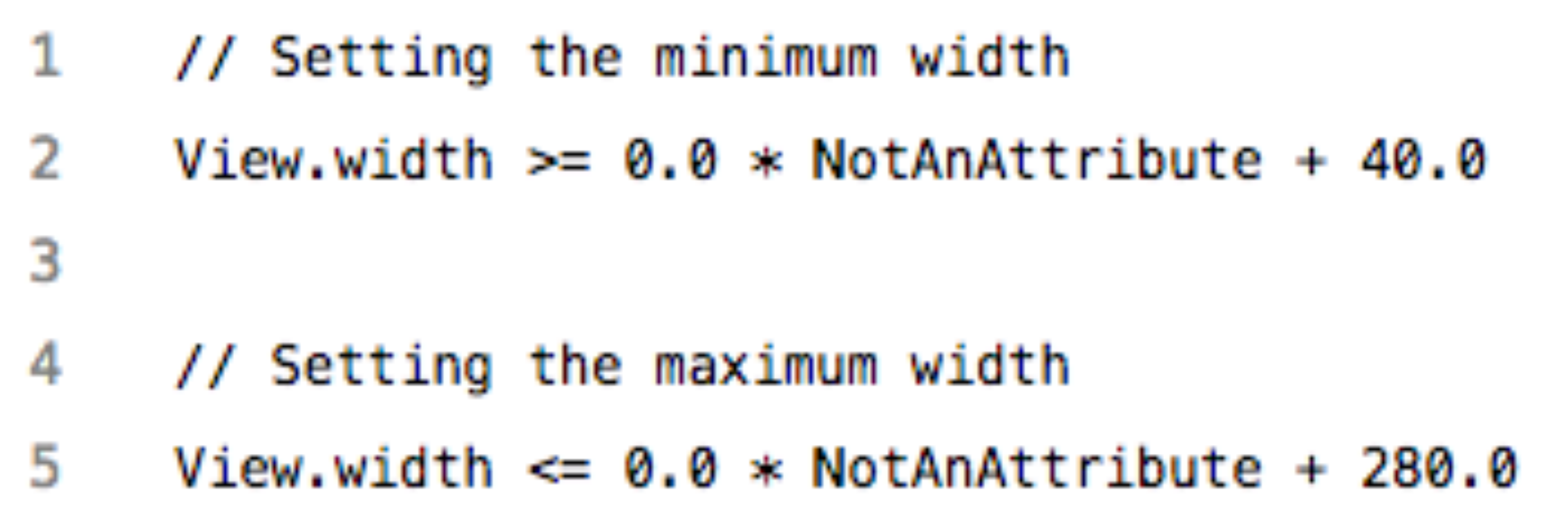
```
// Vertical Constraints
Red.top = 1.0 * Superview.top + 20.0
Superview.bottom = 1.0 * Red.bottom + 20.0
Blue.top = 1.0 * Superview.top + 20.0
Superview.bottom = 1.0 * Blue.bottom + 20.0

// Horizontal Constraints
Red.leading = 1.0 * Superview.leading + 20.0
Blue.leading = 1.0 * Red.trailing + 8.0
Superview.trailing = 1.0 * Blue.trailing + 20.0
Red.width = 1.0 * Blue.width + 0.0
```

Constraint Inequalities

Constraints can represent inequalities.

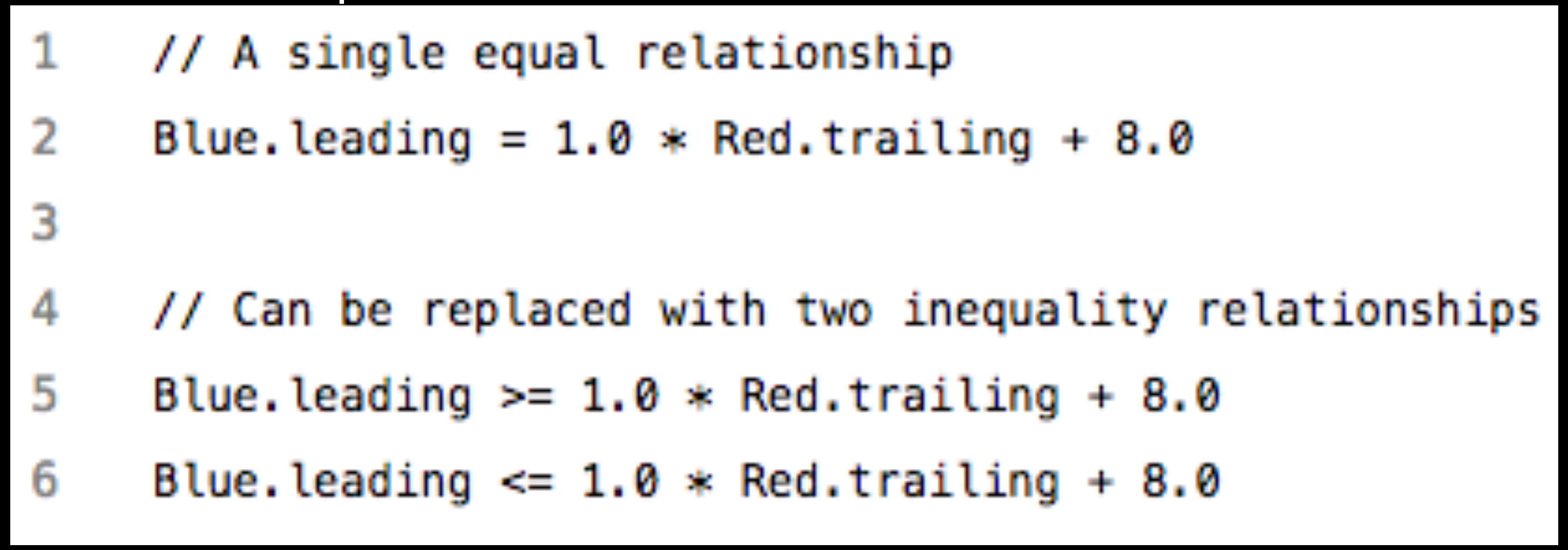
Specifically, **constraints** relationship can be equal to, greater than or equal to, or less than or equal to.

Assigning a minimum and maximum size



```
// Setting the minimum width
View.width >= 0.0 * NotAnAttribute + 40.0

// Setting the maximum width
View.width <= 0.0 * NotAnAttribute + 280.0
```

Replacing a single equal relationship with two inequalities



```
// A single equal relationship
Blue.leading = 1.0 * Red.trailing + 8.0

// Can be replaced with two inequality relationships
Blue.leading >= 1.0 * Red.trailing + 8.0
Blue.leading <= 1.0 * Red.trailing + 8.0
```

Constraint Priorities

By default, all **Constraints** are required.

Auto Layout must calculate a solution that satisfies all the constraints. If it cannot, there is an error.

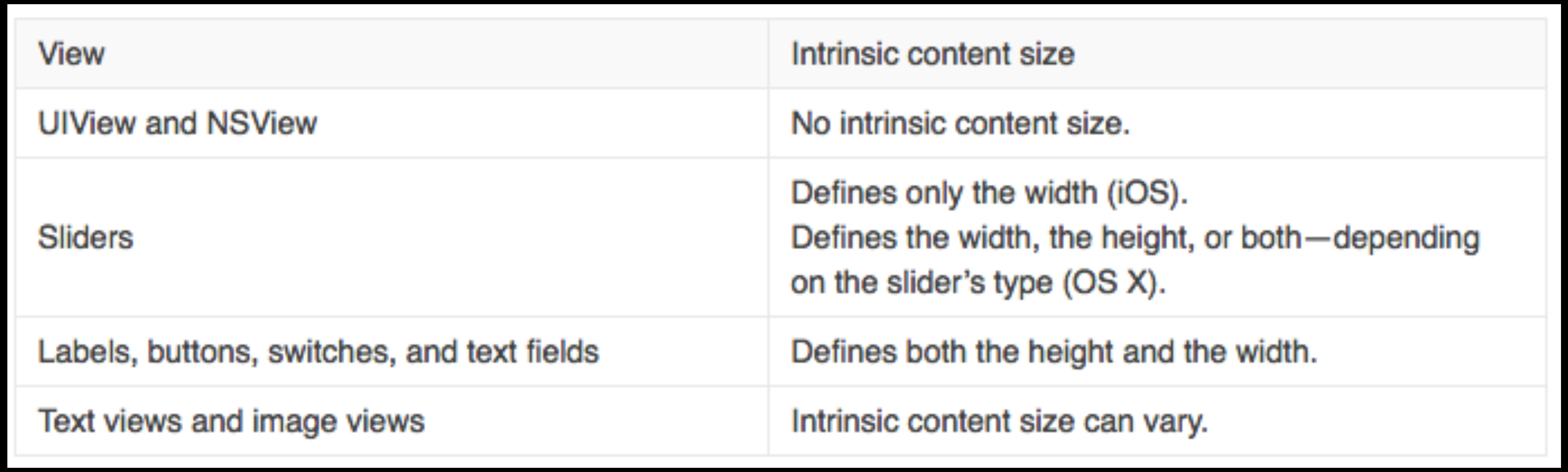
You can create **optional** constraints. All constraints have a priority between 1 and 1000. Constraints with priority of 1000 are **required**. All other constraints are **optional**.

