Class

UIView

An object that manages the content for a rectangular area on the screen.

iOS 2.0+ | iPadOS 2.0+ | Mac Catalyst 13.1+ | tvOS | visionOS 1.0+

```
@MainActor
class UIView
```

Mentioned in

📄 Using responders and the responder chain to handle events

📄 About App Development with UIKit

📄 Customizing a document-based app's launch experience

📄 Customizing drawings

📄 Enhancing your app with fluid transitions

Overview

Views are the fundamental building blocks of your app's user interface, and the UIView class defines the behaviors that are common to all views. A view object renders content within its bounds rectangle, and handles any interactions with that content. The UIView class is a concrete class that you can instantiate and use to display a fixed background color. You can also subclass it to draw more sophisticated content. To display labels, images, buttons, and other interface elements commonly found in apps, use the view subclasses that the UIKit framework provides rather than trying to define your own.

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 UIKit or Core Graphics.

  - You can animate some view properties to new values.

- Layout and subview management

  - Views may contain zero or more subviews.

  - Views can adjust the size and position of their subviews.

  - Use Auto Layout to define the rules for resizing and repositioning your views in response to changes in the view hierarchy.

- Event handling

  - A view is a subclass of `UIResponder` and can respond to touches and other types of events.

  - Views can install gesture recognizers to handle common gestures.

Views can nest inside other views to create view hierarchies, which offer a convenient way to organize related content. Nesting a view creates a parent-child relationship between the nested child view (known as the *subview*) and the parent (known as the *superview*). A parent view may contain any number of subviews, but each subview has only one superview. By default, when a subview's visible area extends outside of the bounds of its superview, no clipping of the subview's content occurs. Use the `clipsToBounds` property to change that behavior.

The `frame` and `bounds` properties define the geometry of each view. The `frame` property defines the origin and dimensions of the view in the coordinate system of its superview. The `bounds` property defines the internal dimensions of the view as it sees them, and its use is almost exclusive to custom drawing code. The center property provides a convenient way to reposition a view without changing its `frame` or `bounds` properties directly.

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

Create a view

Normally, you create views in your storyboards by dragging them from the library to your canvas. You can also create views programmatically. When creating a view, you typically specify its initial size and position relative to its future superview. For example, the following example creates a view and places its top-left corner at the point (10, 10) in the superview's coordinate system (once it is added to that superview).

Swift   Objective-C

```swift
let rect = CGRect(x: 10, y: 10, width: 100, height: 100)
let myView = UIView(frame: rect)
```

To add a subview to another view, call the addSubview(_:) method on the superview. You may add any number of subviews to a view, and sibling views may overlap each other without any issues in iOS. Each call to the addSubview(_:) method places the new view on top of all other siblings. You can specify the relative z-order of subview by adding it using the insert Subview(_:aboveSubview:) and insertSubview(_:belowSubview:) methods. You can also exchange the position of already added subviews using the exchangeSubview(at:with SubviewAt:) method.

After creating a view, create Auto Layout rules to govern how the size and position of the view change in response to changes in the rest of the view hierarchy. For more information, see Auto Layout Guide.

Draw views

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's your responsibility to notify the system that your view needs to be redrawn. You do this by calling your view's setNeedsDisplay() or set NeedsDisplay(_:) 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're 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 detailed information about the view drawing cycle and the role your views have in this cycle, see View Programming Guide for iOS.

Animate views

Changes to several view properties can be animated — that is, changing the property creates an animation starting at the current value and ending at the new value that you specify. The following properties of the `UIView` class are animatable:

- `frame`

- `bounds`

- `center`

- `transform`

- `alpha`

- `backgroundColor`

To animate your changes, create a `UIViewPropertyAnimator` object and use its handler block to change the values of your view's properties. The `UIViewPropertyAnimator` class lets you specify the duration and timing of your animations, but it performs the actual animations. You can pause a property-based animator that's currently running to interrupt the animation and drive it interactively. For more information, see `UIViewPropertyAnimator`.

Threading considerations

Manipulations to your app'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 app. 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 storyboards or nib files 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.

- Layout and 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.

  - `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.

- Event Handling:

  - `gestureRecognizerShouldBegin(_:)` – Implement this method if your view handles touch events directly and might want to prevent attached gesture recognizers from triggering additional actions.

  - `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.)

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 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 Animate views.

Topics

Creating a view object

`init(frame: CGRect)`

 Creates a view with the specified frame rectangle.

```
init?(coder: NSCoder)
```
Creates a view from data in an unarchiver.

