Class

UIView

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

Language

Swift | Objective-C

SDKs

iOS 2.0+

tvOS 2.0+

On This Page

Overview

Symbols

Relationships

Overview

At runtime, a view object handles the rendering of any content in its area and also handles any interactions with that content. The `UIView` class itself provides basic behavior for filling its rectangular area with a background color. More sophisticated content can be presented by subclassing `UIView` and implementing the necessary drawing and event-handling code yourself. The UIKit framework also includes a set of standard subclasses that range from simple buttons to complex tables and can be used as-is. For example, a `UILabel` object draws a text string and a `UIImageView` object draws an image.

Because view objects are the main way your application interacts with the user, they have a number of responsibilities. Here are just a few:

- Drawing and animation

    - Views draw content in their rectangular area using technologies such as UIKit, Core Graphics, and OpenGL ES.
    - Some view Declared property can be animated to new values.
- Layout and subview management
    - A view may contain zero or more subviews.
    - Each view defines its own default resizing behavior in relation to its parent view.
    - A view can define the size and position of its subviews as needed.
- Event handling
    - A view is a responder and can handle touch events and other events defined by the `UIResponder` class.
    - Views can use the `addGestureRecognizer(_:)` method to install gesture recognizers to handle common gestures.

Views can embed other views and create sophisticated visual hierarchies. This creates a parent-child relationship between the view being embedded (known as the **subview**) and the parent view doing the embedding (known as the **superview**). Normally, a subview's visible area is not clipped to the bounds of its superview, but in iOS you can use the `clipsToBounds` property to alter that behavior. A parent view may contain any number of subviews but each subview has only one superview, which is responsible for positioning its subviews appropriately.

The geometry of a view is defined by its `frame`, `bounds`, and `center` properties. The `frame` defines the origin and dimensions of the view in the coordinate system of its superview and is commonly used during layout to adjust the size or position of the view. The `center` property can be used to adjust the position of the view without changing its size. The `bounds` defines the internal dimensions of the view as it sees them and is used almost exclusively in custom drawing code. The

size portion of the `frame` and `bounds` rectangles are coupled together so that changing the size of either rectangle updates the size of both.

For detailed information about how to use the `UIView` class, see View Programming Guide for iOS.

> **Note**
>
> In iOS 2.x, the maximum size of a `UIView` object is 1024 x 1024 points. In iOS 3.0 and later, views are no longer restricted to this maximum size but are still limited by the amount of memory they consume. It is in your best interests to keep view sizes as small as possible. Regardless of which version of iOS is running, you should consider tiling any content that is significantly larger than the dimensions of the screen.

Creating a View

To create a view programmatically, you can use code like the following:

```
CGRect  viewRect = CGRectMake(10, 10, 100, 100);
UIView* myView = [[UIView alloc] initWithFrame:viewRect];
```

This code creates the view and positions it at the point (10, 10) in its superview's coordinate system (once it is added to that superview). To add a subview to another view, you use the `addSubview(_:)` method. In iOS, sibling views may overlap each other without any issues, allowing complex view placement. The `addSubview(_:)` method places the specified view on top of other siblings. You can specify the relative z-order of a subview by adding it using the `insertSubview(_:aboveSubview:)` and `insertSubview(_:belowSubview:)` methods. You can also exchange the position of already added subviews using the `exchangeSubview(at:withSubviewAt:)` method.

When creating a view, it is important to assign an appropriate value to the `autoresizingMask` property to ensure the view resizes correctly. View resizing primarily occurs when the orientation of your application's interface changes but it may happen at other times as well. For example, calling the `setNeedsLayout()` method forces your view to update its layout.

The View Drawing Cycle

View drawing occurs on an as-needed basis. When a view is first shown, or when all or part of it becomes visible due to layout changes, the system asks the view to draw its contents. For views that contain custom content using UIKit or Core Graphics, the system calls the view's `draw(_:)` method. Your implementation of this method is responsible for drawing the view's content into the current graphics context, which is set up by the system automatically prior to calling this method. This creates a static visual representation of your view's content that can then be displayed on the screen.