Intrinsic Content Size

Some views have a natural size given their current content - *Intrinsic Content Size*.

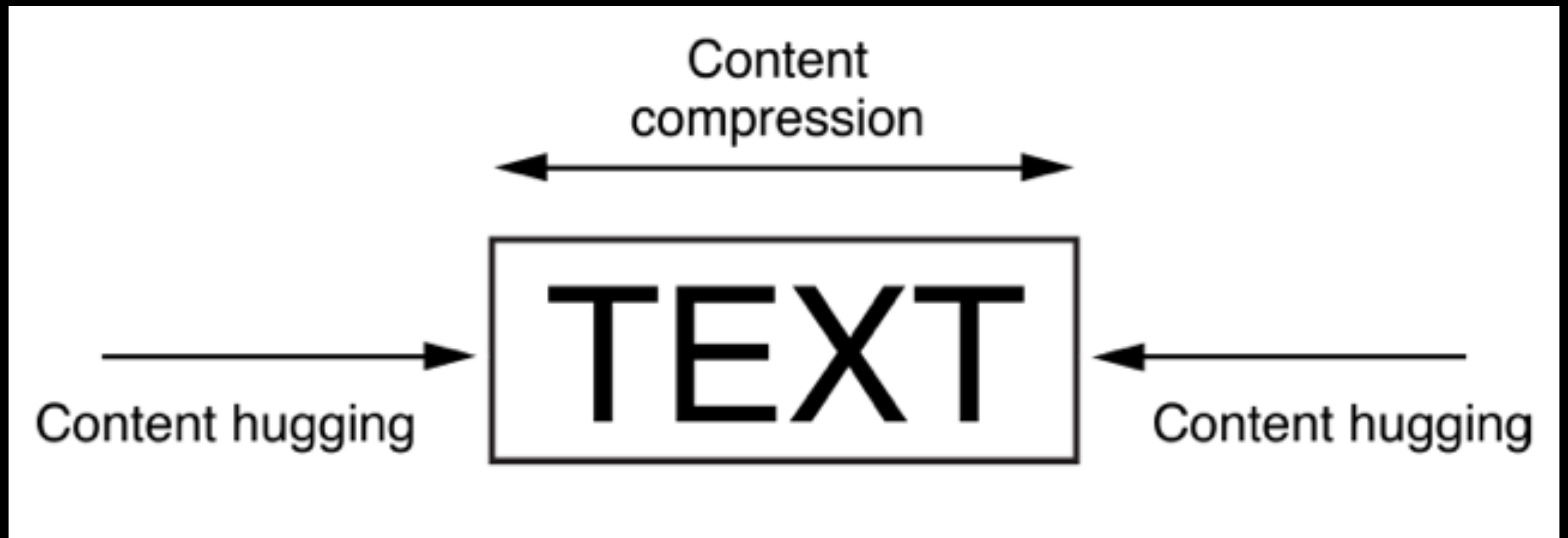
For example, a button's intrinsic content size is the size of its title plus a small margin.

Intrinsic content size for common controls

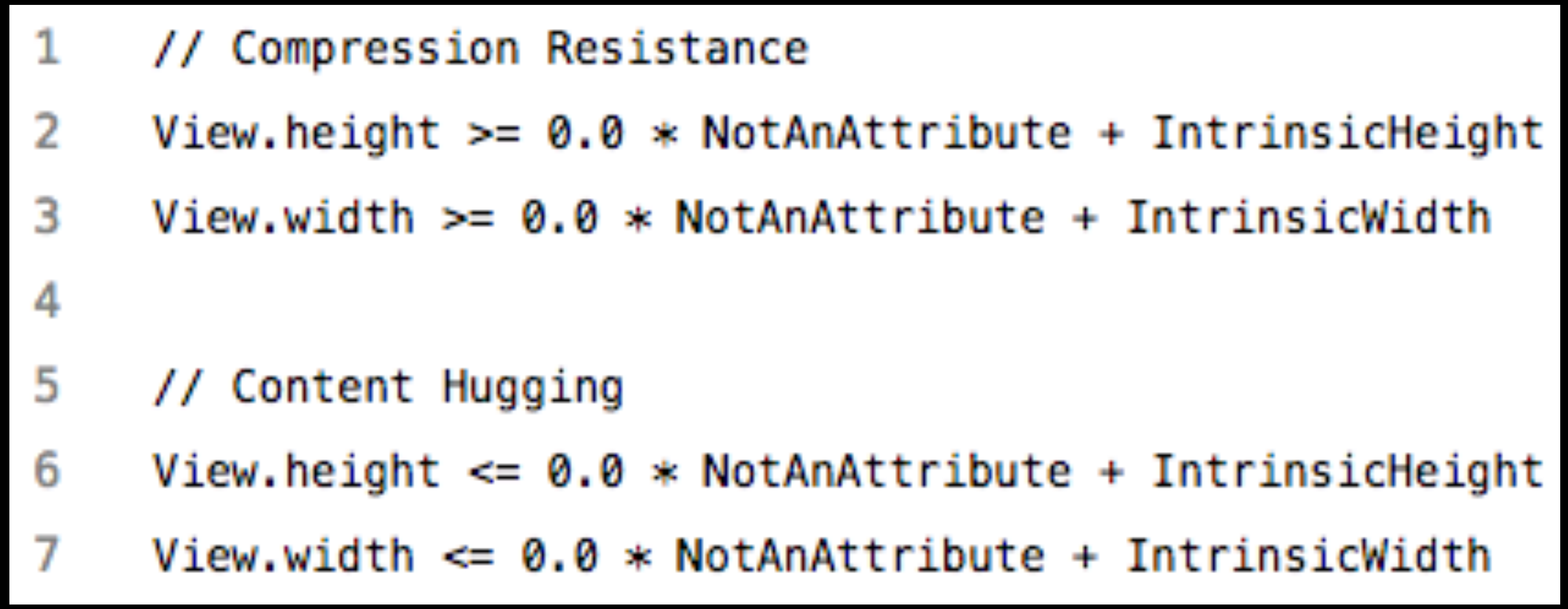


View	Intrinsic content size
UIView and NSView	No intrinsic content size.
Sliders	Defines only the width (iOS). Defines the width, the height, or both—depending on the slider's type (OS X).
Labels, buttons, switches, and text fields	Defines both the height and the width.
Text views and image views	Intrinsic content size can vary.

