Auto Layout & Adaptivity

4/21’ 16

View Geometry

Coordinates System of Views

Points are used when programming with <u>user coordinate space</u>.

UIKit and CoreGraphics objects use “points”.

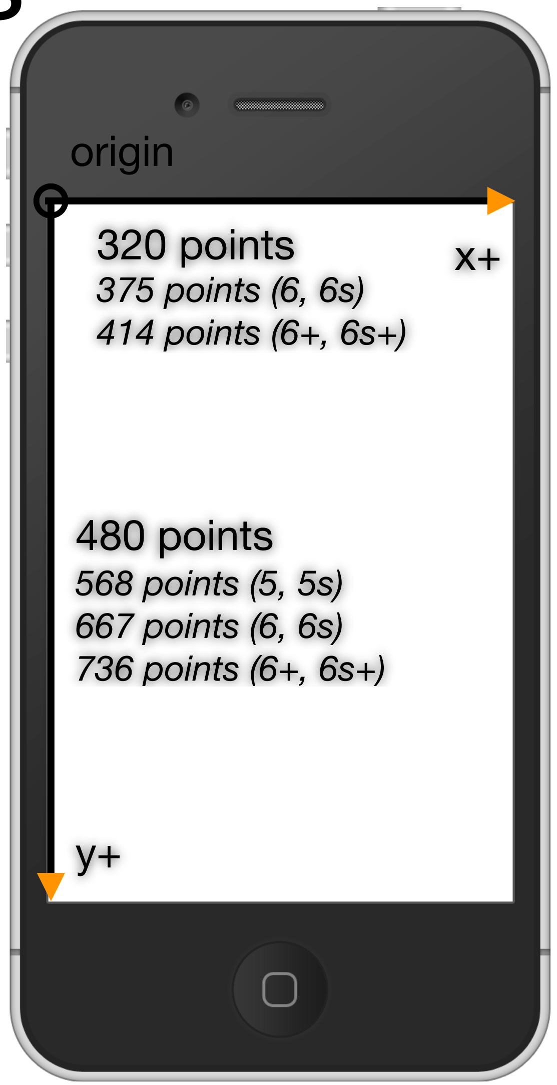
Pixels are used when working with <u>device coordinate space</u>.

Like image drawing and OpenGL ES.

Points to pixels are <u>not always</u> 1:1.

For iPhone 5s, 1 point equals to 2 pixels.