When the actual content of your view changes, it is your responsibility to notify the system that your view needs to be redrawn. You do this by calling your view's `setNeedsDisplay()` or `setNeedsDisplay(_:)` method of the view. These methods let the system know that it should update the view during the next drawing cycle. Because it waits until the next drawing cycle to update the view, you can call these methods on multiple views to update them at the same time.

> **Note**
>
> If you are using OpenGL ES to do your drawing, you should use the `GLKView` class instead of subclassing `UIView`. For more information about how to draw using OpenGL ES, see OpenGL ES Programming Guide for iOS.

For detailed information about the view drawing cycle and the role your views have in this cycle, see View Programming Guide for iOS.

Animations

Changes to several view properties can be animated—that is, changing the property creates an animation that conveys the change to the user over a short period of time. The `UIView` class does most of the work of performing the actual animations but you must still indicate which property changes you want to be animated. There are two different ways to initiate animations:

- In iOS 4 and later, use the block-based animation methods. (Recommended)
- Use the begin/commit animation methods.

The block-based animation methods (such as `animate(withDuration:animations:)`) greatly simplify the creation of animations. With one method call, you specify the animations to be performed and the options for the animation. However, block-based animations are available only in iOS 4 and later. If your application runs on earlier versions of iOS, you must use the `beginAnimations(_:context:)` and `commitAnimations()` class methods to mark the beginning and ending of your animations.

The following properties of the `UIView` class are animatable:

- `frame`
- `bounds`
- `center`
- `transform`
- `alpha`
- `backgroundColor`

For more information about how to configure animations, see View Programming Guide for iOS.

Threading Considerations

Manipulations to your application's user interface must occur on the main thread. Thus, you should always call the methods of the `UIView` class from code running in the main thread of your application. The only time this may not be strictly necessary is when creating the view object itself but all other manipulations should occur on the main thread.

Subclassing Notes

The `UIView` class is a key subclassing point for visual content that also requires user interactions. Although there are many good reasons to subclass `UIView`, it is recommended that you do so only when the basic `UIView` class or the standard system views do not provide the capabilities that you need. Subclassing requires more work on your part to implement the view and to tune its performance.

For information about ways to avoid subclassing, see *Alternatives to Subclassing*.

Methods to Override

When subclassing `UIView`, there are only a handful of methods you should override and many methods that you might override depending on your needs. Because `UIView` is a highly configurable class, there are also many ways to implement sophisticated view behaviors without overriding custom methods, which are discussed in the Alternatives to Subclassing section. In the meantime, the following list includes the methods you might consider overriding in your `UIView` subclasses:

- Initialization:
    - `init(frame:)` - It is recommended that you implement this method. You can also implement custom initialization methods in addition to, or instead of, this method.

    - `init(coder:)` - Implement this method if you load your view from an Interface Builder nib file and your view requires custom initialization.

    - `layerClass` Use this property only if you want your view to use a different Core Animation layer for its backing store. For example, if your view uses tiling to display a large scrollable area, you might want to set the property to the `CATiledLayer` class.

- Drawing and printing:

    - `draw(_:)` - Implement this method if your view draws custom content. If your view does not do any custom drawing, avoid overriding this method.

    - `draw(_:for:)` - Implement this method only if you want to draw your view's content differently during printing.

- Constraints:

    - `requiresConstraintBasedLayout` Use this property if your view class requires constraints to work properly.

    - `updateConstraints()` - Implement this method if your view needs to create custom constraints between your subviews.

    - `alignmentRect(forFrame:)`, `frame(forAlignmentRect:)` - Implement these methods to override how your views are aligned to other views.