Intrinsic Content Size



Compression-Resistance and Content-Hugging equations



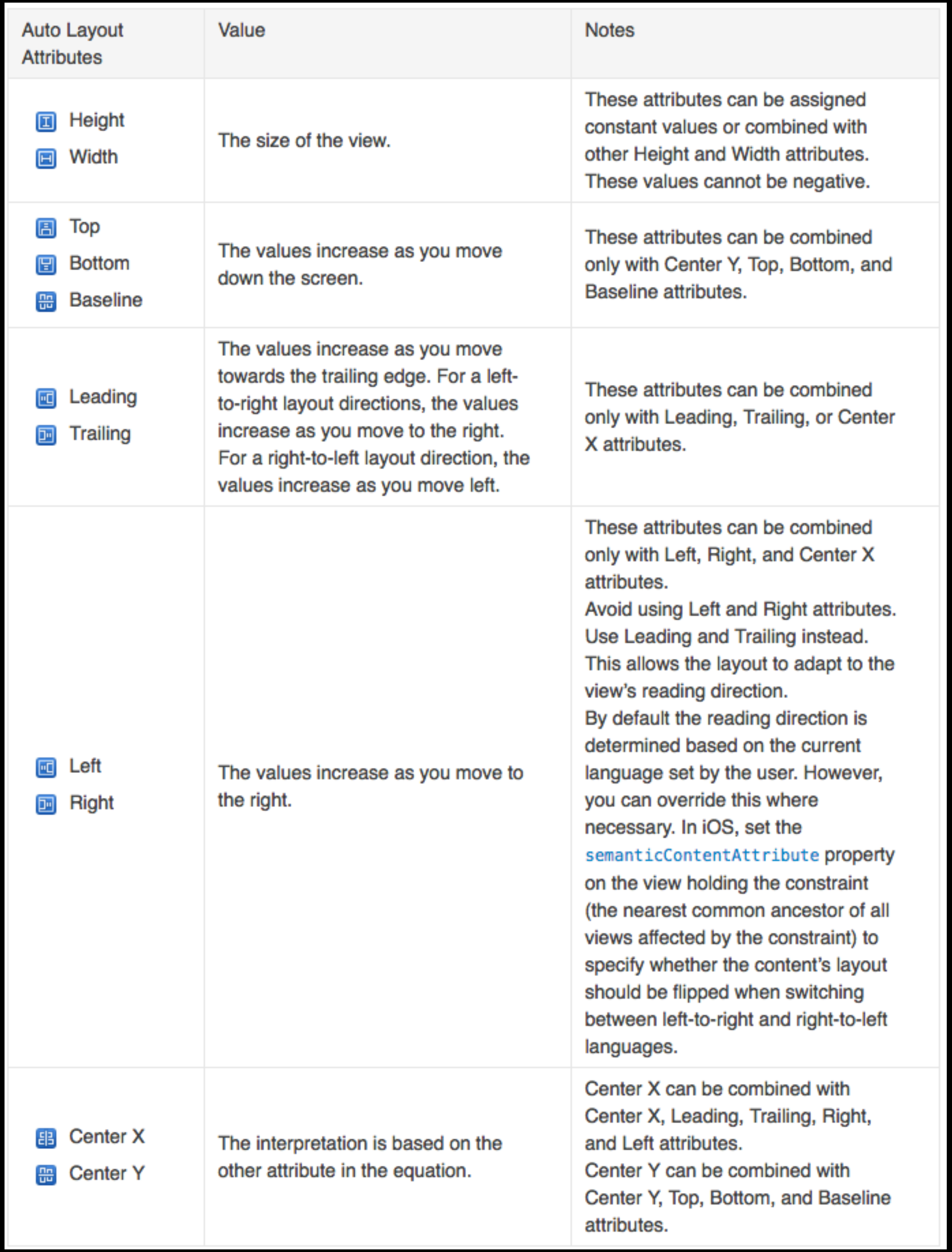
```
// Compression Resistance
View.height >= 0.0 * NotAnAttribute + IntrinsicHeight
View.width >= 0.0 * NotAnAttribute + IntrinsicWidth

// Content Hugging
View.height <= 0.0 * NotAnAttribute + IntrinsicHeight
View.width <= 0.0 * NotAnAttribute + IntrinsicWidth
```

The ***IntrinsicHeight*** and ***IntrinsicWidth*** constants represent the height and width values from the view’s intrinsic content size.

Interpreting Values

Values in **Auto Layout** are always in points.

These measurements can vary depending on the attributes involved and the view’s layout direction.



Auto Layout Attributes	Value	Notes
Height Width	The size of the view.	These attributes can be assigned constant values or combined with other Height and Width attributes. These values cannot be negative.
Top Bottom Baseline	The values increase as you move down the screen.	These attributes can be combined only with Center Y, Top, Bottom, and Baseline attributes.
Leading Trailing	The values increase as you move towards the trailing edge. For a left-to-right layout directions, the values increase as you move to the right. For a right-to-left layout direction, the values increase as you move left.	These attributes can be combined only with Leading, Trailing, or Center X attributes.
Left Right	The values increase as you move to the right.	These attributes can be combined only with Left, Right, and Center X attributes. Avoid using Left and Right attributes. Use Leading and Trailing instead. This allows the layout to adapt to the view’s reading direction. By default the reading direction is determined based on the current language set by the user. However, you can override this where necessary. In iOS, set the `semanticContentAttribute` property on the view holding the constraint (the nearest common ancestor of all views affected by the constraint) to specify whether the content’s layout should be flipped when switching between left-to-right and right-to-left languages.
Center X Center Y	The interpretation is based on the other attribute in the equation.	Center X can be combined with Center X, Leading, Trailing, Right, and Left attributes. Center Y can be combined with Center Y, Top, Bottom, and Baseline attributes.

Working with Constraints in Interface Builder

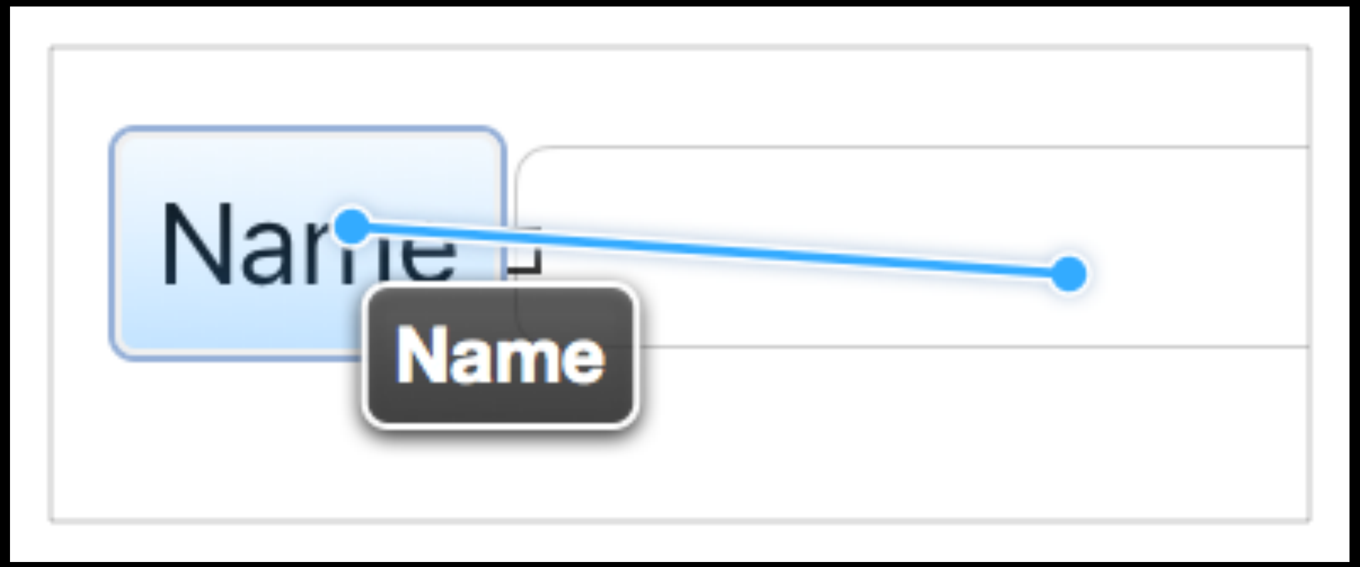
There are three main options for setting up **Auto Layout constraints**:

- Control-Dragging Constraints
- Stack, Align, Pin and Resolve Tools
- Interface Builder

Working with Constraints in Interface Builder

Control-Dragging Constraints

To create a constraint between two views, **Control-click** one of the views and drag to the other.

