



Auto Layout

Outline

1. Understanding Auto Layout
2. Anatomy of a Constraint
3. Auto Layout Without Constraints
4. Self-Sizing Table View Cells
5. Programmatically Create Constraints

1. Understanding Auto Layout

1. Overview
2. External Changes
3. Internal Changes

1.1 Overview

- ❖ Auto Layout dynamically calculates the size and position of all the views in your view hierarchy, based on constraints placed on those views.
- ❖ For 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.
- ❖ This constraint-based approach to design allows you to build user interfaces that dynamically respond to both internal and external changes.

1.2 External Changes

- ❖ External changes occur when the size or shape of your superview changes. With each change, you must update the layout of your view hierarchy to best use the available space. Here are some common sources of external change:
 - The user resizes the window (OS X).
 - The user enters or leaves Split View on an iPad (iOS).
 - The device rotates (iOS).
 - The active call and audio recording bars appear or disappear (iOS).
 - You want to support different size classes.
 - You want to support different screen sizes.

1.3 Internal Changes

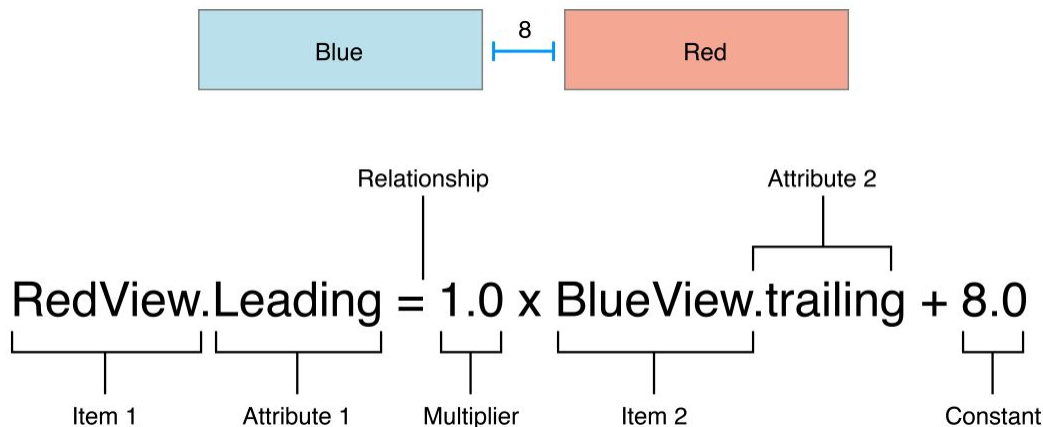
- ❖ Internal changes occur when the size of the views or controls in your user interface change. Here are some common sources of internal change:
 - The content displayed by the app changes.
 - The app supports internationalization.
 - The app supports Dynamic Type (iOS).

2. Anatomy of a Constraint

1. Overview
2. Auto Layout Attributes
3. Sample Equations
4. Equality, Not Assignment
5. Creating Nonambiguous, Satisfiable Layouts
6. Constraint Inequalities
7. Constraint Priorities
8. Intrinsic Content Size
9. Interpreting Values

2.1 Overview

- ❖ The layout of your view hierarchy is defined as a series of linear equations. Each constraint represents a single equation. Your goal is to declare a series of equations that has one and only one possible solution. A sample equation is shown below:

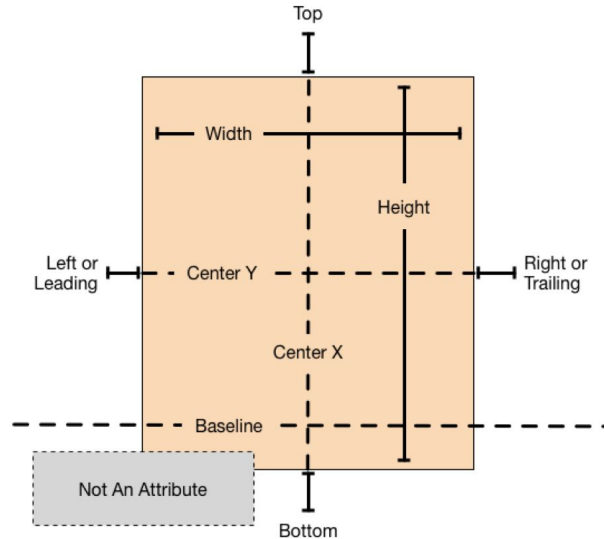


2.1 Overview (cont)

- ❖ Most constraints define a relationship between two items in our user interface. These items can represent either views or layout guides.
- ❖ Constraints can also define the relationship between two different attributes of a single item, for example, setting an aspect ratio between an item's height and width.
- ❖ You can also assign constant values to an item's height or width.
- ❖ When working with constant values, the second item is left blank, the second attribute is set to `Not An Attribute`, and the multiplier is set to `0.0`.

2.2 Auto Layout Attributes

- ❖ In Auto Layout, the attributes define a feature that can be constrained. In general, this includes the four edges (leading, trailing, top, and bottom), as well as the height, width, and vertical and horizontal centers. Text items also have one or more baseline attributes.



2.3 Sample Equations

- ❖ The wide range of parameters and attributes available to these equations lets you create many different types of constraints. You can define the space between views, align the edge of views, define the relative size of two views, or even define a view's aspect ratio. However, not all attributes are compatible.
- ❖ There are two basic types of attributes.
 - ➔ Size attributes (for example, Height and Width) are used to specify how large an item is, without any indication of its location. and location attributes .
 - ➔ Location attributes (for example, Leading, Left, and Top) are used to specify the location of an item relative to something else. However, they carry no indication of the item's size.

2.3 Sample Equations (cont)

- ❖ With these differences in mind, the following rules apply:
 - You cannot constrain a size attribute to a location attribute.
 - You cannot assign constant values to location attributes.
 - You cannot use a nonidentity multiplier (a value other than 1.0) with location attributes.
 - For location attributes, you cannot constrain vertical attributes to horizontal attributes.
 - For location attributes, you cannot constrain Leading or Trailing attributes to Left or Right attributes.

2.3 Sample Equations (cont)

- ❖ Sample equations for common constraints:

```
// Setting a constant height
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
```

2.4 Equality, Not Assignment

- ❖ It's important to note that the equations shown in Note represent equality, not assignment.
- ❖ When Auto Layout solves these equations, it does not just assign the value of the right side to the left. Instead, it calculates the value for both attribute 1 and attribute 2 that makes the relationship true. This means we can often freely reorder the items in the equation.

```
// 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
```

2.4 Equality, Not Assignment (cont)

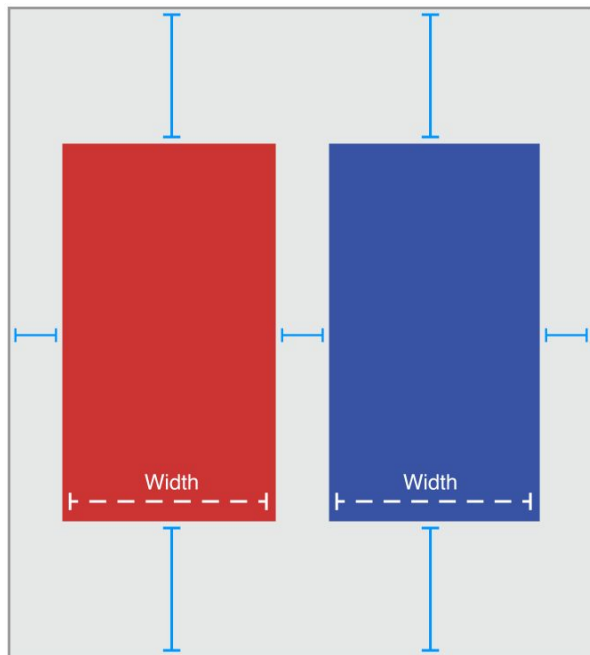
- ❖ You will find that Auto Layout frequently provides multiple ways to solve the same problem. Ideally, you should choose the solution that most clearly describes your intent.
- ❖ However, different developers will undoubtedly disagree about which solution is best. Here, being consistent is better than being right. You will experience fewer problems in the long run if you choose an approach and always stick with it.
- ❖ For example, this guide uses the following rules of thumb:
 - Whole number multipliers are favored over fractional multipliers.
 - Positive constants are favored over negative constants.
 - Wherever possible, views should appear in layout order: leading to trailing, top to bottom.