- Layout:

    - `sizeThatFits(_:)` - Implement this method if you want your view to have a different default size than it normally would during resizing operations. For example, you might use this method to prevent your view from shrinking to the point where subviews cannot be displayed correctly.

    - `layoutSubviews()` - Implement this method if you need more precise control over the layout of your subviews than either the constraint or autoresizing behaviors provide.
    - `didAddSubview(_:)`, `willRemoveSubview(_:)` - Implement these methods as needed to track the additions and removals of subviews.
    - `willMove(toSuperview:)`, `didMoveToSuperview()` - Implement these methods as needed to track the movement of the current view in your view hierarchy.
    - `willMove(toWindow:)`, `didMoveToWindow()` - Implement these methods as needed to track the movement of your view to a different window.
- Event Handling:
    - `touchesBegan(_:with:)`, `touchesMoved(_:with:)`, `touchesEnded(_:with:)`, `touchesCancelled(_:with:)` - Implement these methods if you need to handle touch events directly. (For gesture-based input, use gesture recognizers.)
    - `gestureRecognizerShouldBegin(_:)` - Implement this method if your view handles touch events directly and might want to prevent attached gesture recognizers from triggering additional actions.

Alternatives to Subclassing

Many view behaviors can be configured without the need for subclassing. Before you start overriding methods, consider whether modifying the following properties or behaviors would provide the behavior you need.

- `addConstraint(_:)` - Define automatic layout behavior for the view and its subviews.
- `autoresizingMask` - Provides automatic layout behavior when the superview's frame changes. These behaviors can be combined with constraints.

- `contentMode` - Provides layout behavior for the view's content, as opposed to the `frame` of the view. This property also affects how the content is scaled to fit the view and whether it is cached or redrawn.

- `isHidden` or `alpha` - Change the transparency of the view as a whole rather than hiding or applying alpha to your view's rendered content.

- `backgroundColor` - Set the view's color rather than drawing that color yourself.

- Subviews - Rather than draw your content using a `draw(_:)` method, embed image and label subviews with the content you want to present.

- Gesture recognizers - Rather than subclass to intercept and handle touch events yourself, you can use gesture recognizers to send an Target-Action to a target object.

- Animations - Use the built-in animation support rather than trying to animate changes yourself. The animation support provided by Core Animation is fast and easy to use.

- Image-based backgrounds - For views that display relatively static content, consider using a `UIImageView` object with gesture recognizers instead of subclassing and drawing the image yourself. Alternatively, you can also use a generic `UIView` object and assign your image as the content of the view's `CALayer` object.

Animations are another way to make visible changes to a view without requiring you to subclass and implement complex drawing code. Many properties of the `UIView` class are animatable, which means changes to those properties can trigger system-generated animations. Starting animations requires as little as one line of code to indicate that any changes that follow should be animated. For more information about animation support for views, see Animations.

For more information about appearance and behavior configuration, see About Views in UIKit User Interface Catalog.

Symbols

Initializing a View Object

`init(frame: CGRect)`

Initializes and returns a newly allocated view object with the specified frame rectangle.

Configuring a View's Visual Appearance

`var backgroundColor: UIColor?`

The view's background color.

`var isHidden: Bool`

A Boolean value that determines whether the view is hidden.

`var alpha: CGFloat`

The view's alpha value.

`var isOpaque: Bool`

A Boolean value that determines whether the view is opaque.

`var tintColor: UIColor!`

The first nondefault tint color value in the view's hierarchy, ascending from and starting with the view itself.

`var tintAdjustmentMode: UIViewTintAdjustmentMode`

The first non-default tint adjustment mode value in the view's hierarchy, ascending from and starting with the view itself.

`var clipsToBounds: Bool`

A Boolean value that determines whether subviews are confined to the bounds of the view.

`var clearsContextBeforeDrawing: Bool`

A Boolean value that determines whether the view's bounds should be automatically cleared before drawing.

`var mask: UIView?`

An optional view whose alpha channel is used to mask a view's content.