http://www.paintcodeapp.com/news/ultimate-guide-to-iphone-resolutions



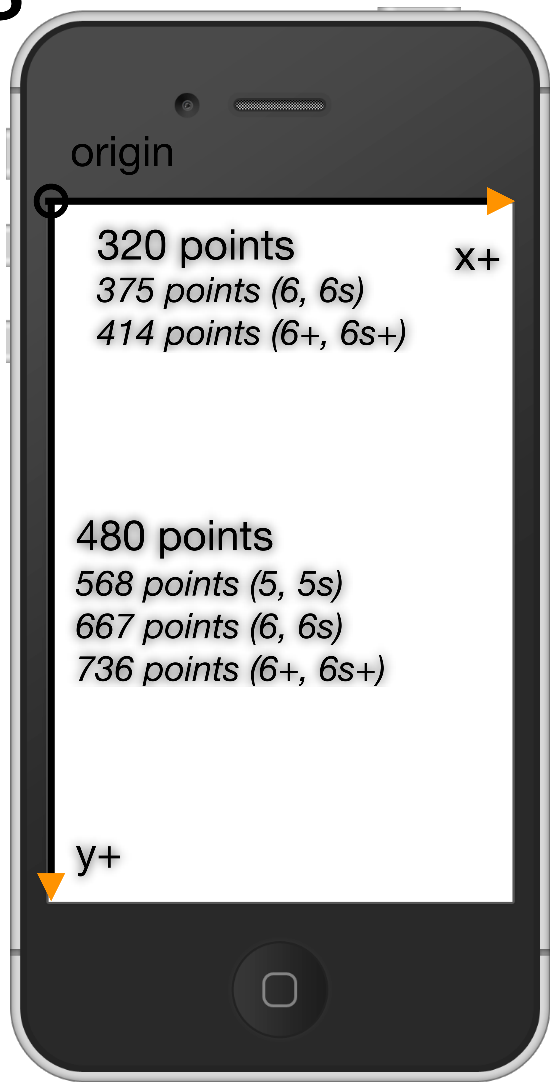
Coordinates System of Views

In UIKit, the origin is at the top-left corner of the view. x+ is from left to right and y+ is from top to bottom.

In AppKit and CoreGraphics, the origin is at "bottom-left" and "y+" is from bottom to top.

For iPad, the width is 768 points and the height is 1,024 points.

For iPad Pro, it's 1,024 points × 1,366 points.



Geometry of Views

a CGRect is a rectangle with <u>origin</u> and <u>size</u>.

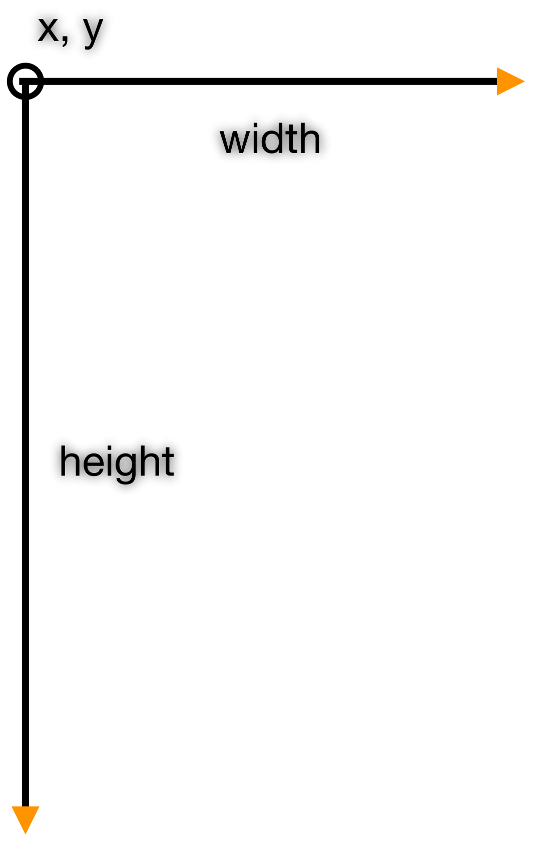
a CGPoint represents points and a CGSize is an area.

CoreGraphics is a C Library.

Hence its APIs are function based.

A view has a frame and a bounds, both are CGRect.

“Frame” is in terms of the superview. “Bounds” is in terms of the view (local). So the “origin” in bounds is (0,0). When “using” a view, use “frame”. When “implementing” (drawing) a view, use “bounds”.



CGRect

UIView Animation

```
UIView.animateWithDuration(1.0) {
    view.frame.origin.x += 50.0
    view.transform = CGAffineTransformMakeRotation(CGFloat(M_PI)/4)
}
```

“frame”, “bounds”, “center”, “transform”, “alpha”, and “backgroundColor” are animatable properties.

Use CGAffineTransform to translate views.

It represents affine transformation matrices in linear algebra.