2.5 Creating Nonambiguous, Satisfiable Layouts

- ❖ When using Auto Layout, the goal is to provide a series of equations that have one and only one possible solution. Ambiguous constraints have more than one possible solution. Unsatisfiable constraints don't have valid solutions.
- ❖ In general, the constraints must define both the size and the position of each view. Assuming the superview's size is already set (for example, the root view of a scene in iOS), a nonambiguous, satisfiable layout requires two constraints per view per dimension (not counting the superview).
- ❖ However, you have a wide range of options when it comes to choosing which constraints you want to use.

2.5 Creating Nonambiguous, Satisfiable Layouts (cont)

- ❖ The following illustration shows one straightforward solution:

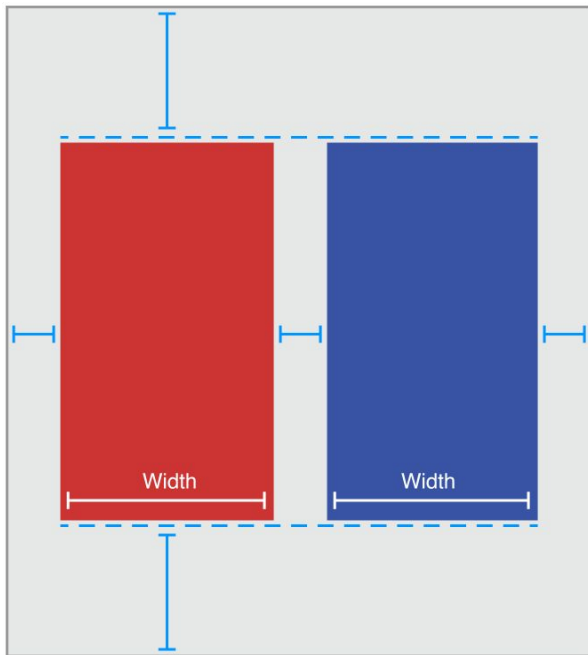


```
// Vertical Constraints
Red.top = 1.0 * Superview.top + 20.0
Superview.bottom = 1.0 * Red.bottom + 20.0
Blue.top = 1.0 * Superview.top + 20.0
Superview.bottom = 1.0 * Blue.bottom + 20.0

// 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
```

2.5 Creating Nonambiguous, Satisfiable Layouts (cont)

- ❖ Still, this is not the only possible solution. Here is an equally valid approach:



```
// 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
```

2.6 Constraint Inequalities

- ❖ Constraints can represent inequalities as well. Specifically, the constraint's relationship can be equal to, greater than or equal to, or less than or equal to.
- ❖ For example, you can use constraints to define the minimum or maximum size for a view:

```
// Setting the minimum width  
View.width ≥ 0.0 * NotAnAttribute + 40.0  
  
// Setting the maximum width  
View.width ≤ 0.0 * NotAnAttribute + 280.0
```

2.6 Constraint Inequalities (cont)

- ❖ As soon as you start using inequalities, the two constraints per view per dimension rule breaks down. You can always replace a single equality relationship with two inequalities.
- ❖ The inverse is not always true, because two inequalities are not always equivalent to a single equals relationship.

```
// 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
```

2.7 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 the constraints to break. It then recalculates the solution without the broken constraint.
- ❖ 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.
- ❖ 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.

2.8 Intrinsic Content Size

- ❖ So far, all of the examples have used constraints to define both the view's position and its size.
- ❖ However, 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 plus a small margin.