`class var layerClass: AnyClass`

The class used to create the layer for instances of this class.

`var layer: CALayer`

The view's Core Animation layer used for rendering.

Configuring the Event-Related Behavior

`var isUserInteractionEnabled: Bool`

A Boolean value that determines whether user events are ignored and removed from the event queue.

`var isMultipleTouchEnabled: Bool`

A Boolean value that indicates whether the receiver handles multi-touch events.

`var isExclusiveTouch: Bool`

A Boolean value that indicates whether the receiver handles touch events exclusively.

Configuring the Bounds and Frame Rectangles

`var frame: CGRect`

The frame rectangle, which describes the view's location and size in its superview's coordinate system.

`var bounds: CGRect`

The bounds rectangle, which describes the view's location and size in its own coordinate system.

`var center: CGPoint`

The center of the frame.

`var transform: CGAffineTransform`

Specifies the transform applied to the receiver, relative to the center of its bounds.

Managing the View Hierarchy

`var superview: UIView?`

The receiver's superview, or `nil` if it has none.

`var subviews: [UIView]`

The receiver's immediate subviews.

`var window: UIWindow?`

The receiver's window object, or `nil` if it has none.

`func addSubview(UIView)`

Adds a view to the end of the receiver's list of subviews.

`func bringSubview(toFront: UIView)`

Moves the specified subview so that it appears on top of its siblings.

`func sendSubview(toBack: UIView)`

Moves the specified subview so that it appears behind its siblings.

`func removeFromSuperview()`

Unlinks the view from its superview and its window, and removes it from the responder chain.

`func insertSubview(UIView, at: Int)`

Inserts a subview at the specified index.

`func insertSubview(UIView, aboveSubview: UIView)`

Inserts a view above another view in the view hierarchy.

`func insertSubview(UIView, belowSubview: UIView)`

Inserts a view below another view in the view hierarchy.

`func exchangeSubview(at: Int, withSubviewAt: Int)`

Exchanges the subviews at the specified indices.

`func isDescendant(of: UIView)`

Returns a Boolean value indicating whether the receiver is a subview of a given view or identical to that view.

Configuring the Resizing Behavior

`var autoresizingMask: UIViewAutoresizing`

An integer bit mask that determines how the receiver resizes itself when its superview's bounds change.

`var autoresizesSubviews: Bool`

A Boolean value that determines whether the receiver automatically resizes its subviews when its bounds change.

`var contentMode: UIViewContentMode`

A flag used to determine how a view lays out its content when its bounds change.

`func sizeThatFits(CGSize)`

Asks the view to calculate and return the size that best fits the specified size.

`func sizeToFit()`

Resizes and moves the receiver view so it just encloses its subviews.

Laying out Subviews

`func layoutSubviews()`

Lays out subviews.

`func setNeedsLayout()`

Invalidates the current layout of the receiver and triggers a layout update during the next update cycle.

`func layoutIfNeeded()`

Lays out the subviews immediately.

`class var requiresConstraintBasedLayout: Bool`

A Boolean value that indicates whether the receiver depends on the constraint-based layout system.

`var translatesAutoresizingMaskIntoConstraints: Bool`

A Boolean value that determines whether the view's autoresizing mask is translated into Auto Layout constraints.

Creating Constraints Using Layout Anchors

`var bottomAnchor: NSLayoutYAxisAnchor`

A layout anchor representing the bottom edge of the view's frame.

`var centerXAnchor: NSLayoutXAxisAnchor`

A layout anchor representing the horizontal center of the view's frame.

`var centerYAnchor: NSLayoutYAxisAnchor`

A layout anchor representing the vertical center of the view's frame.

`var firstBaselineAnchor: NSLayoutYAxisAnchor`

A layout anchor representing the baseline for the topmost line of text in the view.

`var heightAnchor: NSLayoutDimension`

A layout anchor representing the height of the view's frame.