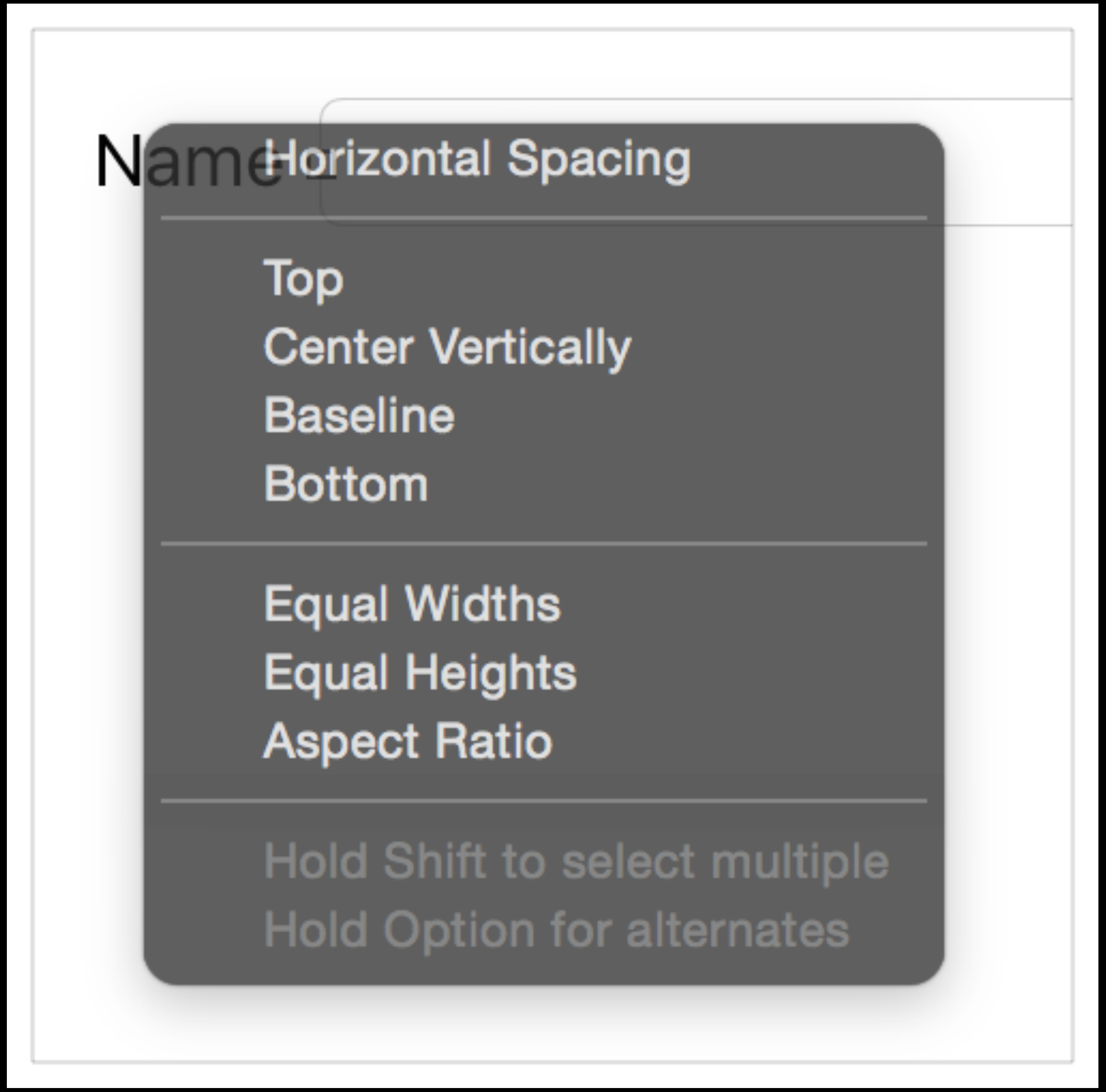


Working with Constraints in Interface Builder

Control-Dragging Constraints

When you **release** the mouse, **Interface Builder** displays a HUD menu with a list of possible constraints.

Interface Builder intelligently selects the set of constraints based on the items you are constraining and the direction of your drag gesture.

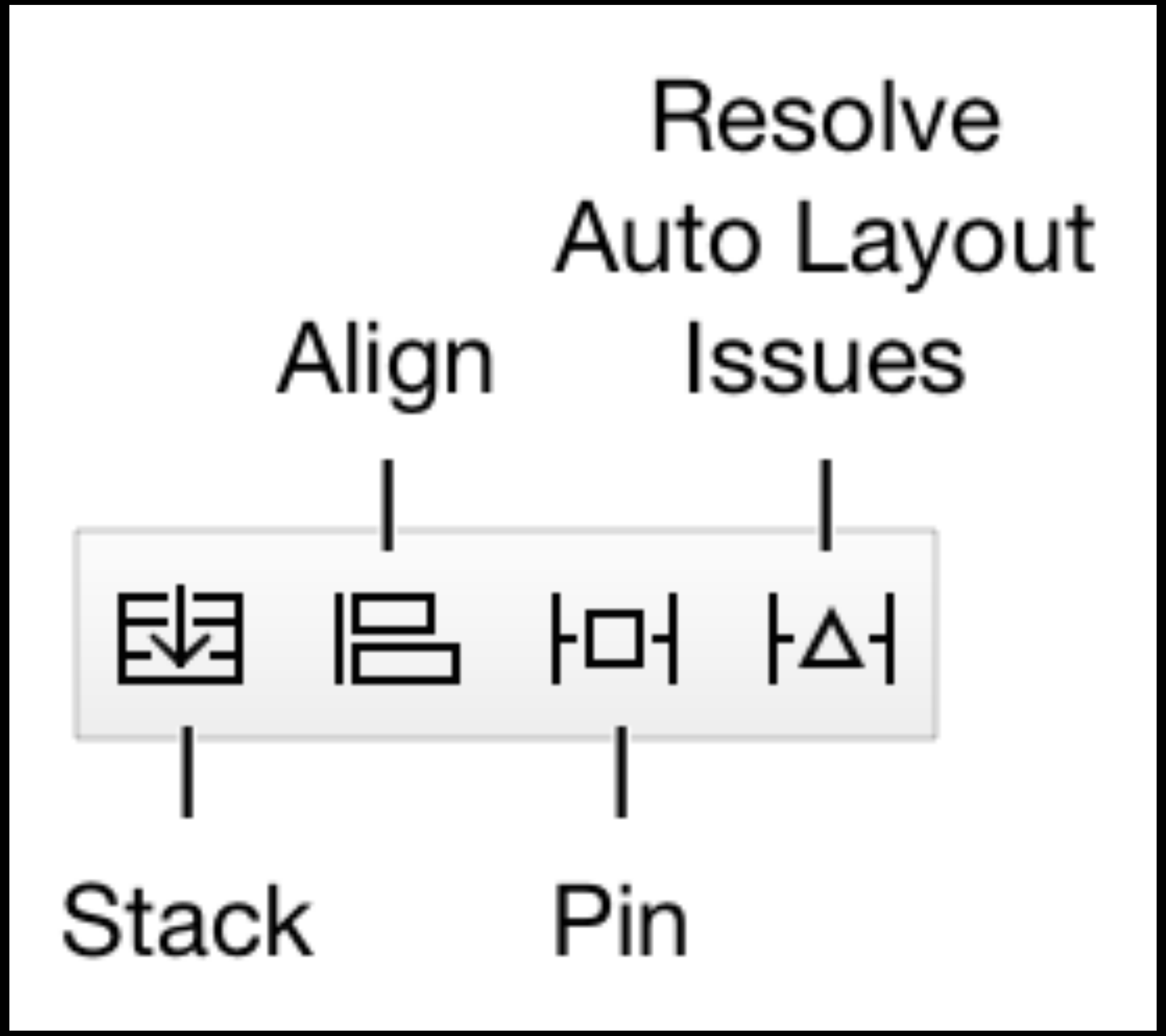


Working with Constraints in Interface Builder

Stack, Align, Pin and Resolve Tools

Interface Builder provides for Auto Layout tools in the bottom-right corner of the Editor window:

- Stack
- Align
- Pin
- Resolve Auto Layout Issues Tools



Working with Constraints in Interface Builder

Stack Tool

Allows you to quickly create a stack view. **Select** one or more items in your layout, and then click on the Stack tool.