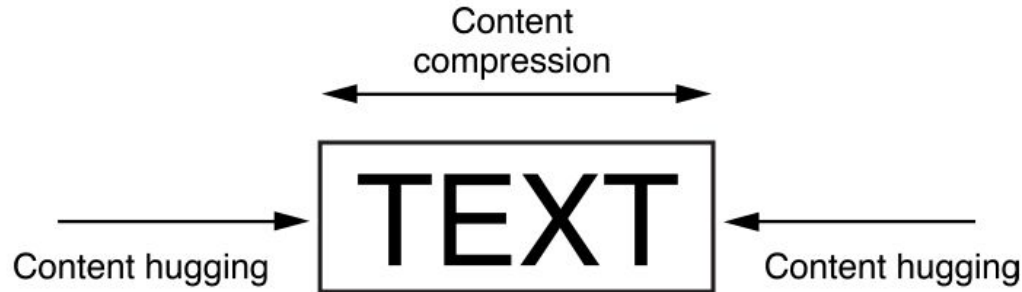
2.8 Intrinsic Content Size (cont)

- ❖ Not all views have an intrinsic content size. For views that do, the intrinsic content size can define the view's height, its width, or both.

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.

2.8 Intrinsic Content Size (cont)

- ❖ Auto Layout represents a view's intrinsic content size using a pair of constraints for each dimension. The content hugging pulls the view inward so that it fits snugly around the content. The compression resistance pushes the view outward so that it does not clip the content.



2.8 Intrinsic Content Size (cont)

- ❖ Here, the `IntrinsicHeight` and `IntrinsicWidth` constants represent the height and width values from the view's intrinsic content size.

```
// 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
```

2.8 Intrinsic Content Size (cont)

- ❖ Each of these constraints can have its own priority. By default, views use a 250 priority for their content hugging, and a 750 priority for their compression resistance. Therefore, it's easier to stretch a view than it is to shrink it.
- ❖ Whenever possible, use the view's intrinsic content size in your layout.
 - ➔ It lets your layout dynamically adapt as the view's content changes.
 - ➔ It also reduces the number of constraints you need to create a non-ambiguous, non-conflicting layout, but you will need to manage the view's content-hugging and compression-resistance (CHCR) priorities.

2.8 Intrinsic Content Size (cont)

Here are some guidelines for handling intrinsic content sizes:

- ❖ When stretching a series of views to fill a space, if all the views have an identical content-hugging priority, the layout is ambiguous. Auto Layout doesn't know which view should be stretched.
- ❖ Odd and unexpected layouts often occur when views with invisible backgrounds (like buttons or labels) are accidentally stretched beyond their intrinsic content size. The actual problem may not be obvious, because the text simply appears in the wrong location. To prevent unwanted stretching, increase the content-hugging priority.






2.8 Intrinsic Content Size (cont)

Here are some guidelines for handling intrinsic content sizes:



- ❖ Baseline constraints work only with views that are at their intrinsic content height. If a view is vertically stretched or compressed, the baseline constraints no longer align properly.
- ❖ Some views, like switches, should always be displayed at their intrinsic content size. Increase their CHCR priorities as needed to prevent stretching or compressing.
- ❖ Avoid giving views required CHCR priorities. It's usually better for a view to be the wrong size than for it to accidentally create a conflict. If a view should always be its intrinsic content size, consider using a very high priority (999) instead. This approach generally keeps the view from being stretched or compressed but still provides an emergency pressure valve, just in case your view is displayed in an environment that is bigger or smaller than you expected.

2.9 Interpreting Values





- ❖ Values in Auto Layout are always in points. However, the exact meaning of 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.

2.9 Interpreting Values (cont)

Auto Layout Attributes	Value	Notes
 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.

2.9 Interpreting Values (cont)

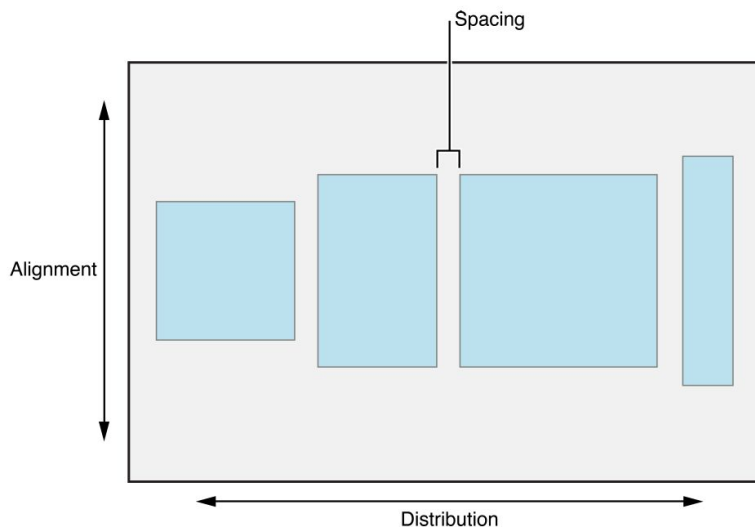
Auto Layout Attributes	Value	Notes
 Left  Right	<p>The values increase as you move to the right.</p>	<p>These attributes can be combined only with Left, Right, and Center X attributes. Avoid using Left and Right attributes. Use Leading and Trailing instead. This allows the layout to adapt to the view's reading direction.</p>
 Center X  Center Y	<p>The interpretation is based on the other attribute in the equation.</p>	<p>Center X can be combined with Center X, Leading, Trailing, Right, and Left attributes.</p> <p>Center Y can be combined with Center Y, Top, Bottom, and Baseline attributes.</p>