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: UIView.TintAdjustmentMode
```
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
```
Returns the class used to create the layer for instances of this class.

```
var layer: CALayer
```
The view's Core Animation layer to use for rendering.

Configuring a view's corners

`var cornerConfiguration: UICornerConfiguration`

A configuration that defines the corners of the view.

`struct UICornerConfiguration`

A configuration that defines how corner radii are mapped to the corners of a rectangle.

`struct UICornerRadius`

A type that represents the radius the system uses to round a corner.

`func effectiveRadius(corner: UIRectCorner) -> CGFloat`

Returns the effective radius for the corner you provide, calculated using the view's current corner configuration.

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 view receives more than one touch at a time.

`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 point of the view's frame rectangle.

var `transform: CGAffineTransform`

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

var `transform3D: CATransform3D`

The three-dimensional transform to apply to the view.

var `anchorPoint: CGPoint`

The anchor point of the view's bounds rectangle.

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 `bringSubviewToFront(UIView)`

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

func `sendSubviewToBack(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) -> Bool
```
Returns a Boolean value indicating whether the receiver is a subview of a given view or identical to that view.

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 trait changes

```
protocol UITraitChangeObservable
```
A type that calls your code in reaction to changes in the trait environment.

Requesting trait updates

```
func updateTraitsIfNeeded()
```

Overriding trait values

```
var traitOverrides: UITraitOverrides
```

```
struct UITraitOverrides
```

Configuring content margins

📄 **Positioning content within layout margins**

Position views so that they aren't crowded by other content.

`var directionalLayoutMargins: NSDirectionalEdgeInsets`

The default spacing to use when laying out content in a view, taking into account the current language direction.

`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.

Getting the safe area

📄 **Positioning content relative to the safe area**

Position views so that they aren't obstructed by other content.

`var safeAreaInsets: UIEdgeInsets`

The insets that you use to determine the safe area for this view.

`var safeAreaLayoutGuide: UILayoutGuide`

The layout guide representing the portion of your view that is unobscured by bars and other content.

`func safeAreaInsetsDidChange()`

Called when the safe area of the view changes.

`var insetsLayoutMarginsFromSafeArea: Bool`

A Boolean value indicating whether the view's layout margins are updated automatically to reflect the safe area.

Managing the view's constraints

Adjust the size and position of the view using Auto Layout 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.

Creating constraints using layout anchors

Attach Auto Layout constraints to one of the view's 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.

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) -> CGSize`

Returns the optimal size of the view based on its current constraints.

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

Returns the optimal size of the view based on its constraints and the specified fitting
priorities.

`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: NSLayoutConstraint.Axis) -> UILayoutPriority`

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

`func setContentCompressionResistancePriority(UILayoutPriority, for: NSLayoutConstraint.Axis)`

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

`func contentHuggingPriority(for: NSLayoutConstraint.Axis) -> UILayoutPriority`

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

`func setContentHuggingPriority(UILayoutPriority, for: NSLayoutConstraint.Axis)`

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

Aligning views in Auto Layout

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

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

`func frame(forAlignmentRect: CGRect) -> 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.

`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() -> Bool

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

func constraintsAffectingLayout(for: NSLayoutConstraint.Axis) -> [NSLayoutConstraint]

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.

Configuring the resizing behavior

Define how a view adjusts its content when its bounds change.

var contentMode: UIView.ContentMode

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

enum ContentMode

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

func sizeThatFits(CGSize) -> 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.

`var` `autoresizesSubviews:` `Bool`

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

`var` `autoresizingMask:` `UIView.AutoresizingMask`

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

Laying out subviews

Lay out views manually if your app doesn't use Auto Layout.

`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, if layout updates are pending.

`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.

Accessing insets and layout guides

`struct` `LayoutRegion`

`func` `directionalEdgeInsets(for:` `UIView.LayoutRegion)` `->` `NSDirectional` `EdgeInsets`

`func` `edgeInsets(for:` `UIView.LayoutRegion)` `->` `UIEdgeInsets`

`func` `layoutGuide(for:` `UIView.LayoutRegion)` `->` `UILayoutGuide`

Adjusting the user interface

`var` `overrideUserInterfaceStyle:` `UIUserInterfaceStyle`

   The user interface style adopted by the view and all of its subviews.

`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:` `UIUserInterfaceLayout` `Direction`

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

`class func` `userInterfaceLayoutDirection(for:` `UISemanticContentAttribute` `)` `->` `UIUserInterfaceLayoutDirection`

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

`class func` `userInterfaceLayoutDirection(for:` `UISemanticContentAttribute` `,` `relativeTo:` `UIUserInterfaceLayoutDirection)` `->` `UIUserInterfaceLayout` `Direction`

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

Constraining views to the keyboard

`var` `keyboardLayoutGuide:` `UIKeyboardLayoutGuide`

   A layout guide that tracks the keyboard's position in your app's layout.

Adding and removing interactions

`func` `addInteraction(any UIInteraction)`

   Adds an interaction to the view.

`func` `removeInteraction(any UIInteraction)`

   Removes an interaction from the view.

`var` `interactions:` `[any UIInteraction]`

   The array of interactions for the view.

`protocol` `UIInteraction`

   The protocol that an interaction implements to access the view that owns it.

Drawing and updating the view

`func draw(CGRect)`

Draws the view'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 tint color property changes.

Updating the view when property values change

`struct Invalidating`

A property wrapper that notifies the system that a property value change has invalidated an aspect of the containing view.

`protocol UIViewInvalidating`

Implements a type of invalidation that can occur on a view that requires an update.

Formatting printed view content

`func viewPrintFormatter() -> UIViewPrintFormatter`

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) -> Bool
```

