

# Autolayout

# View uses Autolayout if...

- Your code adds an autolayout constraint to a view. The views involved in this constraint use autolayout.
- Your app loads a nib for which “Use Auto Layout” is checked. Every view instantiated from that nib uses autolayout.
- A view in the interface, which would be an instance of a custom UIView subclass of yours, returns true from the class method `requiresConstraintBasedLayout`. That view uses autolayout.

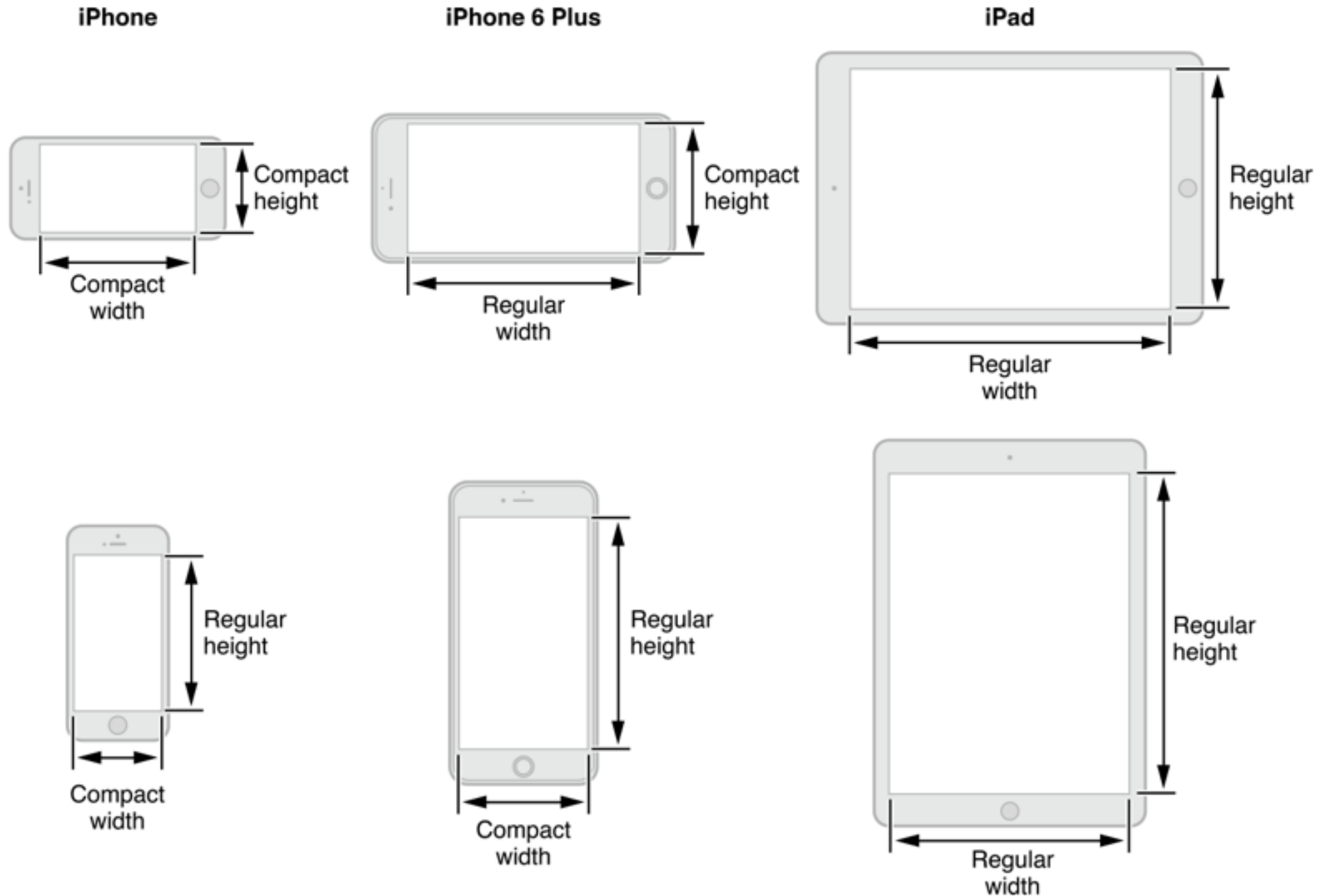
# External Changes

- The user enters or leaves Split View on an iPad.
- The device rotates.
- The active call and audio recording bars appear or disappear.
- You want to support different size classes.
- You want to support different screen sizes.