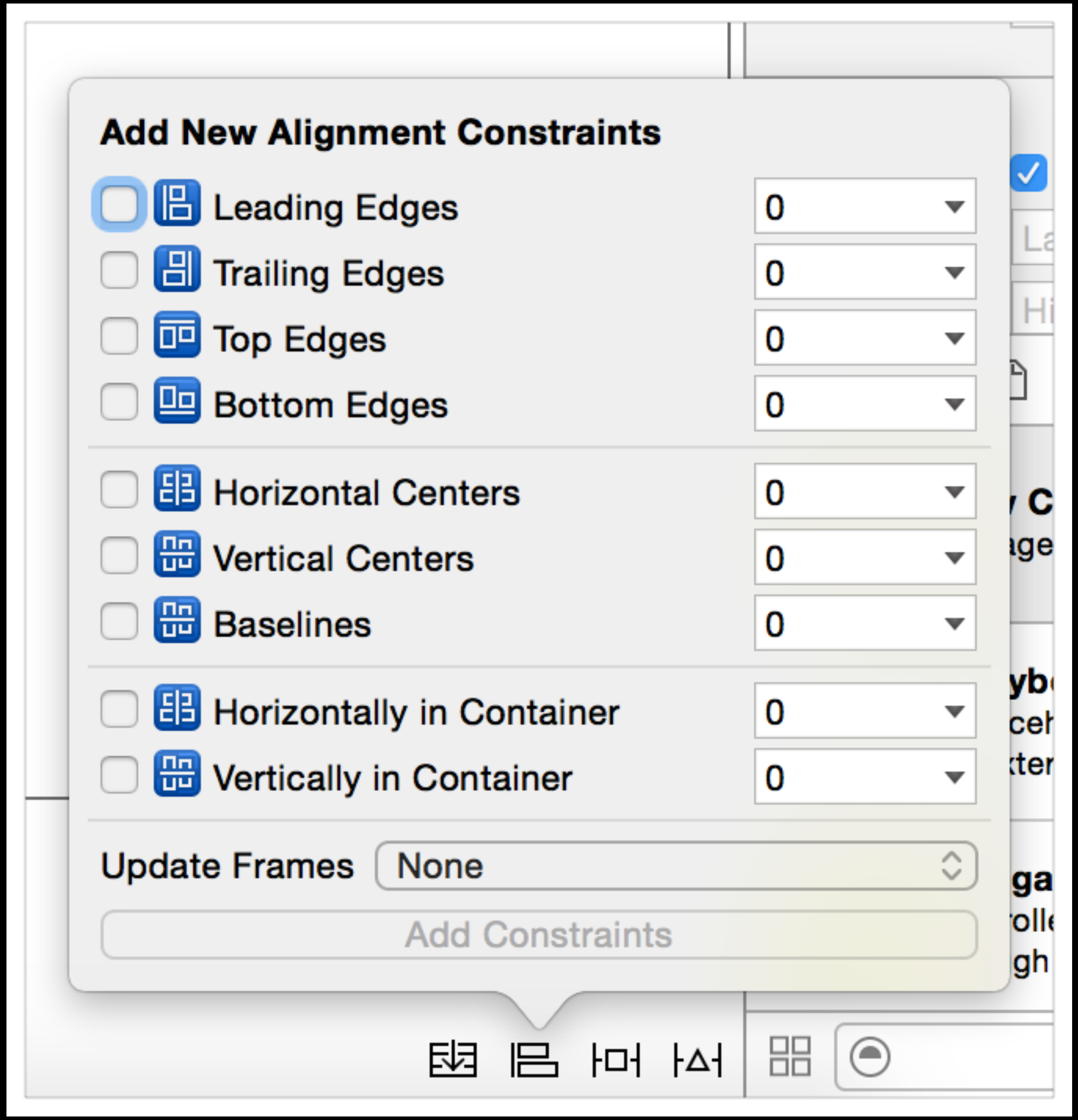
Interface Builder embeds the selected items in a stack view and resizes the stack to its current fitting size based on its contents.

Working with Constraints in Interface Builder

Align Tool

Align tool lets you quickly align items in your layout. Select the items you want to align, and then click the Align tool.

Interface Builder presents a popover view containing a number of possible alignments.



Add New Alignment Constraints

- □ Leading Edges 0
- □ Trailing Edges 0
- □ Top Edges 0
- □ Bottom Edges 0
- □ Horizontal Centers 0
- □ Vertical Centers 0
- □ Baselines 0
- □ Horizontally in Container 0
- □ Vertically in Container 0

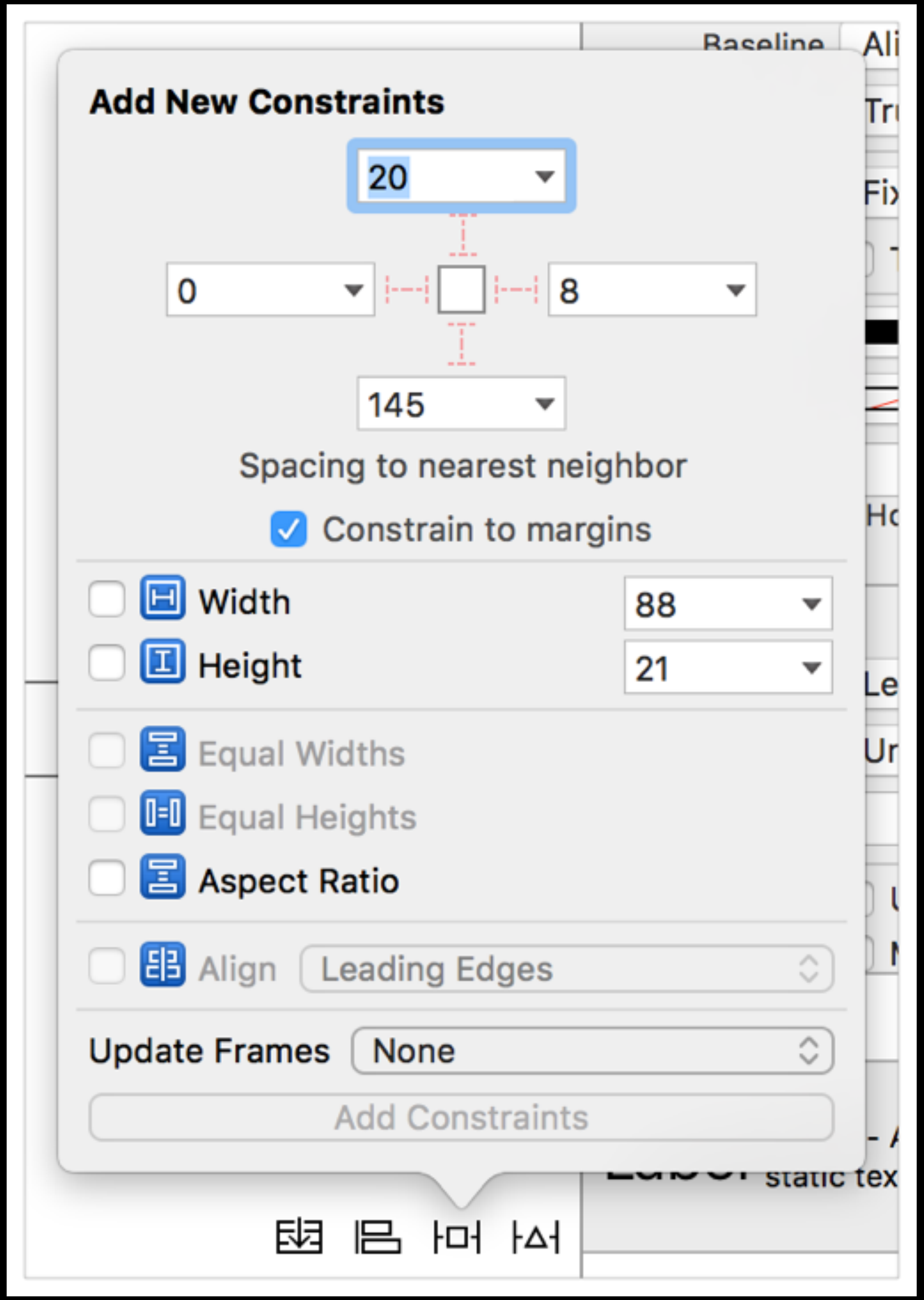
Update Frames: None

Working with Constraints in Interface Builder

Pin Tool

Pin tool lets you quickly defines a view’s position relative to its neighbors or quickly define its size.

Interface Builder presents a popover view containing a number of options.