`var lastBaselineAnchor: NSLayoutYAxisAnchor`

A layout anchor representing the baseline for the bottommost line of text in the view.

`var leadingAnchor: NSLayoutXAxisAnchor`

A layout anchor representing the leading edge of the view's frame.

`var leftAnchor: NSLayoutXAxisAnchor`

A layout anchor representing the left edge of the view's frame.

`var rightAnchor: NSLayoutXAxisAnchor`

A layout anchor representing the right edge of the view's frame.

`var topAnchor: NSLayoutYAxisAnchor`

A layout anchor representing the top edge of the view's frame.

`var trailingAnchor: NSLayoutXAxisAnchor`

A layout anchor representing the trailing edge of the view's frame.

`var widthAnchor: NSLayoutDimension`

A layout anchor representing the width of the view's frame.

Managing the View's Constraints

`var constraints: [NSLayoutConstraint]`

The constraints held by the view.

`func addConstraint(NSLayoutConstraint)`

Adds a constraint on the layout of the receiving view or its subviews.

`func addConstraints([NSLayoutConstraint])`

Adds multiple constraints on the layout of the receiving view or its subviews.

`func removeConstraint(NSLayoutConstraint)`

Removes the specified constraint from the view.

`func removeConstraints([NSLayoutConstraint])`

Removes the specified constraints from the view.

Working with Layout Guides

`func addLayoutGuide(UILayoutGuide)`

Adds the specified layout guide to the view.

`var layoutGuides: [UILayoutGuide]`

The array of layout guide objects owned by this view.

`var layoutMarginsGuide: UILayoutGuide`

A layout guide representing the view's margins.

`var readableContentGuide: UILayoutGuide`

A layout guide representing an area with a readable width within the view.

`func removeLayoutGuide(UILayoutGuide)`

Removes the specified layout guide from the view.

Measuring in Auto Layout

`func systemLayoutSizeFitting(CGSize)`

Returns the size of the view that satisfies the constraints it holds.

`func systemLayoutSizeFitting(CGSize, withHorizontalFittingPriority: UILayoutPriority, verticalFittingPriority: UILayoutPriority)`

Returns the size of the view that satisfies the constraints it holds.

`var intrinsicContentSize: CGSize`

The natural size for the receiving view, considering only properties of the view itself.

`func invalidateIntrinsicContentSize()`

Invalidates the view's intrinsic content size.

`func contentCompressionResistancePriority(for: UILayoutConstraintAxis)`

Returns the priority with which a view resists being made smaller than its intrinsic size.

`func setContentCompressionResistancePriority(UILayoutPriority, for: UILayoutConstraintAxis)`

Sets the priority with which a view resists being made smaller than its intrinsic size.

```
func contentHuggingPriority(for: UILayoutConstraintAxis)
```

Returns the priority with which a view resists being made larger than its intrinsic size.

```
func setContentHuggingPriority(UILayoutPriority, for: UILayoutConstraintAxis)
```

Sets the priority with which a view resists being made larger than its intrinsic size.

Aligning Views in Auto Layout

```
func alignmentRect(forFrame: CGRect)
```

Returns the view's alignment rectangle for a given frame.

```
func frame(forAlignmentRect: CGRect)
```

Returns the view's frame for a given alignment rectangle.

```
var alignmentRectInsets: UIEdgeInsets
```

The insets from the view's frame that define its alignment rectangle.

~~`func forBaselineLayout()`~~

Returns a view used to satisfy baseline constraints.

Deprecated

```
var forFirstBaselineLayout: UIView
```

Returns a view used to satisfy first baseline constraints.

```
var forLastBaselineLayout: UIView
```

Returns a view used to satisfy last baseline constraints.

Triggering Auto Layout

`func needsUpdateConstraints()`

A Boolean value that determines whether the view's constraints need updating.

`func setNeedsUpdateConstraints()`

Controls whether the view's constraints need updating.