# Internal Changes

- Internal changes occur when the size of the views or controls in your user interface change.
- Here are some common sources of internal change:
  - The content displayed by the app changes.
  - The app supports internationalization.
  - The app supports Dynamic Type

# Size Classes



# Constraint



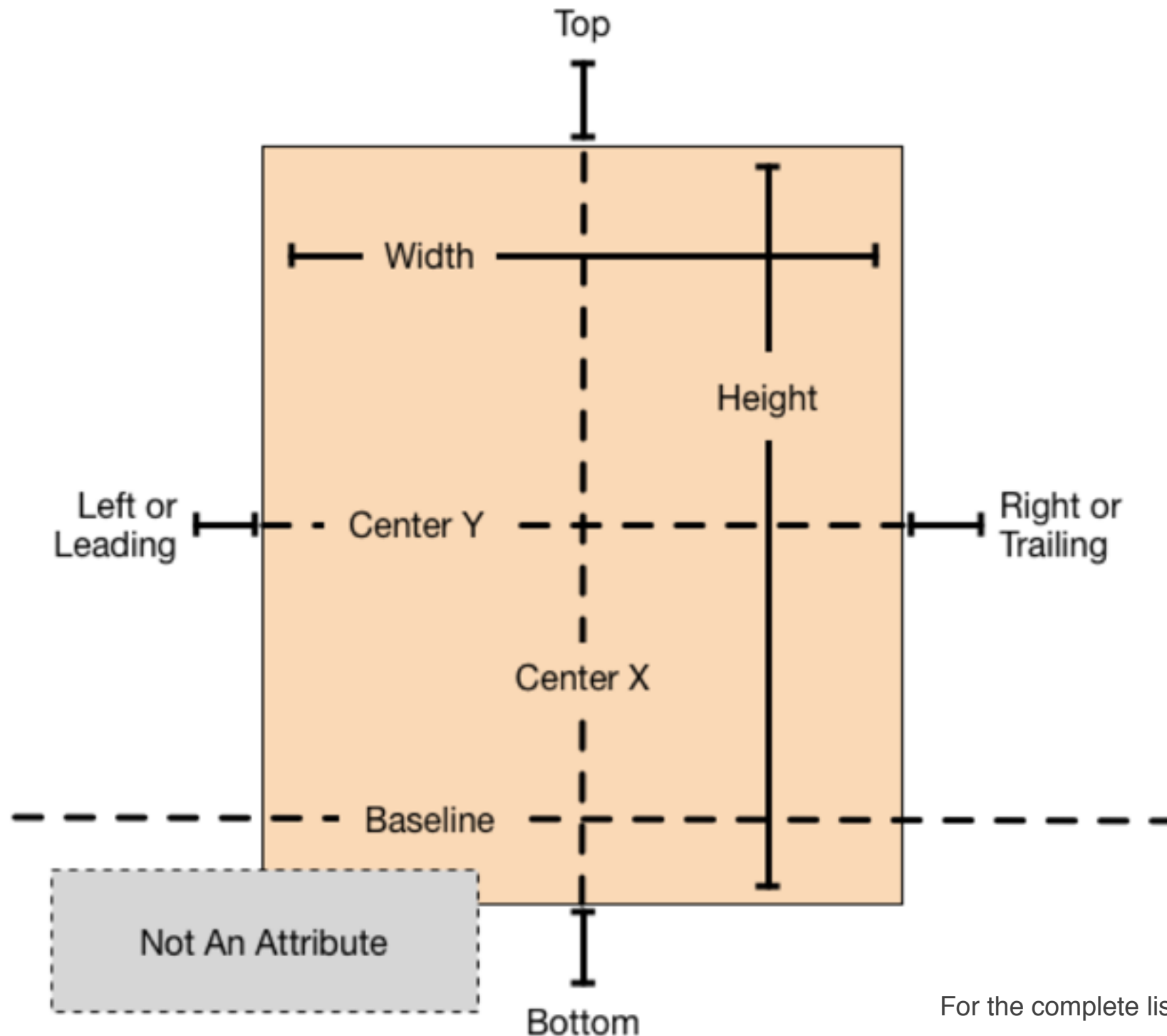
$$\underbrace{\text{RedView.Leading}}_{\text{Item 1}} = \underbrace{1.0}_{\text{Multiplier}} \times \underbrace{\text{BlueView.trailing}}_{\text{Item 2}} + \underbrace{8.0}_{\text{Constant}}$$

Relationship

Attribute 2

Attribute 1

# Auto Layout Attributes



For the complete list of attributes, see the [NSLayoutAttribute](#) enum.

# Sample Equations

// Setting a constant height

$\text{View.height} = 0.0 * \text{NotAnAttribute} + 40.0$

// Setting a fixed distance between two buttons

$\text{Button}_2.\text{leading} = 1.0 * \text{Button}_1.\text{trailing} + 8.0$

// Aligning the leading edge of two buttons

$\text{Button}_1.\text{leading} = 1.0 * \text{Button}_2.\text{leading} + 0.0$

// Give two buttons the same width

$\text{Button}_1.\text{width} = 1.0 * \text{Button}_2.\text{width} + 0.0$

// Center a view in its superview

$\text{View.centerX} = 1.0 * \text{Superview.centerX} + 0.0$

$\text{View.centerY} = 1.0 * \text{Superview.centerY} + 0.0$

// Give a view a constant aspect ratio

$\text{View.height} = 2.0 * \text{View.width} + 0.0$



# Inverted equations

```
// Setting a fixed distance between two buttons  
Button_1.trailing = 1.0 * Button_2.leading - 8.0
```












```
// Aligning the leading edge of two buttons  
Button_2.leading = 1.0 * Button_1.leading + 0.0
```

```
// Give two buttons the same width  
Button_2.width = 1.0 * Button.width + 0.0
```