3. Auto Layout Without Constraints

- ❖ 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**: (`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.

3. Auto Layout Without Constraints (cont)

- ❖ The stack view manages the layout of all the views in its `arrangedSubviews` property. These views are arranged along the stack view's axis, based on their order in the `arrangedSubviews` array.



4. Self-Sizing Table View Cells

- ❖ In iOS, you can use Auto Layout to define the height of a table view cell; however, the feature is not enabled by default.
- ❖ Normally, a cell's height is determined by the table view delegate's `tableView:heightForRowAtIndexPath:` method. To enable self-sizing table view cells, you must set the table view's `rowHeight` property to `UITableViewAutomaticDimension`. You must also assign a value to the `estimatedRowHeight` property. As soon as both of these properties are set, the system uses Auto Layout to calculate the row's actual height.

```
tableView.estimatedRowHeight = 85.0  
tableView.rowHeight = UITableViewAutomaticDimension
```

4. Self-Sizing Table View Cells (cont)

- ❖ Try to make the estimated row height as accurate as possible. The system calculates items such as the scroll bar heights based on these estimates. The more accurate the estimates, the more seamless the user experience becomes.
- ❖ When working with table view cells, you cannot change the layout of the predefined content (for example, the `titleLabel`, `detailTextLabel`, and `imageView` properties). The following constraints are supported:
 - Constraints that position your subview relative to the cell's content view.
 - Constraints that position your subview relative to the cell's bounds.
 - Constraints that position your subview relative to the predefined content.

5. Programmatically Creating Constraints

1. NSLayoutAnchor
2. NSLayoutConstraint

5.1 NSLayoutAnchor

- ❖ The `NSLayoutAnchor` class provides a fluent interface for creating constraints. To use this API, access the anchor properties on the items you want to constrain.

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

// Pin the leading edge of myView to the margin's leading edge
myView.leadingAnchor.constraint(equalTo: margins.leadingAnchor).isActive = true

// Pin the trailing edge of myView to the margin's trailing edge
myView.trailingAnchor.constraint(equalTo: margins.trailingAnchor).isActive = true

// Give myView a 1:2 aspect ratio
myView.heightAnchor.constraint(equalTo: myView.widthAnchor, multiplier: 2.0).isActive = true
```

5.2 NSLayoutConstraint

- ❖ You can also create constraints directly using the `NSLayoutConstraint` class's `constraintWithItem:attribute:relatedBy toItem:attribute:multipplier:constant:` convenience method.
- ❖ Unlike the approach taken by the layout anchor API, you must specify a value for each parameter, even if it doesn't affect the layout. The end result is a considerable amount of boilerplate code, which is usually harder to read.

```
NSLayoutConstraint(item: myView, attribute: .leading, relatedBy: .equal, toItem: view, attribute: .leadingMargin,
    multiplier: 1.0, constant: 0.0).isActive = true

NSLayoutConstraint(item: myView, attribute: .trailing, relatedBy: .equal, toItem: view, attribute: .trailingMargin,
    multiplier: 1.0, constant: 0.0).isActive = true

NSLayoutConstraint(item: myView, attribute: .height, relatedBy: .equal, toItem: myView, attribute: .width,
    multiplier: 2.0, constant: 0.0).isActive = true
```

Question & Answer?