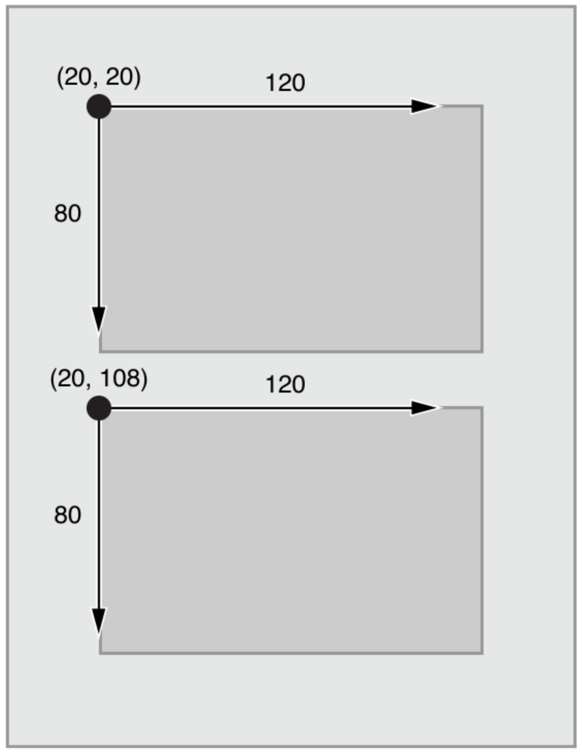
Auto Layout

Auto Layout

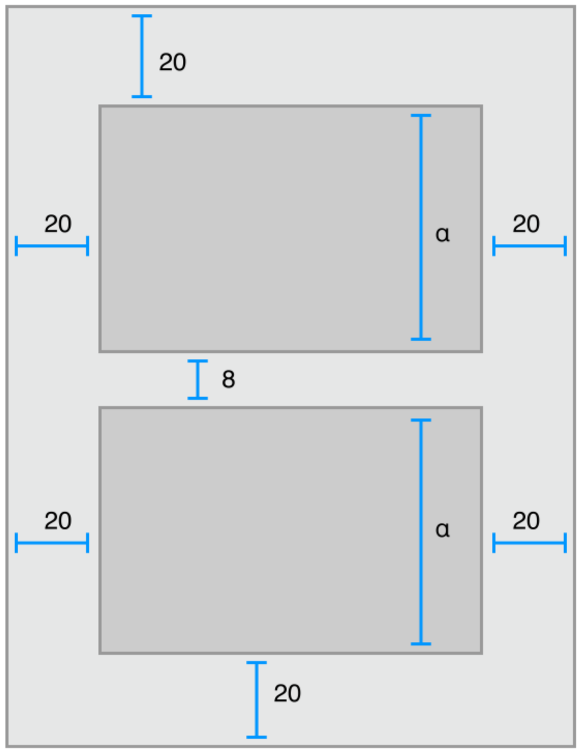
Auto Layout <u>dynamically</u> calculates the size and position of all the views in your view hierarchy, based on constraints placed on those views.

Auto Layout defines your user interface using a series of <u>constraints</u>. Constraints typically represent a relationship between two views. Auto Layout then calculates the size and location of each view based on these constraints. This produces layouts that dynamically respond to both internal and external changes.

Auto Layout



Frame Based



Constraint Based

UIStackView

A stack view defines a row or column of user interface elements.

It provides an easy way to leverage the power of Auto Layout without introducing the complexity of constraints.

Stack views arrange its elements based on its properties.