`func updateConstraints()`

Updates constraints for the view.

`func updateConstraintsIfNeeded()`

Updates the constraints for the receiving view and its subviews.

Debugging Auto Layout

See Auto Layout Guide for more details on debugging constraint-based layout.

`func constraintsAffectingLayout(for: UILayoutConstraintAxis)`

Returns the constraints impacting the layout of the view for a given axis.

`var hasAmbiguousLayout: Bool`

A Boolean value that determines whether the constraints impacting the layout of the view incompletely specify the location of the view.

`func exerciseAmbiguityInLayout()`

Randomly changes the frame of a view with an ambiguous layout between the different valid values.

Managing the User Interface Direction

`var semanticContentAttribute: UISemanticContentAttribute`

A semantic description of the view's contents, used to determine whether the view should be flipped when switching between left-to-right and right-to-left layouts.

`var effectiveUserInterfaceLayoutDirection: UIUserInterfaceLayoutDirection`

The user interface layout direction appropriate for arranging the immediate content of the view.

`class func userInterfaceLayoutDirection(for: UISemanticContentAttribute)`

Returns the user interface direction for the given semantic content attribute.

`class func userInterfaceLayoutDirection(for: UISemanticContentAttribute, relativeTo: UIUserInterfaceLayoutDirection)`

Returns the layout direction implied by the specified semantic content attribute, relative to the specified layout direction.

Configuring Content Margins

`var layoutMargins: UIEdgeInsets`

The default spacing to use when laying out content in the view.

`var preservesSuperviewLayoutMargins: Bool`

A Boolean value indicating whether the current view also respects the margins of its superview.

`func layoutMarginsDidChange()`

Notifies the view that the layout margins changed.

Drawing and Updating the View

`func draw(CGRect)`

Draws the receiver's image within the passed-in rectangle.

`func setNeedsDisplay()`

Marks the receiver's entire bounds rectangle as needing to be redrawn.

`func setNeedsDisplay(CGRect)`

Marks the specified rectangle of the receiver as needing to be redrawn.

`var contentScaleFactor: CGFloat`

The scale factor applied to the view.

`func tintColorDidChange()`

Called by the system when the `tintColor` property changes.

Formatting Printed View Content

`func viewPrintFormatter()`

Returns a print formatter for the receiving view.

`func draw(CGRect, for: UIViewPrintFormatter)`

Implemented to draw the view's content for printing.

Managing Gesture Recognizers

`func addGestureRecognizer(UIGestureRecognizer)`

Attaches a gesture recognizer to the view.

`func removeGestureRecognizer(UIGestureRecognizer)`

Detaches a gesture recognizer from the receiving view.

`var gestureRecognizers: [UIGestureRecognizer]?`

The gesture-recognizer objects currently attached to the view.

`func gestureRecognizerShouldBegin(UIGestureRecognizer)`

Asks the view if the gesture recognizer should be allowed to continue tracking touch events.

Animating Views with Block Objects

```
class func animate(withDuration: TimeInterval, delay: TimeInterval, options: UIViewAnimationOptions = [], animations: @escaping () -> Void, completion: ((Bool) -> Void)? = nil)
```

Animate changes to one or more views using the specified duration, delay, options, and completion handler.

```
class func animate(withDuration: TimeInterval, animations: @escaping () -> Void, completion: ((Bool) -> Void)? = nil)
```

Animate changes to one or more views using the specified duration and completion handler.

```
class func animate(withDuration: TimeInterval, animations: @escaping () -> Void)
```

Animate changes to one or more views using the specified duration.

```
class func transition(with: UIView, duration: TimeInterval, options: UIViewAnimationOptions = [], animations: (() -> Void)?, completion: ((Bool) -> Void)? = nil)
```

Creates a transition animation for the specified container view.

```
class func transition(from: UIView, to: UIView, duration: TimeInterval, options: UIViewAnimationOptions = [], completion: ((Bool) -> Void)? = nil)
```