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

Working with focus

```
var canBecomeFocused: Bool
```

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

```
class var inheritedAnimationDuration: TimeInterval
```

Returns the inherited duration of the current animation.

```
var isFocused: Bool
```

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

```
var focusGroupIdentifier: String?
```

The identifier of the focus group that this view belongs to.

```
var focusEffect: UIFocusEffect?
```

The visual effect to apply when the view becomes focused.

```
var focusGroupPriority: UIFocusGroupPriority
```

The importance of the item within a focus group, used by the focus system to determine the group's primary item.

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.

Managing the hover appearance

`var hoverStyle: UIHoverStyle?`

The hover style for the view.

`class UIHoverStyle`

The hover style to apply to a view, including an effect and a shape to use for displaying that effect.

`class UIHoverEffectLayer`

Managing font-sizing preferences

`var minimumContentSizeCategory: UIContentSizeCategory?`

The minimum content size category for the view and its subviews.

`var maximumContentSizeCategory: UIContentSizeCategory?`

The maximum content size category for the view and its subviews.

`var appliedContentSizeCategoryLimitsDescription: String`

A string that lists each of the view's superviews, its content size category, and whether that view has content size category limits.

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) -> UIView?`

Returns a snapshot view based on the contents of the current view.

```
func resizableSnapshotView(from: CGRect, afterScreenUpdates: Bool, with
CapInsets: UIEdgeInsets) -> UIView?
```
Returns a snapshot view based on the specified contents of the current view, with stretchable insets.

```
func drawHierarchy(in: CGRect, afterScreenUpdates: Bool) -> 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) -> UIView?
```
Returns the view whose tag matches the specified value.

Converting between view coordinate systems

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

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

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

```
func convert(CGRect, from: UIView?) -> CGRect
```
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?) -> UIView?
```
Returns the farthest descendant in the view hierarchy of the current view, including itself, that contains the specified point.

```
func point(inside: CGPoint, with: UIEvent?) -> Bool
```
Returns a Boolean value indicating whether the receiver contains the specified point.

Ending a view-editing session

`func endEditing(Bool) -> Bool`

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

Modifying the accessibility behavior

`var accessibilityIgnoresInvertColors: Bool`

A Boolean value indicating whether the view ignores an accessibility request to invert its colors.

`var largeContentImage: UIImage?`

An image that represents the view in the large content viewer.

`var largeContentImageInsets: UIEdgeInsets`

Insets to adjust the position of the view's image so it appears centered in the large content viewer.

`var largeContentTitle: String?`

A string that describes the view in the large content viewer.

`var scalesLargeContentImage: Bool`

A Boolean value that indicates whether the large content viewer scales the item's image to a larger size.

`var showsLargeContentViewer: Bool`

A Boolean value that indicates whether to show the view in the large content viewer.

Animating views

`static func animate(Animation, changes: () -> Void, completion: (() -> Void)?)`

`class func animate(springDuration: TimeInterval, bounce: CGFloat, initialSpringVelocity: CGFloat, delay: TimeInterval, options: UIView.AnimationOptions, animations: () -> Void, completion: ((Bool) -> Void)?)`

Animates changes to one or more views using a spring animation with the specified duration, bounce, initial velocity, delay, options, and completion handler.