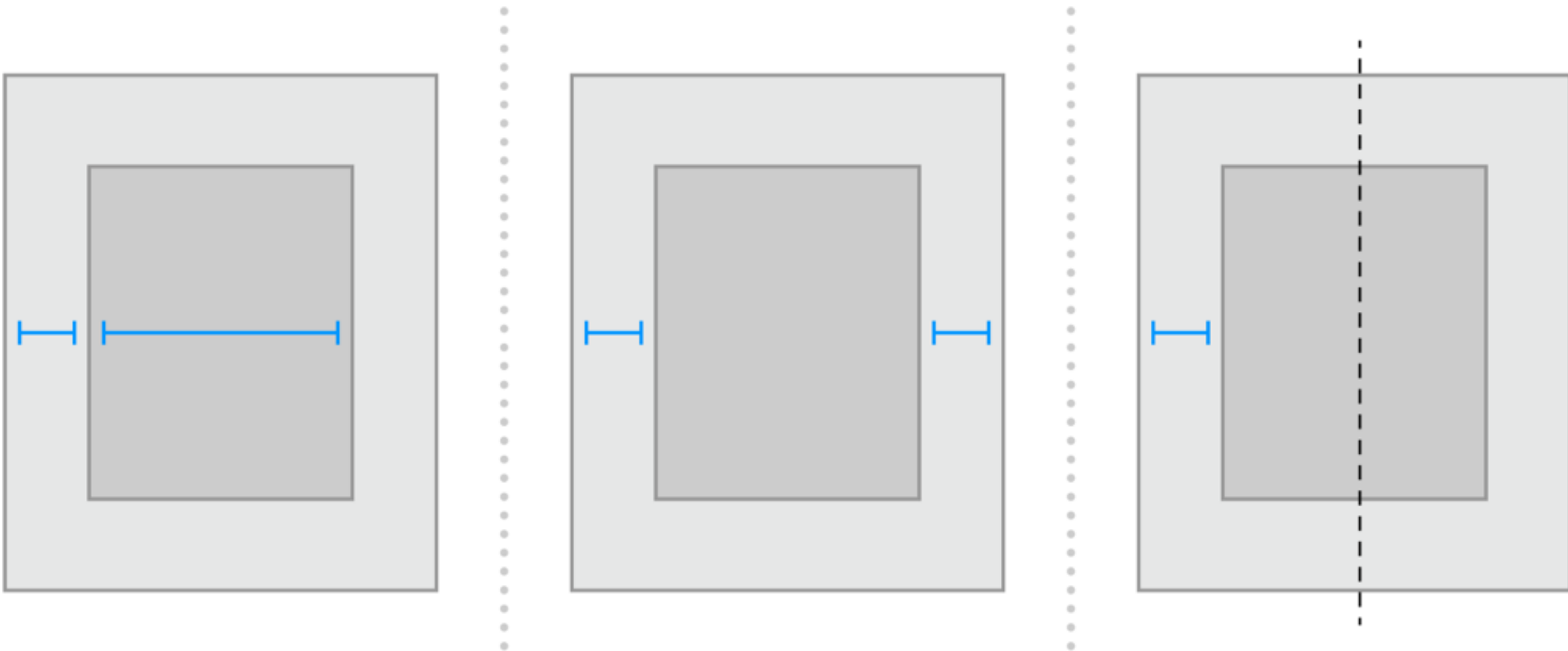
```
// Center a view in its superview  
Superview.centerX = 1.0 * View.centerX + 0.0  
Superview.centerY = 1.0 * View.centerY + 0.0
```

```
// Give a view a constant aspect ratio  
View.width = 0.5 * View.height + 0.0
```