Creates a transition animation between the specified views using the given parameters.

```
class func animateKeyframes(withDuration: TimeInterval, delay: TimeInterval, options: UIViewKeyframeAnimationOptions = [], animations: @escaping () -> Void, completion: ((Bool) -> Void)? = nil)
```

Creates an animation block object that can be used to set up keyframe-based animations for the current view.

```
class func addKeyframe(withRelativeStartTime: Double,
relativeDuration: Double, animations: @escaping () -> Void)
```

Specifies the timing and animation values for a single frame of a keyframe animation.

```
class func perform(UISystemAnimation, on: [UIView], options: UIViewAni
mationOptions = [], animations: (() -> Void)?, completion: ((Bool) ->
Void)? = nil)
```

Performs a specified system-provided animation on one or more views, along with optional parallel animations that you define.

```
class func animate(withDuration: TimeInterval, delay: TimeInterval, us
ingSpringWithDamping: CGFloat, initialSpringVelocity: CGFloat,
options: UIViewAnimationOptions = [], animations: @escaping () -> Voi
d, completion: ((Bool) -> Void)? = nil)
```

Performs a view animation using a timing curve corresponding to the motion of a physical spring.

```
class func performWithoutAnimation(() -> Void)
```

Disables a view transition animation.

Animating Views

Use of the methods in this section is discouraged in iOS 4 and later. Use the block-based animation methods instead.

```
class func beginAnimations(String?, context: UnsafeMutableRawPointer?)
```

Marks the beginning of a begin/commit animation block.

```
class func commitAnimations()
```

Marks the end of a begin/commit animation block and schedules the animations for execution.

```
class func setAnimationStart(Date)
```

Sets the start time for the current animation block.

`class func setAnimationsEnabled(Bool)`

Sets whether animations are enabled.

`class func setAnimationDelegate(Any?)`

Sets the delegate for any animation messages.

`class func setAnimationWillStart(Selector?)`

Sets the message to send to the animation delegate when the animation starts.

`class func setAnimationDidStop(Selector?)`

Sets the message to send to the animation delegate when animation stops.

`class func setAnimationDuration(TimeInterval)`

Sets the duration (measured in seconds) of the animations in an animation block.

`class func setAnimationDelay(TimeInterval)`

Sets the amount of time (in seconds) to wait before animating property changes within an animation block.

`class func setAnimationCurve(UIViewAnimationCurve)`

Sets the curve to use when animating property changes within an animation block.

`class func setAnimationRepeatCount(Float)`

Sets the number of times animations within an animation block repeat.

`class func setAnimationRepeatAutoreverses(Bool)`

Sets whether the animations within an animation block automatically reverse themselves.

```
class func setAnimationBeginsFromCurrentState(Bool)
```
Sets whether the animation should begin playing from the current state.

```
class func setAnimationTransition(UIViewAnimationTransition, for: UIVi
ew, cache: Bool)
```
Sets a transition to apply to a view during an animation block.

```
class var areAnimationsEnabled: Bool
```
A Boolean value indicating whether animations are enabled.

Using Motion Effects

```
func addMotionEffect(UIMotionEffect)
```
Begins applying a motion effect to the view.

```
var motionEffects: [UIMotionEffect]
```
The array of motion effects for the view.

```
func removeMotionEffect(UIMotionEffect)
```
Stops applying a motion effect to the view.

Preserving and Restoring State

```
var restorationIdentifier: String?
```
The identifier that determines whether the view supports state restoration.

```
func encodeRestorableState(with: NSCoder)
```
Encodes state-related information for the view.

```
func decodeRestorableState(with: NSCoder)
```
Decodes and restores state-related information for the view.