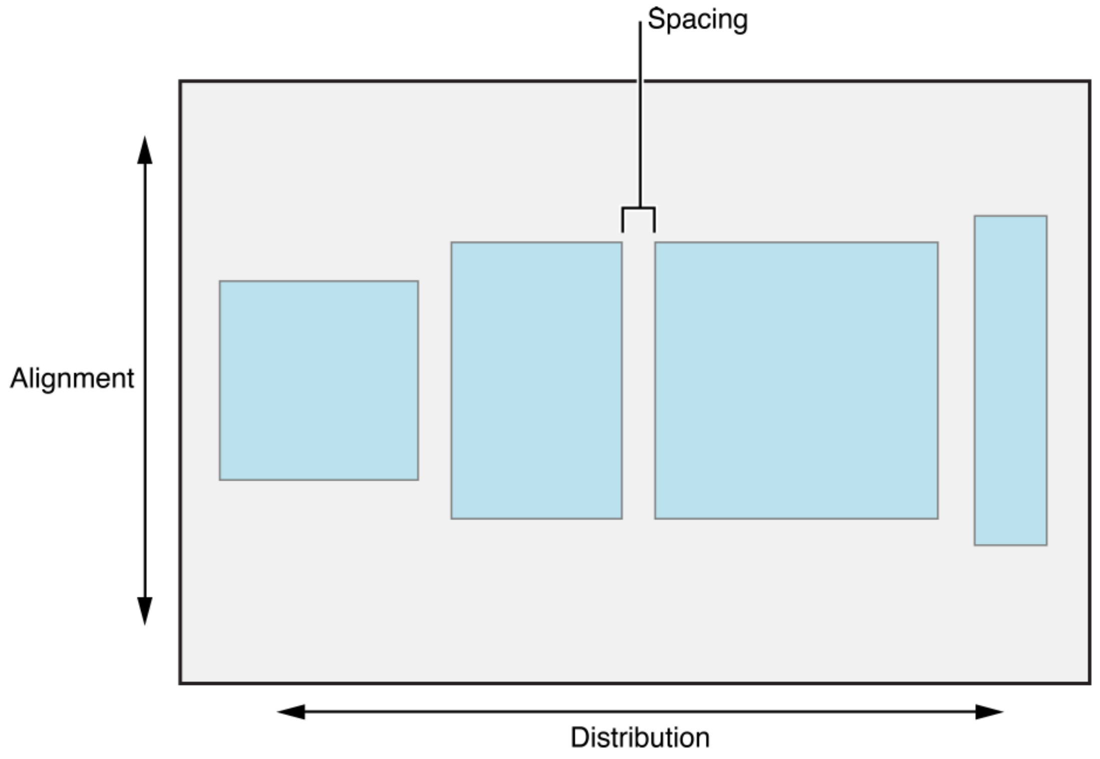
axis: defines orientation, either vertical or horizontal.

distribution: defines the layout of the views along the axis.

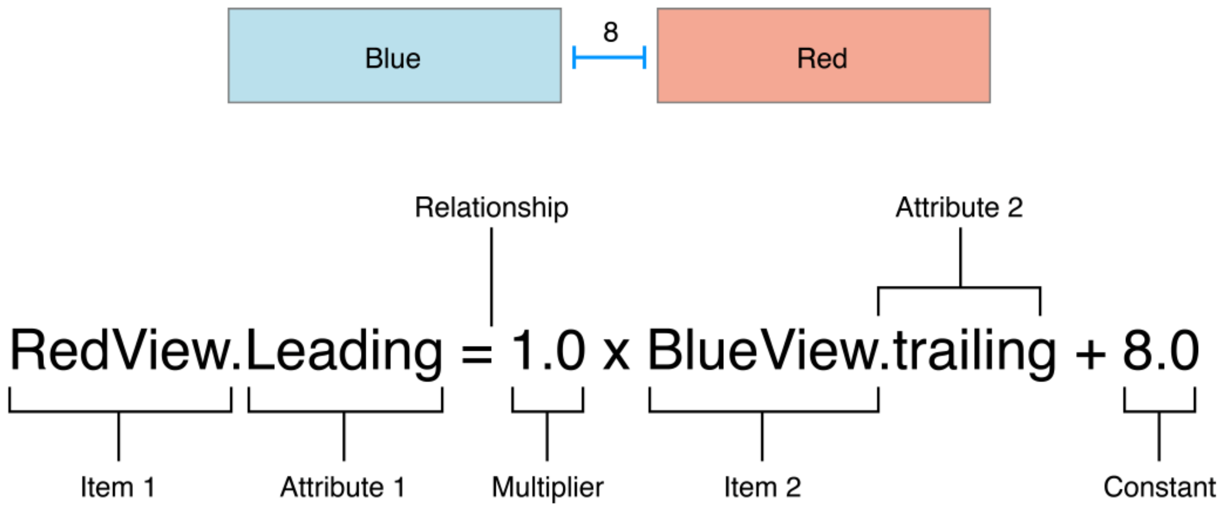
alignment: defines the layout of the views perpendicular to its axis.

spacing: defines the space between adjacent views.