```
class func animate(withDuration: TimeInterval, delay: TimeInterval,
options: UIView.AnimationOptions, animations: () -> Void, completion:
((Bool) -> Void)?)
```

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

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

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

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

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

```
class func transition(with: UIView, duration: TimeInterval, options:
UIView.AnimationOptions, animations: (() -> Void)?, completion: ((Bool)
-> Void)?)
```

Creates a transition animation for the specified container view.

```
class func transition(from: UIView, to: UIView, duration: TimeInterval,
options: UIView.AnimationOptions, completion: ((Bool) -> Void)?)
```

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

```
class func animateKeyframes(withDuration: TimeInterval, delay: Time
Interval, options: UIView.KeyframeAnimationOptions, animations: () ->
Void, completion: ((Bool) -> Void)?)
```

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: () -> Void)
```

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

```
class func perform(UIView.SystemAnimation, on: [UIView], options:
UIView.AnimationOptions, animations: (() -> Void)?, completion: ((Bool)
-> Void)?)
```

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,
usingSpringWithDamping: CGFloat, initialSpringVelocity: CGFloat,
options: UIView.AnimationOptions, animations: () -> Void, completion:
((Bool) -> Void)?)
```

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.

`class func modifyAnimations(withRepeatCount: CGFloat, autoreverses: Bool, animations: () -> Void)`

Repeats the specified animations a specific number of times, optionally running the animation forward and backward.

Constants

`enum AnimationCurve`

Specifies the supported animation curves.

`struct AnimationOptions`

Options for animating views using block objects.

`enum AnimationTransition`

Animation transition options for use in an animation block object.

`enum SystemAnimation`

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

`struct KeyframeAnimationOptions`

Options for configuring keyframe-based animations.

`enum Axis`

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

`enum TintAdjustmentMode`

The tint adjustment mode for the view.

`class let layoutFittingCompressedSize: CGSize`

The option to use the smallest possible size.

`class let layoutFittingExpandedSize: CGSize`

The option to use the largest possible size.

`class let noIntrinsicMetric: CGFloat`

The absence of an intrinsic metric for a given numeric view property.

struct AutoresizingMask

Options for automatic view resizing.

enum 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.

Deprecated

☰   Deprecated symbols

Symbols that views no longer support.

Initializers

convenience init()

Instance Methods

func setNeedsUpdateProperties()

Call to manually request a properties update for the view. Multiple requests may be coalesced into a single update alongside the next layout pass.

func updateProperties()

Override point for subclasses to update properties of this view. Never call this method directly; use setNeedsUpdateProperties to schedule an update.

func updatePropertiesIfNeeded()

Forces an immediate properties update for this view (and its view controller, if applicable) and any subviews, including any view controllers or views in its subtree.

Enumerations

enum Invalidations

Changes that cause an aspect of a view to be invalid and require an update.

Relationships

Inherits From

UIResponder

Inherited By

UIActionSheet
UIActivityIndicatorView
UIAlertView
UIBackgroundExtensionView
UICalendarView
UICollectionReusableView
UIContentUnavailableView
UIControl
UIEventAttributionView
UIImageView
UIInputView
UILabel
UIListContentView
UINavigationBar
UIPickerView
UIPopoverBackgroundView
UIProgressView
UIScrollView
UISearchBar
UIStackView
UIStandardTextCursorView
UITabBar
UITableViewCell
UITableViewHeaderFooterView
UIToolbar
UIVisualEffectView
UIWebView
UIWindow

Conforms To

CALayerDelegate
CVarArg
Copyable
CustomDebugStringConvertible

CustomStringConvertible
Equatable
Hashable
NSCoding
NSObjectProtocol
NSTouchBarProvider
Sendable
SendableMetatype
UIAccessibilityIdentification
UIActivityItemsConfigurationProviding
UIAppearance
UIAppearanceContainer
UICoordinateSpace
UIDynamicItem
UIFocusEnvironment
UIFocusItem
UIFocusItemContainer
UILargeContentViewerItem
UIPasteConfigurationSupporting
UIPopoverPresentationControllerSourceItem
UIResponderStandardEditActions
UITraitChangeObservable
UITraitEnvironment
UIUserActivityRestoring

See Also

View fundamentals

{}   UIKit Catalog: Creating and customizing views and controls

Customize your app's user interface with views and controls.