Capturing a View Snapshot

```
func snapshotView(afterScreenUpdates: Bool)
```
Returns a snapshot view based on the contents of the current view.

```
func resizableSnapshotView(from: CGRect, afterScreenUpdates: Bool, wit
hCapInsets: UIEdgeInsets)
```
Returns a snapshot view based on the specified contents of the current view, with stretchable insets.

```
func drawHierarchy(in: CGRect, afterScreenUpdates: Bool)
```
Renders a snapshot of the complete view hierarchy as visible onscreen into the current context.

Identifying the View at Runtime

```
var tag: Int
```
An integer that you can use to identify view objects in your application.

```
func viewWithTag(Int)
```
Returns the view whose tag matches the specified value.

Converting Between View Coordinate Systems

```
func convert(CGPoint, to: UIView?)
```
Converts a point from the receiver's coordinate system to that of the specified view.

```
func convert(CGPoint, from: UIView?)
```
Converts a point from the coordinate system of a given view to that of the receiver.

```
func convert(CGRect, to: UIView?)
```
Converts a rectangle from the receiver's coordinate system to that of another view.

```
func convert(CGRect, from: UIView?)
```

Converts a rectangle from the coordinate system of another view to that of the receiver.

Hit Testing in a View

```
func hitTest(CGPoint, with: UIEvent?)
```

Returns the farthest descendant of the receiver in the view hierarchy (including itself) that contains a specified point.

```
func point(inside: CGPoint, with: UIEvent?)
```

Returns a Boolean value indicating whether the receiver contains the specified point.

Ending a View Editing Session

```
func endEditing(Bool)
```

Causes the view (or one of its embedded text fields) to resign the first responder status.

Observing View-Related Changes

```
func didAddSubview(UIView)
```

Tells the view that a subview was added.

```
func willRemoveSubview(UIView)
```

Tells the view that a subview is about to be removed.

```
func willMove(toSuperview: UIView?)
```

Tells the view that its superview is about to change to the specified superview.

```
func didMoveToSuperview()
```

Tells the view that its superview changed.

`func willMove(toWindow: UIWindow?)`

Tells the view that its window object is about to change.

`func didMoveToWindow()`

Tells the view that its window object changed.

Observing Focus

`var canBecomeFocused: Bool`

A Boolean value that indicates whether the view is currently capable of being focused.

`class var inheritedAnimationDuration: TimeInterval`

The inherited duration of the current animation.

`var isFocused: Bool`

A Boolean value that indicates whether the item is currently focused.

Constants

`UIViewAnimationOptions`

Options for animating views using block objects.

`UIViewAnimationCurve`

Specifies the supported animation curves.

`UIViewKeyframeAnimationOptions`

Key frame animation options used with the `animateKeyframes(withDuration:delay:options:animations:completion:)` method.

`UIViewContentMode`

Options to specify how a view adjusts its content when its size changes.

`UILayoutConstraintAxis`

Keys that specify a horizontal or vertical layout constraint between objects.

`UIViewTintAdjustmentMode`

The tint adjustment mode for the view.

`UISystemAnimation`

Option to remove the views from the hierarchy when animation is complete.

Fitting Size

View fitting options used in the `systemLayoutSizeFitting(_:)` method.

UIView Intrinsic Metric Constan

t

The option to indicate that a view has no intrinsic metric for a given numeric property.

`UIViewAutoresizing`

Options for automatic view resizing.

`UIViewAnimationTransition`

Animation transition options for use in an animation block object.

`UISemanticContentAttribute`

A semantic description of the view's contents, used to determine whether the view should be flipped when switching between left-to-right and right-to-left layouts.

Initializers

`init?(coder: NSCoder)`

Relationships

Inherits From

`UIResponder`

Conforms To

- `CALayerDelegate`
- `CVarArg`
- `Equatable`
- `Hashable`
- `NSCoding`
- `UIAccessibilityIdentification`
- `UIAppearance`
- `UIAppearanceContainer`
- `UICoordinateSpace`
- `UIDynamicItem`
- `UIFocusItem`
- `UITraitEnvironment`