Framework

UIKit

Construct and manage your app’s user interface for iOS and tvOS. Respond to user interactions and system events, access various device features, enable accessibility, and work with animations, text, and images. In watchOS apps, enable accessibility and work with fonts and images.

Language

Swift | Objective-C

SDKs

iOS 2.0+

tvOS 9.0+

watchOS 2.0+

On This Page

Overview

The UIKit framework (`UIKit.framework`) provides the crucial infrastructure needed to construct and manage iOS and tvOS apps. This framework provides the window and view architecture needed to manage an app’s user interface, the event handling infrastructure needed to respond to user input, and the app model needed to drive the main run loop and interact with the system.

Additional UIKit Features

In addition to the core app behaviors, UIKit provides support for the following features:

- A view controller model to encapsulate the contents of your user interface
- Handling touch- and motion-based events

- A document model that includes iCloud integration; see Document-Based App Programming Guide for iOS
- Graphics and windowing, including support for external displays; see View Programming Guide for iOS
- Managing the app's foreground and background execution
- Printing; see Drawing and Printing Guide for iOS
- Customizing the appearance of standard UIKit controls
- Animating user-interface content
- Integration with other apps on the system through URL schemes and framework interfaces
- Working with various accessibility settings and preferences
- PDF creation
- The user's photo library

In iOS, UIKit also supports the following features, some of which are device specific:

- Cut, copy, and paste actions
- The Apple Push Notification service; see Local and Remote Notification Programming Guide
- Local notification scheduling and delivery; see Local and Remote Notification Programming Guide
- Using custom input views that behave like the system keyboard
- Creating custom text views that interact with the system keyboard

- Sharing content through email, Twitter, Facebook, and other services
- The built-in camera (where present)
- Device name and model information
- Battery state information
- Proximity sensor information
- Remote control information from attached headsets

> **Note**
>
> For the most part, use UIKit classes only from your app's main thread. This is particularly true for classes derived from `UIResponder` or that involve manipulating your app's user interface in any way.

Symbols

Classes

`NSDataAsset`

The `NSDataAsset` class enables you to access an object from a data set type stored in an asset catalog. The object's content is stored as a set of one or more files with associated device attributes. These sets can also be tagged for use as on-demand resources.

`NSFileProviderExtension`

`NSFileProviderExtension` is the principal class for the File Provider extension. The file provider acts as the back end for the Document Picker extension. It ensures that all the URLs returned by the document picker are backed by files on disk. It can respond to both coordinated-read and coordinated-write operations from the host app, letting you download and upload files as needed. You must provide a file provider if you want to support the open and move modes.

`NSLayoutAnchor`

The `NSLayoutAnchor` class is a factory class for creating `NSLayoutConstraint` objects using a fluent API. Use these constraints to programatically define your layout using Auto Layout.

`NSLayoutConstraint`

A constraint defines a relationship between two user interface objects that must be satisfied by the constraint-based layout system. Each constraint is a linear equation with the following format:

`NSLayoutDimension`

The `NSLayoutDimension` class is a factory class for creating size-based `NSLayoutConstraint` objects using a fluent API. Use these constraints to programatically define your layout using Auto Layout. All sizes are measured in points.

`NSLayoutManager`

An `NSLayoutManager` object coordinates the layout and display of characters held in an `NSTextStorage` object. It maps Unicode character codes to glyphs, sets the glyphs in a series of `NSTextContainer` objects, and displays them in a series of `NSTextView` objects. In addition to its core function of laying out text, an `NSLayoutManager` object coordinates its `NSTextView` objects, provides services to those text views to support `NSRulerView` instances for editing paragraph styles, and handles the layout and display of text attributes not inherent in glyphs (such as underline or strikethrough). You can create a subclass of `NSLayoutManager` to handle additional text attributes, whether inherent or not.

`NSLayoutXAxisAnchor`

The `NSLayoutXAxisAnchor` class is a factory class for creating horizontal `NSLayoutConstraint` objects using a fluent interface (an interface designed to produce more readable code). Use these constraints to programatically define your layout using Auto Layout.

`NSLayoutYAxisAnchor`

The `NSLayoutYAxisAnchor` class is a factory class for creating vertical `NSLayoutConstraint` objects using a fluent interface (an interface designed to produce more readable code). Use these constraints to programatically define your layout using Auto Layout.

`NSMutableParagraphStyle`

The `NSMutableParagraphStyle` class adds methods to its superclass, `NSParagraphStyle`, for changing the values of the subattributes in a paragraph style attribute. See the `NSParagraphStyle` and `NSAttributedString` specifications for more information.

`NSParagraphStyle`

The `NSParagraphStyle` class and its subclass `NSMutableParagraphStyle` encapsulate the paragraph or ruler attributes used by the `NSAttributedString` classes. Instances of these classes are often referred to as paragraph style objects or, when no confusion will result, paragraph styles.

`NSShadow`

An `NSShadow` object encapsulates the attributes used to create a drop shadow during drawing operations.

`NSStringDrawingContext`

The `NSStringDrawingContext` class manages metrics used when drawing attributed strings. Prior to drawing, you can create an instance of this class and use it to specify the minimum scale factor and tracking adjustments for a string. After drawing, you can retrieve the actual values that were used during drawing.