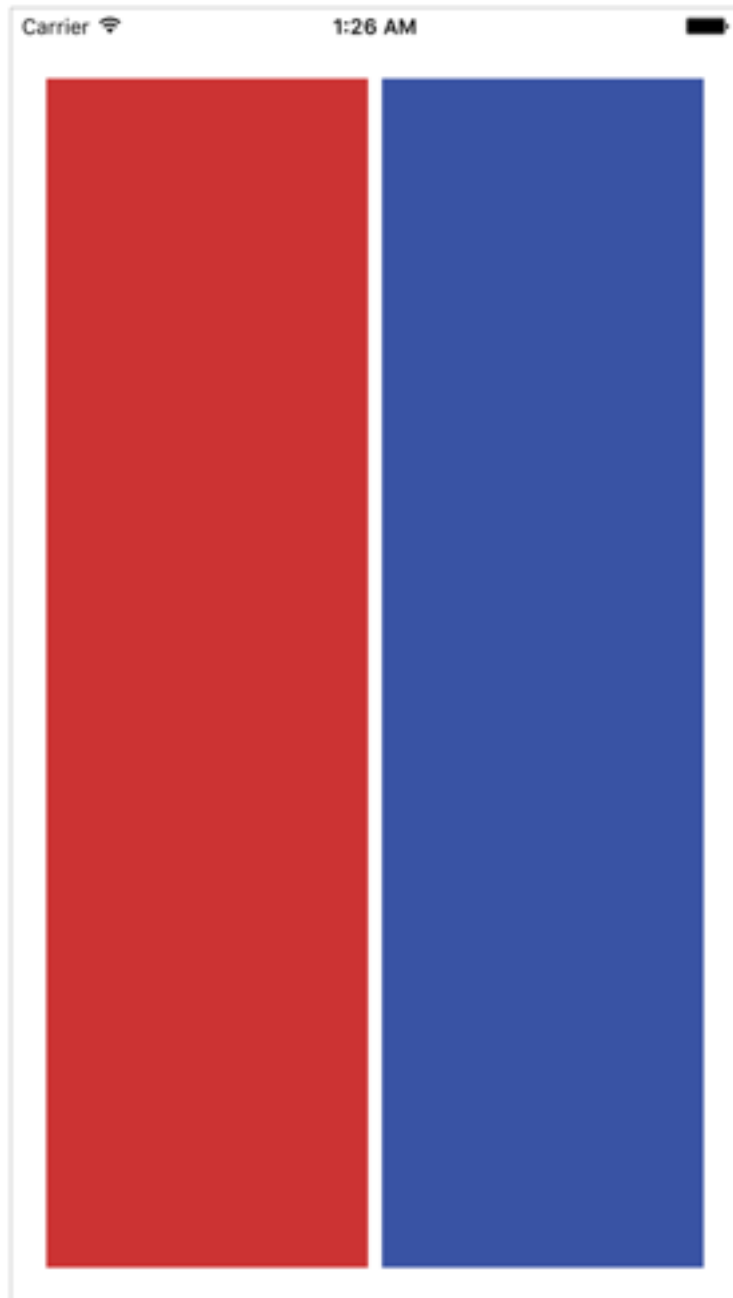
# Interpreting Values

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.
 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, and then set the superview's semantic content attribute 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.