UIStackView



Constraints



The layout of your view hierarchy is defined as a series of linear equations. Each <u>constraint</u> represents a single equation.

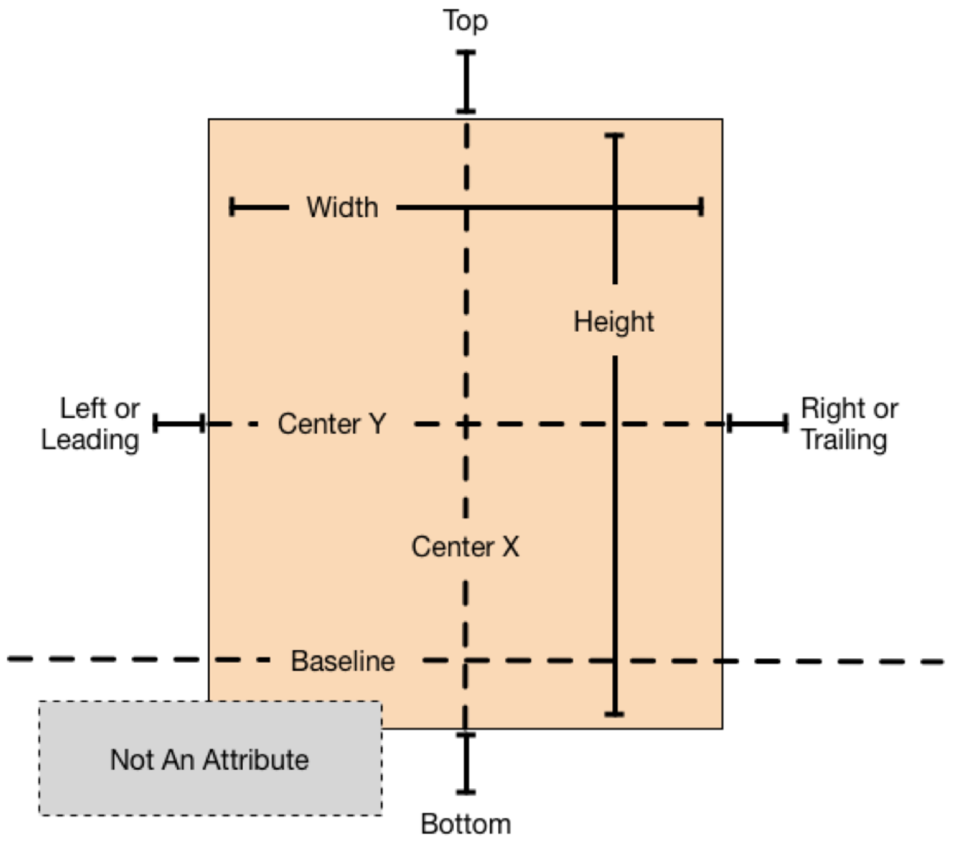
Your goal is to declare a series of equations that has one and only one possible layout result.

Constraints

Check NSLayoutAttribute enum for complete list.

The direction of Leading/Trailing would be flipped when using RTL language.

"Baseline" comes from glyph metrics.



Constraints - Examples

```
// Setting Setting the minimum width with a constant
View.height >= 0.0 * NotAnAttribute + 40.0

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

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

// Give two buttons the same width
Button_1.width = 1.0 * Button_2.width + 0.0

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

// Give a view a constant aspect ratio
View.height = 2.0 * View.width + 0.0
```

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. Auto Layout prints information about the unsatisfiable constraints to the console, and chooses one of them to break.

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

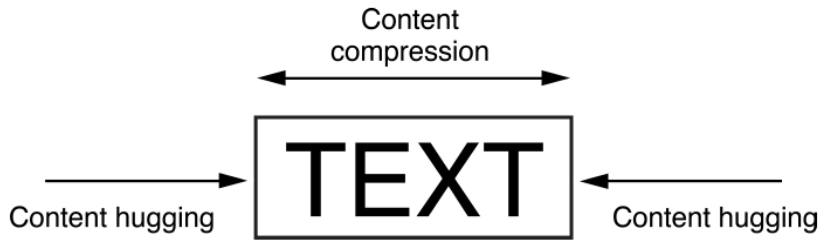
Intrinsic Content Sizes

Some views have a natural size given their current content. This is referred to as their intrinsic content size.
For example, a button’s intrinsic content size is the size of its title and margin.