Add New Constraints

20

0 8

145

Spacing to nearest neighbor

☑ Constrain to margins

□ Width 88

□ Height 21

□ Equal Widths

□ Equal Heights

□ Aspect Ratio

□ Align Leading Edges

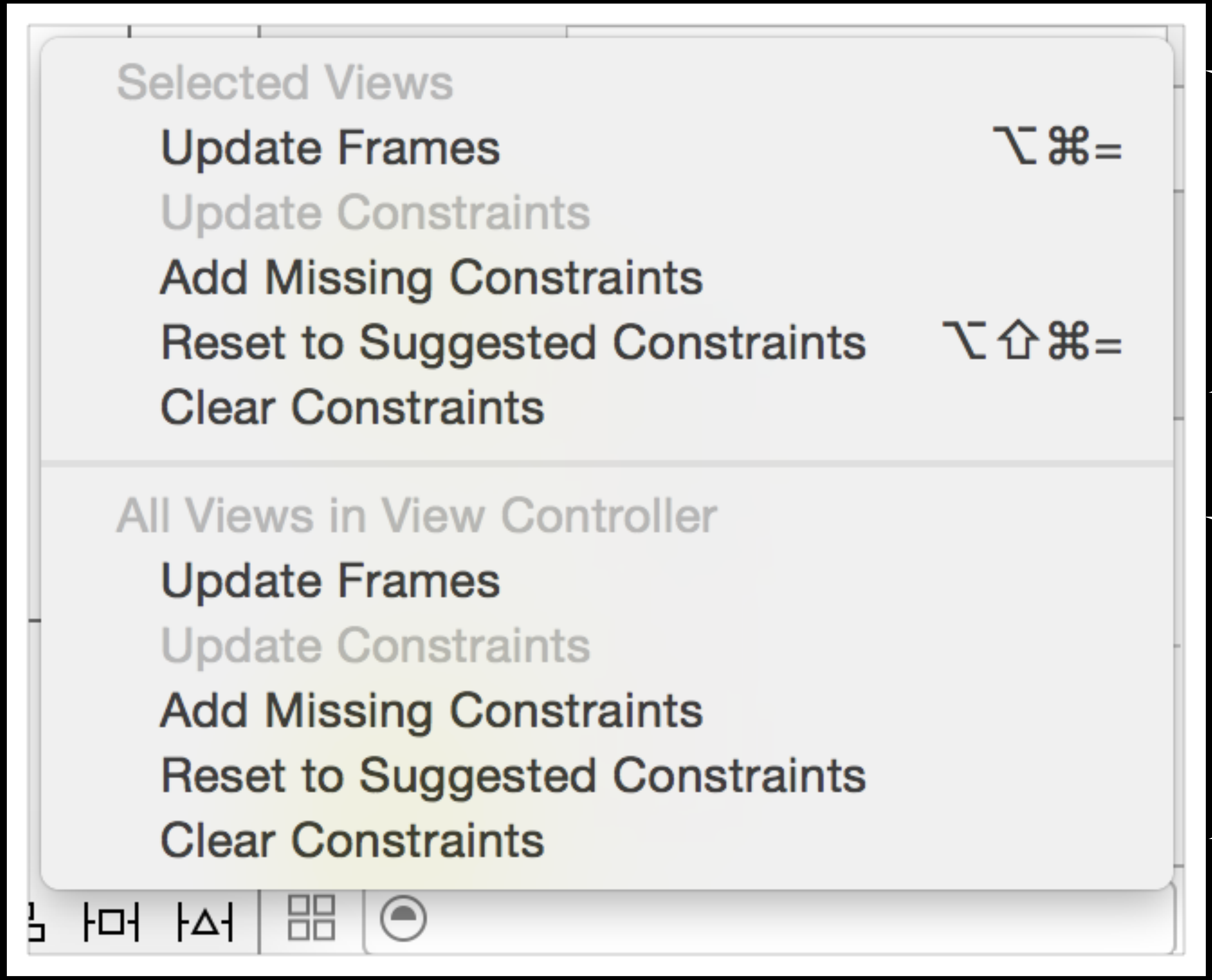
Update Frames None

Add Constraints

Working with Constraints in Interface Builder

Resolve Auto Layout Issues Tool

Provides a number of options for fixing common Auto Layout issues.



Affect only the currently selected views.

Affect all views in the scene.

Letting Interface Builder Create Constraints

Interface Builder can create some or all of the constraints for you.

Interface Builder attempts to infer the best constraints given the view's current size and position in the canvas.

Finding and Editing Constraints

After added a constraint, you need to find it, view it, and edit it.

There are a number of options for accessing the constraints. Each option offers a unique method of organizing and presenting the constraints.

Finding and Editing Constraints

Viewing Constraints in the Canvas

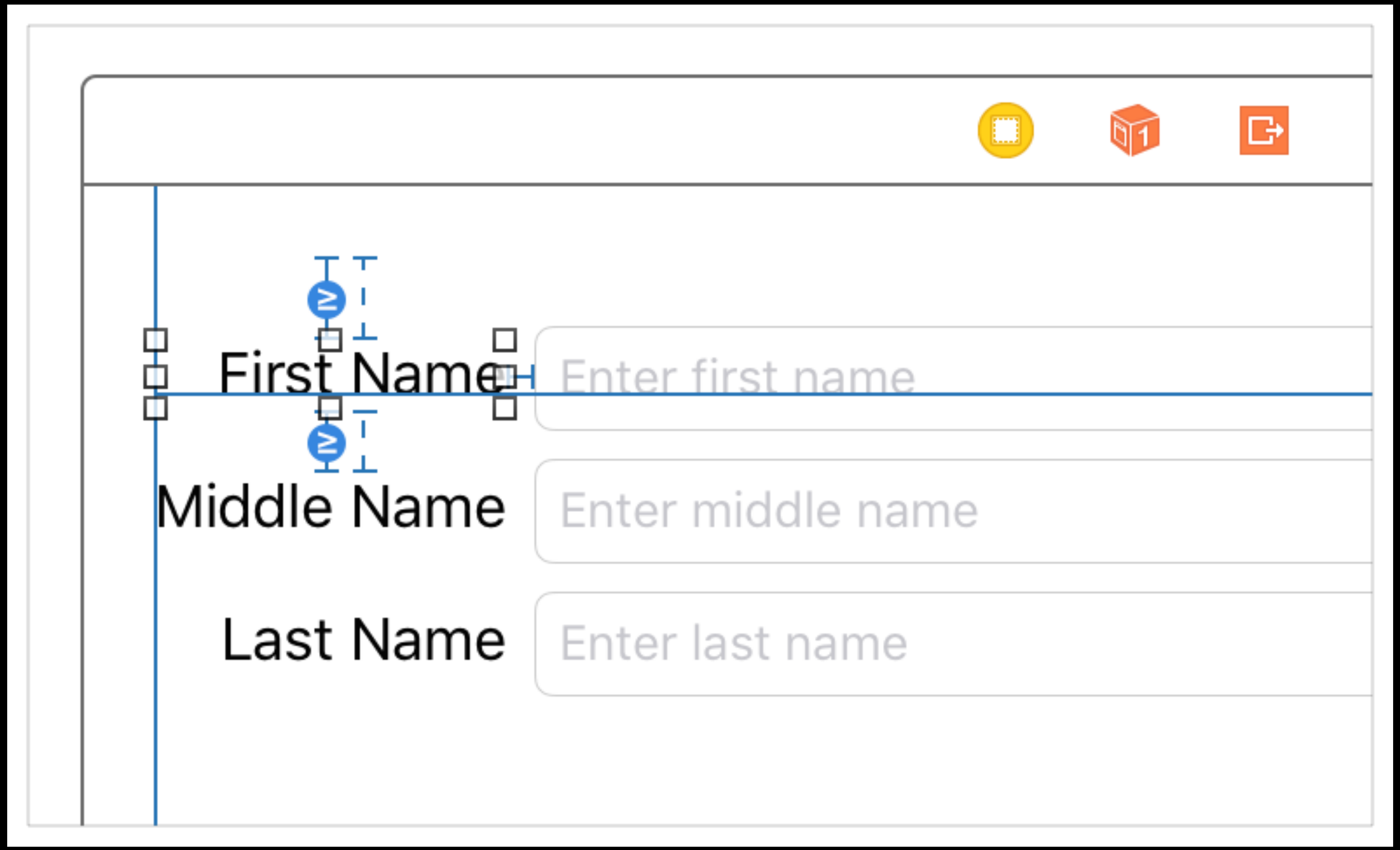
The editor displays all the constraints affecting the currently selected view as colored lines on the canvas.

The shape, stroke type, and line color can tell you a lot about the current state os the constraint.