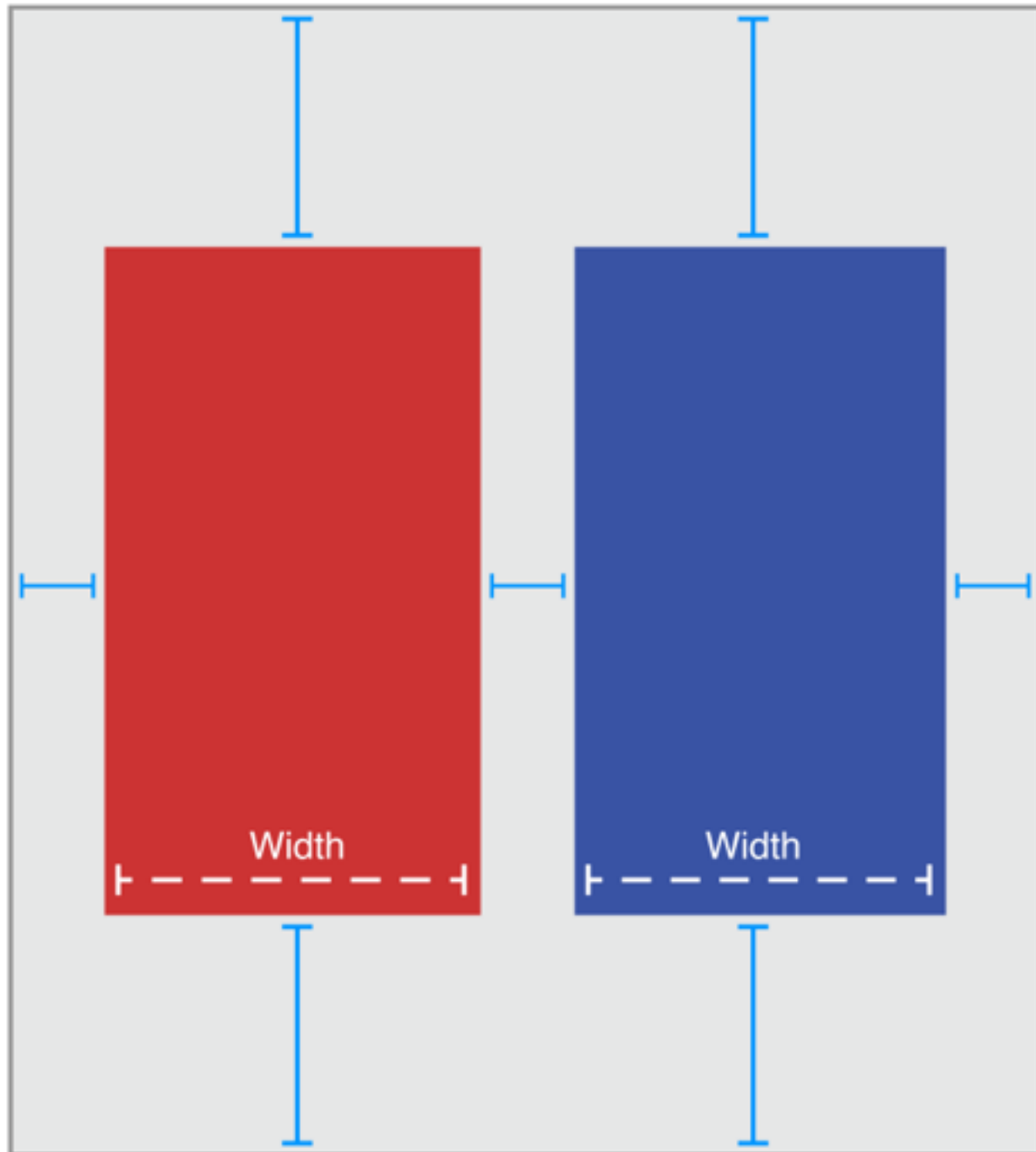
# Nonambiguous, Satisfiable Layouts



# Sample



# Solution 1



// 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

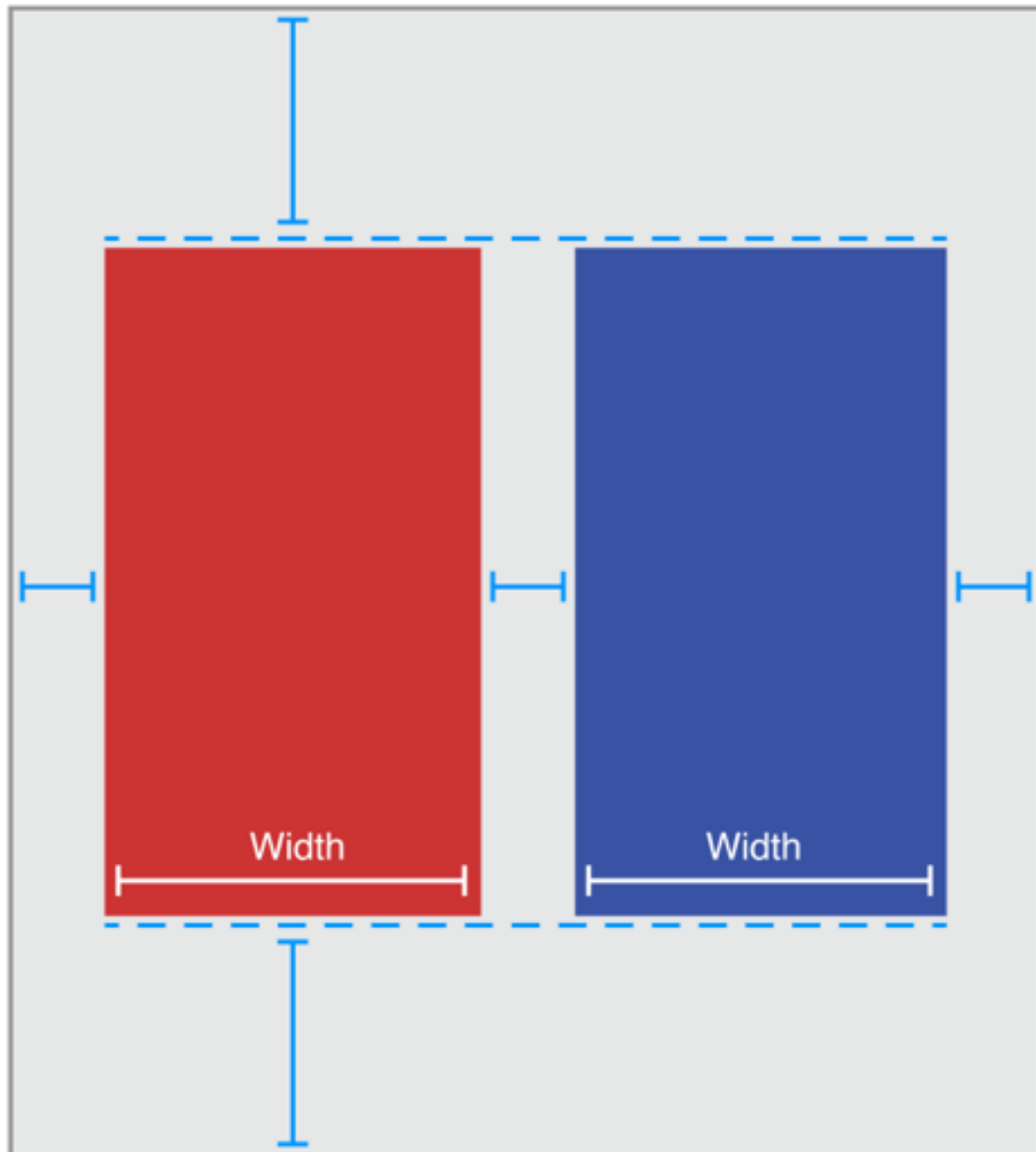
`Red.leading = 1.0 * Superview.leading + 20.0`