A label or button’s intrinsic content size is based on the amount of text shown and the font used. For other views, the intrinsic content size is even more complex. For example, an empty image view does not have an intrinsic content size. As soon as you add an image, though, its intrinsic content size is set to the image’s size.

Auto Layout > Constraints > Intrinsic Content Size

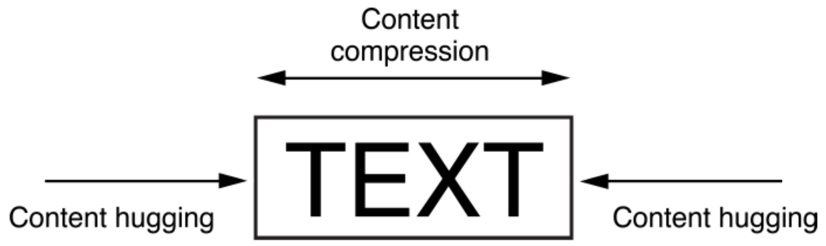
Content Hugging & Compression



Auto Layout represents a view's intrinsic content size using a pair of constraints for each dimension.

The content hugging pulls the view inward to fits the content. The compression resistance pushes the view outward so that it does not clip the content.

Content Hugging & Compression



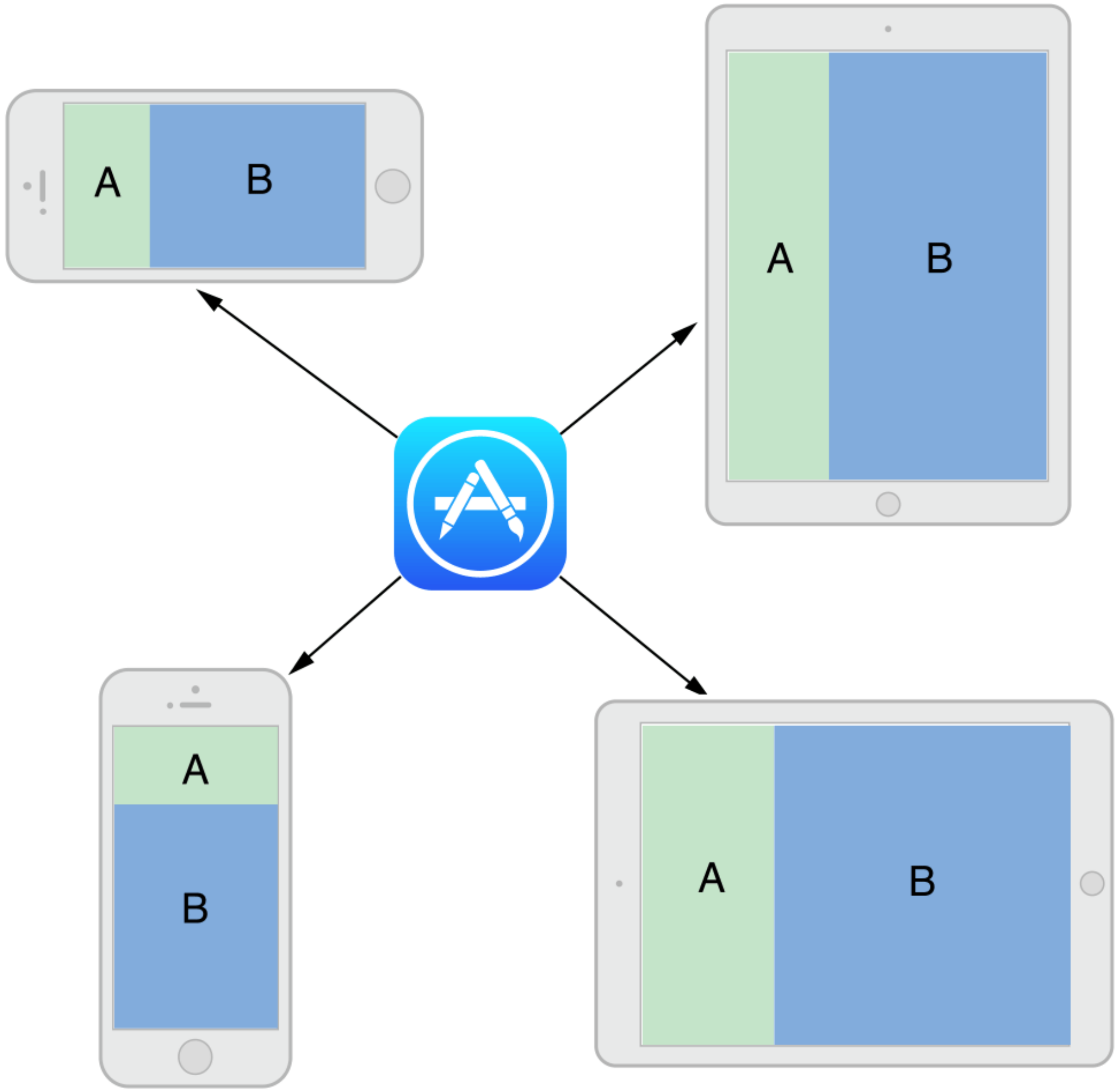
```
// Compression Resistance (default priority is 750)
View.height >= 0.0 * NotAnAttribute + IntrinsicHeight
View.width >= 0.0 * NotAnAttribute + IntrinsicHeight

// Content Hugging (default priority is 250)
View.height <= 0.0 * NotAnAttribute + IntrinsicHeight
View.width <= 0.0 * NotAnAttribute + IntrinsicHeight
```