Finding and Editing Constraints

Viewing Constraints in the Canvas

- I-bars (lines with T-shaped end-caps
- Plain lines (straight lines with no end-caps)
- Plain lines (straight lines with no end-caps)
- Solid lines
- Dashed Lines
- Red Lines
- Orange Lines
- Blue Lines
- Equal Badges
- Greater-than-or-equal and less-than-or-equals badges



First Name

Enter first name

Middle Name

Enter middle name

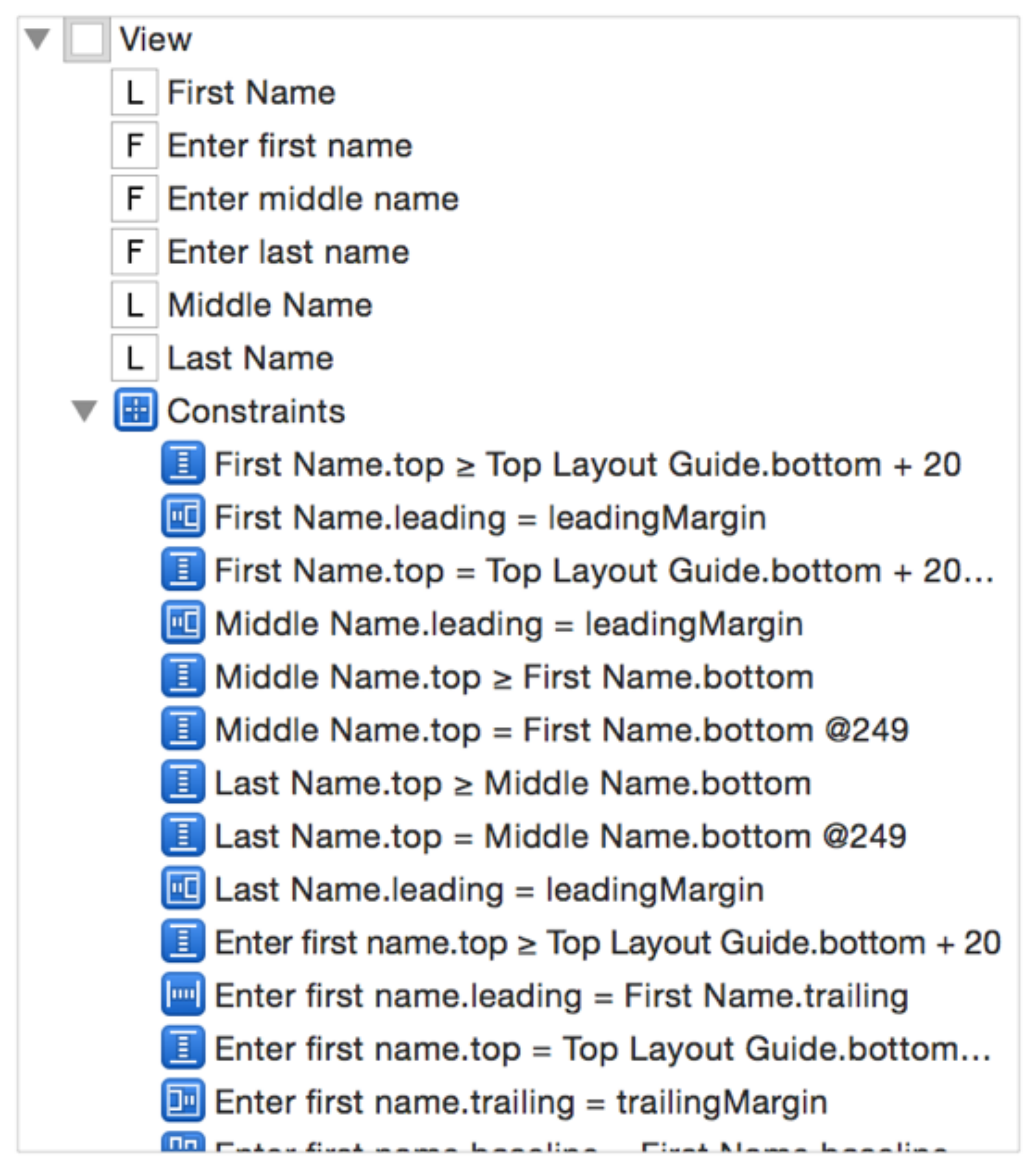
Last Name

Enter last name

Finding and Editing Constraints

Listing Constraints in the Document Outline

Interface Builder lists all the constraints in the document outline, grouping them under the view that holds them



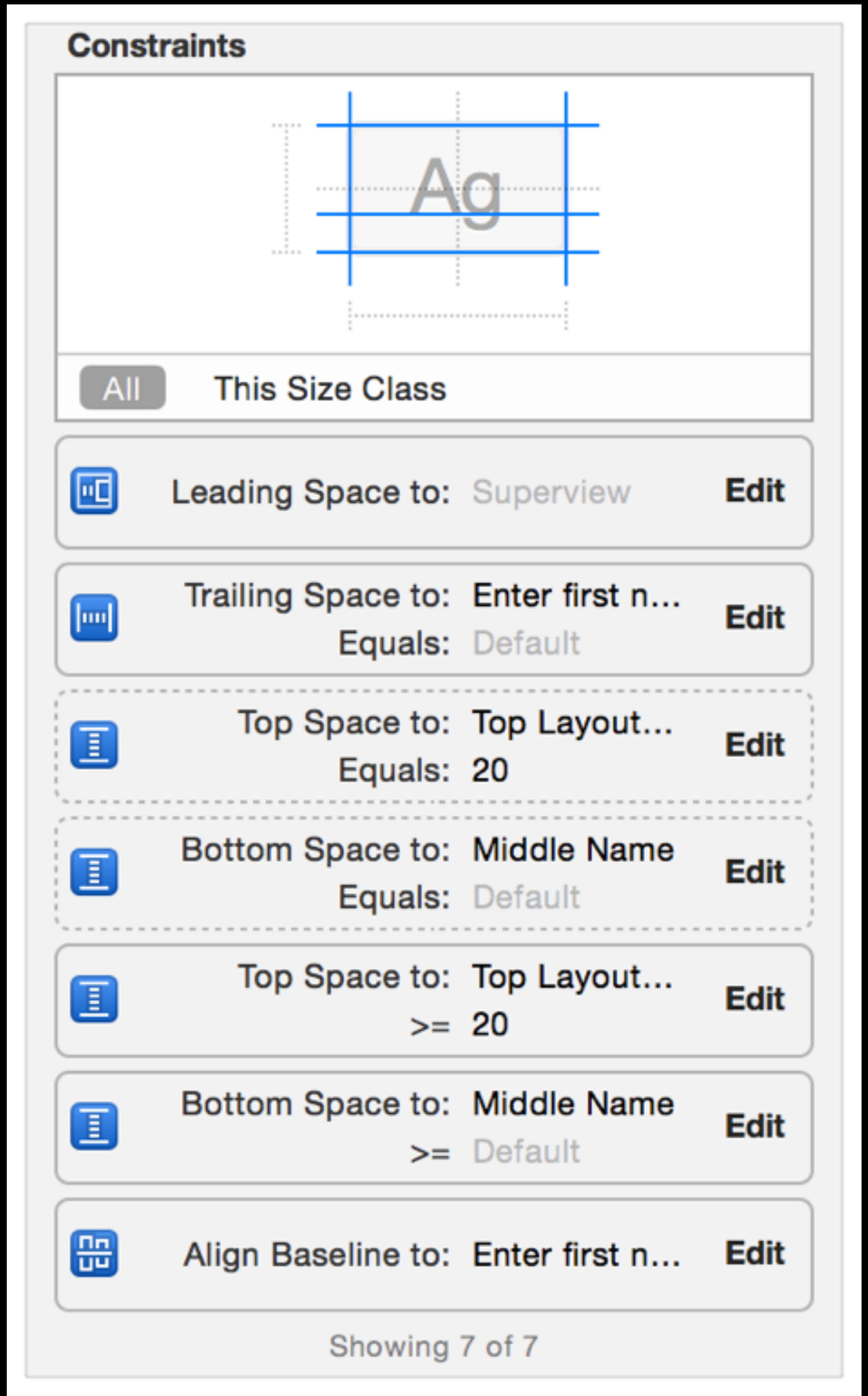
- View
  - L First Name
  - F Enter first name
  - F Enter middle name
  - F Enter last name
  - L Middle Name
  - L Last Name
  - Constraints
    - First Name.top ≥ Top Layout Guide.bottom + 20
    - First Name.leading = leadingMargin
    - First Name.top = Top Layout Guide.bottom + 20…
    - Middle Name.leading = leadingMargin
    - Middle Name.top ≥ First Name.bottom
    - Middle Name.top = First Name.bottom @249
    - Last Name.top ≥ Middle Name.bottom
    - Last Name.top = Middle Name.bottom @249
    - Last Name.leading = leadingMargin
    - Enter first name.top ≥ Top Layout Guide.bottom + 20
    - Enter first name.leading = First Name.trailing
    - Enter first name.top = Top Layout Guide.bottom…
    - Enter first name.trailing = trailingMargin
    - Enter first name.baseline = First Name.baseline

Finding and Editing Constraints

Finding Constraints in the Size Inspector

Size inspector lists all the constraints affecting the currently selected view.

Required constraints appears with a solid outline, and optional constraints appear with a dashed outline.