`Blue.leading = 1.0 * Red.trailing + 8.0`

`Superview.trailing = 1.0 * Blue.trailing + 20.0`

`Red.width = 1.0 * Blue.width + 0.0`

# Solution 2



// Vertical Constraints

`Red.top = 1.0 * Superview.top + 20.0`

`Superview.bottom = 1.0 * Red.bottom + 20.0`

`Red.top = 1.0 * Blue.top + 0.0`

`Red.bottom = 1.0 * Blue.bottom + 0.0`

//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

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

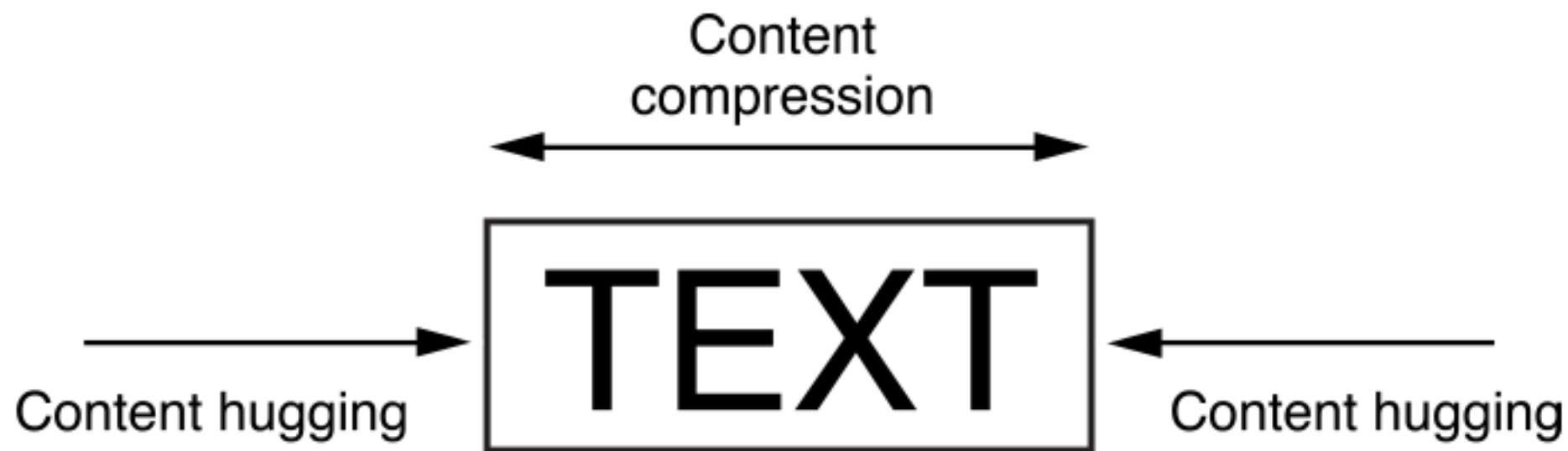
```
// Setting the maximum width  
View.width <= 0.0 * NotAnAttribute + 280.0
```

# Constraint Priorities

- All constraints have a priority between 1 and 1000. Constraints with a priority of 1000 are required. All other constraints are optional.
- When calculating solutions, Auto Layout attempts to satisfy all the constraints in priority order from highest to lowest. If it cannot satisfy an optional constraint, that constraint is skipped and it continues on to the next constraint.
- Priorities should general cluster around the system-defined: low (250), medium (500), high (750), and required (1000).



# Content-hugging and Compression-resistance (CHCR)



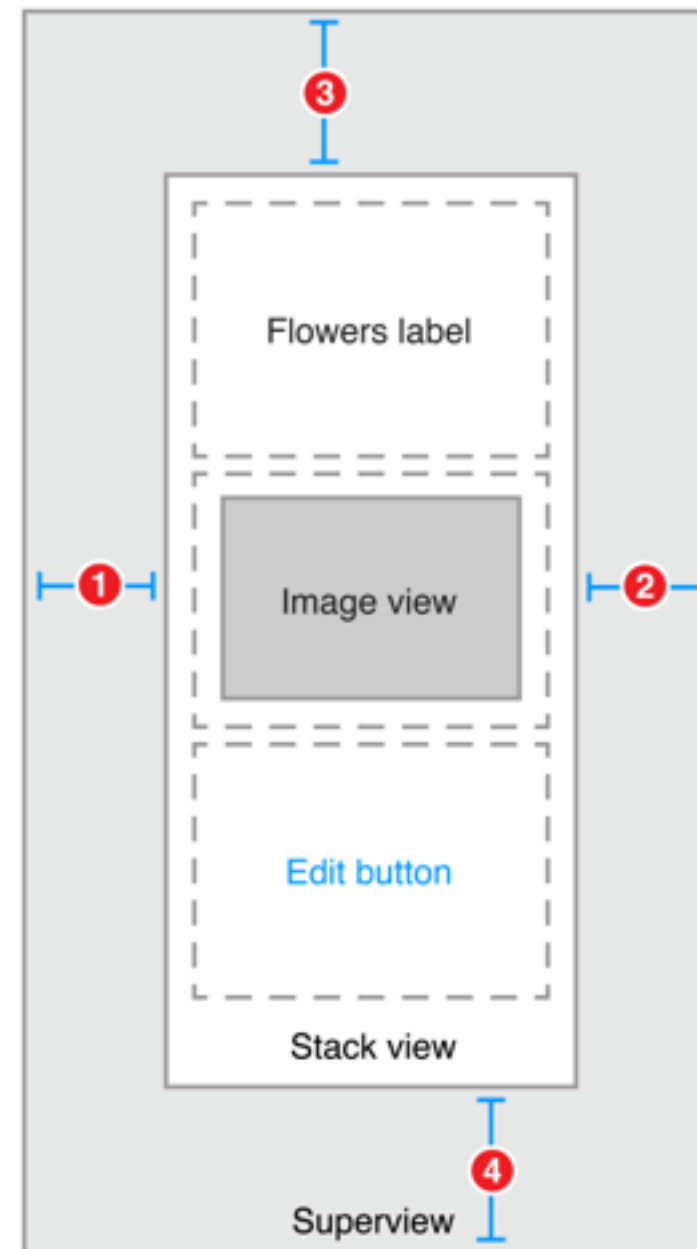
```
// Compression Resistance  
View.height >= 0.0 * NotAnAttribute + IntrinsicHeight  
View.width >= 0.0 * NotAnAttribute + IntrinsicWidth
```