Adaptivity

Adaptivity

An adaptive interface is one that makes the best use of the available space.
It's similar to the responsive web design concept.

Being adaptive means being able to adjust your content so that it fits well on any iOS device.



Adaptivity

Traits describe the environment in which your view controllers and views must operate. Traits help you make high-level decisions about your interface.

horizontalSizeClass: conveys the general width of your interface.

verticalSizeClass: conveys the general height of your interface.

displayScale: conveys whether the content is displayed on a Retina display or a standard-resolution display.

userInterfaceIdiom: provided for backward compatibility and conveys the type of device on which your app is running.

Adaptivity

```
// Get current trait collection
public var traitCollection: UITraitCollection { get }

// Handle trait collection and size changes
public func willTransitionToTraitCollection(newCollection: UITraitCollection,
    withTransitionCoordinator coordinator: UIViewControllerTransitionCoordinator)
public func viewWillTransitionToSize(size: CGSize,
    withTransitionCoordinator coordinator: UIViewControllerTransitionCoordinator)
public func traitCollectionDidChange(previousTraitCollection: UITraitCollection?)
```

A trait collection describes the iOS interface environment for your app, including traits such as horizontal and vertical size class, display scale, and user interface idiom.

Adaptivity

	Regular Width	Compact Width
Regular Height	All iPads in *both directions*.	All iPhones in *portrait direction*.
Compact Height	iPhone 6 <u>Plus</u> and iPhone 6s <u>Plus</u> in *horizontal direction*.	Other iPhones in *horizontal direction*.

Adaptivity

You could set different Auto Layout constraints and some View properties for each size classes combination.

A constraint or a View is installed when it’s available in current size classes combination.

Adaptivity

Size classes of a view controller may be overridden by its parent view controller.

Don’t reference the screen size directly. You should check the size classes to layout your views for different display size.

Some View Controllers have different behaviors when the size classes is changed.

Like UISplitViewController and UIModalPresentationPopover.

Demo - Auto Layout & Adaptivity