Constraints

All This Size Class

Leading Space to: Superview Edit

Trailing Space to: Enter first n... Equals: Default Edit

Top Space to: Top Layout... Equals: 20 Edit

Bottom Space to: Middle Name Equals: Default Edit

Top Space to: Top Layout... >= 20 Edit

Bottom Space to: Middle Name >= Default Edit

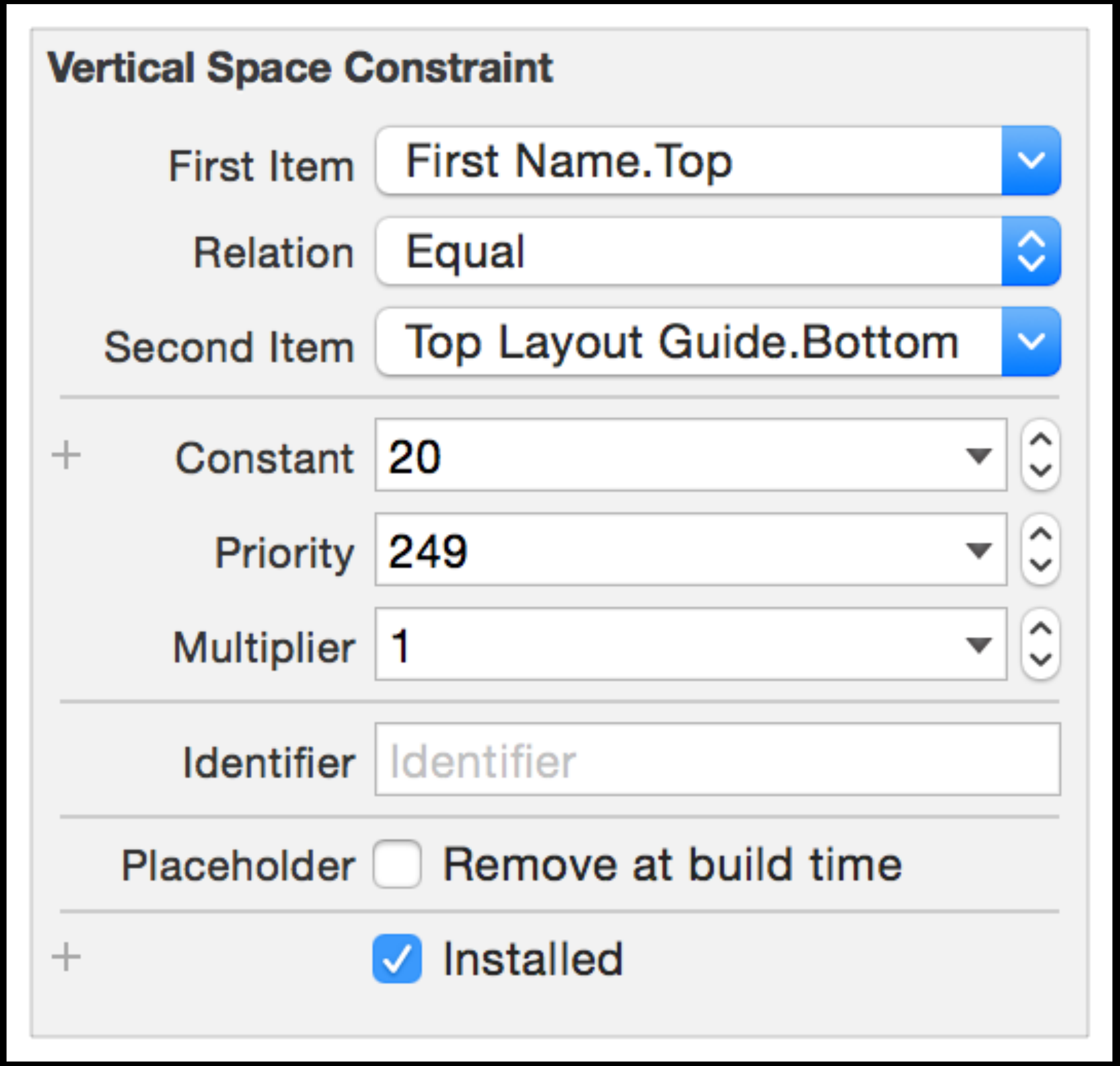
Align Baseline to: Enter first n... Edit

Showing 7 of 7

Finding and Editing Constraints

Examining and Editing Constraints

When you select a constraint either in the canvas or in the document outline, the Attribute inspector shows all of the constraint's attributes.



Vertical Space Constraint

First Item: First Name.Top

Relation: Equal

Second Item: Top Layout Guide.Bottom

Constant: 20

Priority: 249

Multiplier: 1

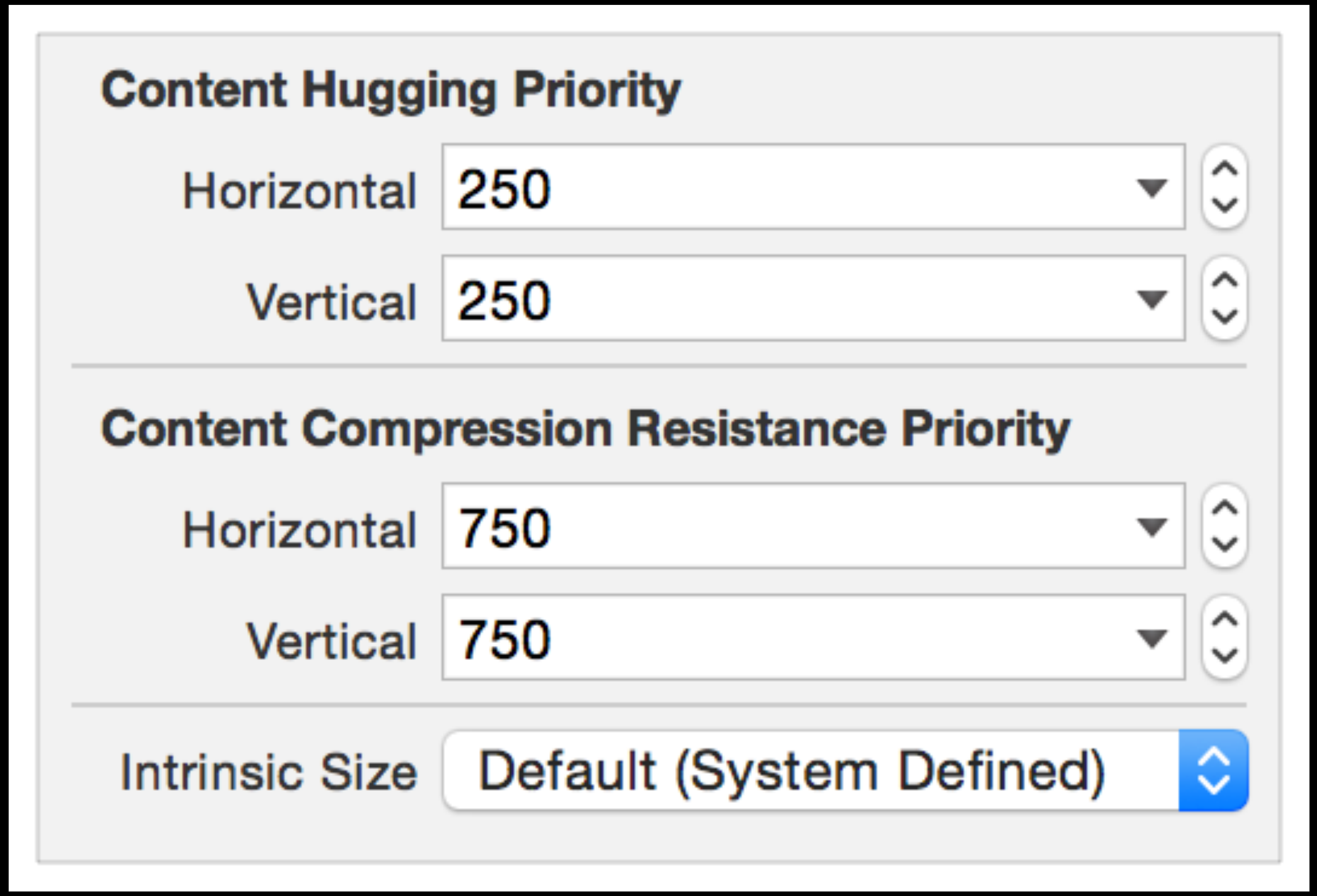
Identifier: Identifier

Placeholder: □ Remove at build time

☑ Installed

Setting Content-Hugging and Compression-Resistance Priorities

To set a view's content-hugging and compression-resistance priorities (CHCR priorities), select the view either in the canvas or in the document outline.



Content Hugging Priority

Horizontal 250

Vertical 250

Content Compression Resistance Priority

Horizontal 750

Vertical 750

Intrinsic Size Default (System Defined)