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

# UIStackView

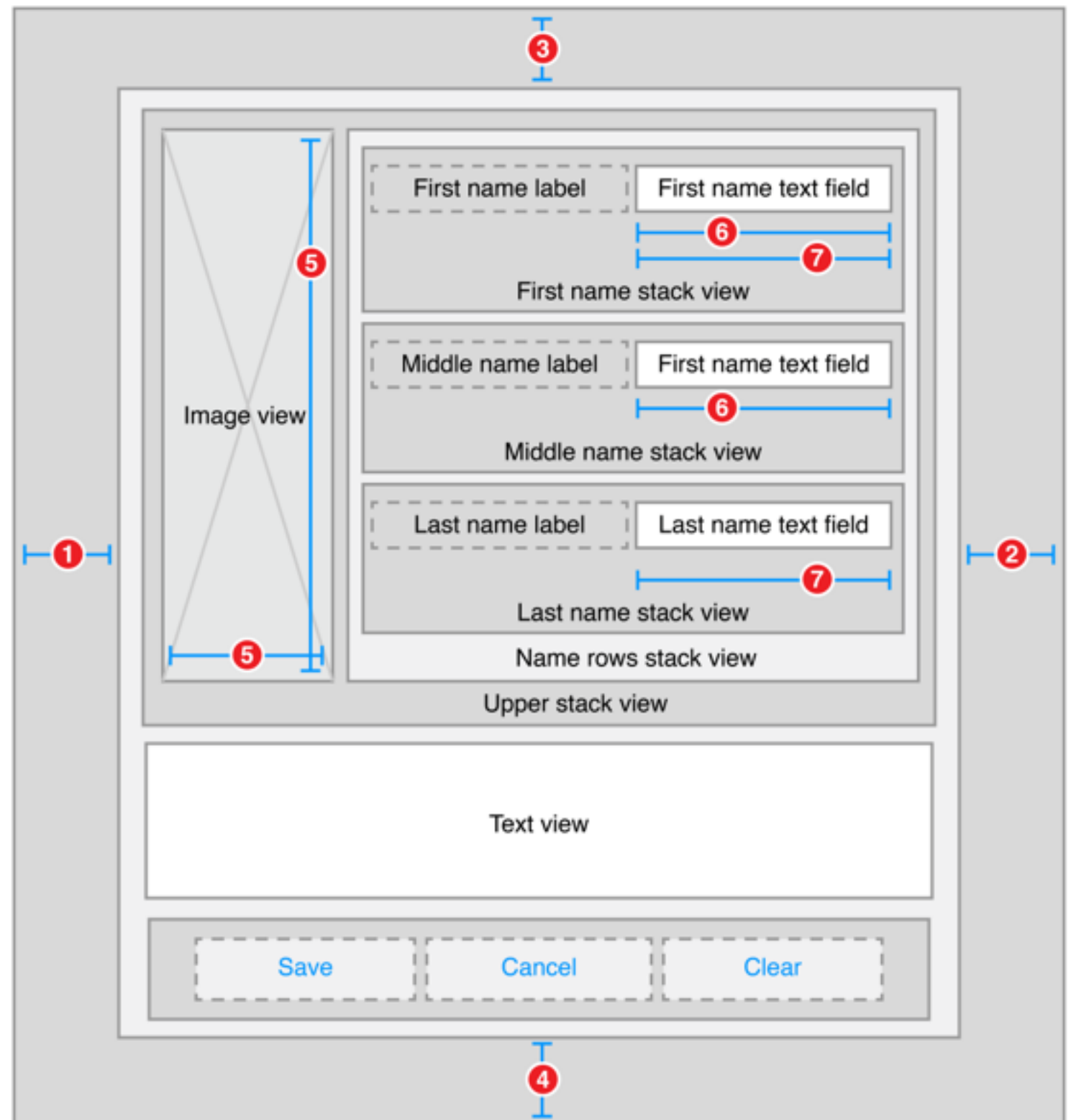
- Stack views provide an easy way to leverage the power of Auto Layout without introducing the complexity of constraints.
- A single stack view defines a row or column of user interface elements.
- The stack view arranges these elements based on its properties:
  - *axis*: defines the stack view's orientation, 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.

# Simple Stack View



Stack	Axis	Alignment	Distribution	Spacing
Stack View	Vertical	Fill	Fill	8

# Nested Stack Views



# 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

- 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.
- 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.

# 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.
    - ➔ After you determine which views are ambiguous, 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

- **hasAmbiguousLayout.** Call this method on a misplaced view. It returns YES if the view's frame is ambiguous. Otherwise, it returns NO.
- **exerciseAmbiguityInLayout.** Call this method on a view with ambiguous layout. This will toggle the system between the possible valid solutions.
- **constraintsAffectingLayoutForAxis:.** Call this method on a view. It returns an array of all the constraints affecting that view along the specified axis.
- **\_autolayoutTrace.** A private method. 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.



# Creating constraints in code



```

let v1 = UIView(frame:CGRectMake(100, 111, 132, 194))
v1.backgroundColor = UIColor(red: 1, green: 0.4, blue: 1, alpha: 1)
let v2 = UIView()
v2.backgroundColor = UIColor(red: 0.5, green: 1, blue: 0, alpha: 1)
let v3 = UIView()
v3.backgroundColor = UIColor(red: 1, green: 0, blue: 0, alpha: 1)
mainview.addSubview(v1)
v1.addSubview(v2)
v1.addSubview(v3)
v2.translatesAutoresizingMaskIntoConstraints = false
v3.translatesAutoresizingMaskIntoConstraints = false
v1.addConstraint(
    NSLayoutConstraint(item: v2,
        attribute: .Leading,
        relatedBy: .Equal,
        toItem: v1,
        attribute: .Leading,
        multiplier: 1, constant: 0)
)
v1.addConstraint(
    NSLayoutConstraint(item: v2,
        attribute: .Trailing,
        relatedBy: .Equal,
        toItem: v1,
        attribute: .Trailing,
        multiplier: 1, constant: 0)
)
v1.addConstraint(
    NSLayoutConstraint(item: v2,
        attribute: .Top,
        relatedBy: .Equal,
        toItem: v1,
        attribute: .Top,
        multiplier: 1, constant: 0)
)

```

```

v2.addConstraint(
    NSLayoutConstraint(item: v2,
        attribute: .Height,
        relatedBy: .Equal,
        toItem: nil,
        attribute: .NotAnAttribute,
        multiplier: 1, constant: 10)
)
v3.addConstraint(
    NSLayoutConstraint(item: v3,
        attribute: .Width,
        relatedBy: .Equal,
        toItem: nil,
        attribute: .NotAnAttribute,
        multiplier: 1, constant: 20)
)
v3.addConstraint(
    NSLayoutConstraint(item: v3,
        attribute: .Height,
        relatedBy: .Equal,
        toItem: nil,
        attribute: .NotAnAttribute,
        multiplier: 1, constant: 20)
)
v1.addConstraint(
    NSLayoutConstraint(item: v3,
        attribute: .Trailing,
        relatedBy: .Equal,
        toItem: v1,
        attribute: .Trailing,
        multiplier: 1, constant: 0)
)
v1.addConstraint(
    NSLayoutConstraint(item: v3,
        attribute: .Bottom,
        relatedBy: .Equal,
        toItem: v1,
        attribute: .Bottom,
        multiplier: 1, constant: 0)
)

```

# Anchor notation

```
NSLayoutConstraint.activateConstraints([
    v2.leadingAnchor.constraintEqualToAnchor(v1.leadingAnchor),
    v2.trailingAnchor.constraintEqualToAnchor(v1.trailingAnchor),
    v2.topAnchor.constraintEqualToAnchor(v1.topAnchor),
    v2.heightAnchor.constraintEqualToConstant(10),
    v3.widthAnchor.constraintEqualToConstant(20),
    v3.heightAnchor.constraintEqualToConstant(20),
    v3.trailingAnchor.constraintEqualToAnchor(v1.trailingAnchor),
    v3.bottomAnchor.constraintEqualToAnchor(v1.bottomAnchor)
])
```



# Visual format notation

```
let d = ["v2":v2,"v3":v3]
NSLayoutConstraint.activateConstraints([
    NSLayoutConstraint.constraintsWithVisualFormat(
        "H:[v2]!", options: [], metrics: nil, views: d),
    NSLayoutConstraint.constraintsWithVisualFormat(
        "V:[v2(10)]", options: [], metrics: nil, views: d),
    NSLayoutConstraint.constraintsWithVisualFormat(
        "H:[v3(20)]!", options: [], metrics: nil, views: d),
    NSLayoutConstraint.constraintsWithVisualFormat(
        "V:[v3(20)]!", options: [], metrics: nil, views: d)
]).flatten().map{$0})
```

# Layout Anchors

```
// Get the superview's layout
let margins = view.layoutMarginsGuide

// Pin the leading edge of myView to the margin's leading edge
myView.leadingAnchor.constraintEqualToAnchor(margins.leadingAnchor).active = true

// Pin the trailing edge of myView to the margin's trailing edge
myView.trailingAnchor.constraintEqualToAnchor(margins.trailingAnchor).active = true

// Give myView a 1:2 aspect ratio
myView.heightAnchor.constraintEqualToAnchor(myView.widthAnchor, multiplier: 2.0)
```

“Auto Layout Guide”

iOS Developer Library

[https://github.com/  
InnovationSpace/  
iOSCourse](https://github.com/InnovationSpace/iOSCourse)