`NSTextAttachment`

`NSTextAttachment` objects are used by the `NSAttributedString` class cluster as the values for attachment attributes (stored in the attributed string under the key named

NSAttachmentAttributeName). The objects you create with this class are referred to as text attachment objects, or when no confusion will result, as text attachments or merely attachments.

NSTextContainer

The NSTextContainer class defines a region where text is laid out. An NSLayoutManager uses NSTextContainer to determine where to break lines, lay out portions of text, and so on. An NSTextContainer object normally defines rectangular regions, but you can define exclusion paths inside the text container to create regions where text does not flow. You can also subclass to create text containers with nonrectangular regions, such as circular regions, regions with holes in them, or regions that flow alongside graphics.

NSTextStorage

The NSTextStorage class defines the fundamental storage mechanism of TextKit. This class is a semi concrete subclass of NSMutableAttributedString that adds behavior for managing a set of client NSLayoutManager objects. A text storage object notifies its layout managers of changes to its characters or attributes, which lets the layout managers redisplay the text as needed.

NSTextTab

An NSTextTab object represents a tab in an NSParagraphStyle object, storing an alignment type and location. NSTextTab objects are most frequently used with the TextKit system and with NSRulerView and NSRulerMarker objects.

UIAccessibilityCustomAction

A UIAccessibilityCustomAction object represents a custom action to be performed on an accessible object. Apps that support custom actions can create instances of this class, specifying the user-readable name of the action and the object and selector to use when performing the action. Assistive technologies display custom actions in response to specific user cues. For example, VoiceOver lets users access actions quickly using the Actions rotor.

UIAccessibilityCustomRotor

UIAccessibilityCustomRotorItemResult

UIAccessibilityCustomRotorSearchPredicate

UIAccessibilityElement

The UIAccessibilityElement class encapsulates information about an item that should be accessible to users with disabilities, but that isn't accessible by default. For example, an icon or text image is not automatically accessible because it does not inherit from UIView (or UIControl). A view that contains such nonview items creates an instance of UIAccessibilityElement to represent each item that needs to be accessible.

~~UIActionSheet~~

Important: UIActionSheet is deprecated in iOS 8. (Note that UIActionSheetDelegate is also deprecated.) To create and manage action sheets in iOS 8 and later, instead use UIAlertController with a preferredStyle of actionSheet.

Deprecated

UIActivity

The UIActivity class is an abstract class that you subclass in order to implement application-specific services. A service takes data that is passed to it, does something to that data, and returns the results. For example, an social media service might take whatever text, images, or other content is provided to it and post them to the user's account. Activity objects are used in conjunction with a UIActivityViewController object, which is responsible for presenting services to the user.

UIActivityIndicatorView

Use an activity indicator to show that a task is in progress. An activity indicator appears as a "gear" that is either spinning or stopped.

`UIActivityItemProvider`

A `UIActivityItemProvider` object is a proxy for data passed to an activity view controller. You can use a provider object in situations where you want to make data available for use by an activity but you want to delay providing that data until it is actually needed. For example, you might use a provider object to represent a large video file that needs to be processed before it can be shared to a user's social media account.

`UIActivityViewController`

The `UIActivityViewController` class is a standard view controller that you can use to offer various services from your application. The system provides several standard services, such as copying items to the pasteboard, posting content to social media sites, sending items via email or SMS, and more. Apps can also define custom services.

`UIAlertAction`

A `UIAlertAction` object represents an action that can be taken when tapping a button in an alert. You use this class to configure information about a single action, including the title to display in the button, any styling information, and a handler to execute when the user taps the button. After creating an alert action object, add it to a `UIAlertController` object before displaying the corresponding alert to the user.

`UIAlertController`

A `UIAlertController` object displays an alert message to the user. This class replaces the `UIActionSheet` and `UIAlertView` classes for displaying alerts. After configuring the alert controller with the actions and style you want, present it using the `present(_:animated:completion:)` method.

~~`UIAlertView`~~

In apps that run in versions of iOS prior to iOS 8, use the `UIAlertView` class to display an alert message to the user. An alert view functions similar to but differs in appearance from an action sheet (an instance of `UIActionSheet`).

Deprecated

`UIApplication`

The `UIApplication` class provides a centralized point of control and coordination for apps running in iOS. Every app has exactly one instance of `UIApplication` (or, very rarely, a subclass of `UIApplication`). When an app is launched, the system calls the `UIApplicationMain(_:_:_:_:)` function; among its other tasks, this function creates a Singleton `UIApplication` object. Thereafter you access the object by calling the *shared()* class method.

`UIApplicationShortcutIcon`

An application shortcut, or *quick action*, icon is an image you can optionally associate with a Home screen quick action to improve its appearance and usability.

`UIApplicationShortcutItem`

An application shortcut item, also called a *Home screen dynamic quick action*, specifies a user-initiated action for your app.

`UIAttachmentBehavior`

item affect the movement of the other in a prescribed way. When an item is attached to an anchor point, the movement of that item is affected by its attachment to the specified anchor point. Some attachment behaviors support both two items and an anchor point.

`UIBarButtonItem`

A bar button item is a button specialized for placement on a `UIToolbar` or `UINavigationBar` object. It inherits basic button behavior from its abstract superclass, `UIBarItem`. The `UIBarButtonItem` defines additional initialization methods and properties for use on toolbars and navigation bars.

`UIBarButtonItemGroup`

A `UIBarButtonItemGroup` object manages a set of bar button items on the shortcuts bar above the keyboard on iPad. A group contains one or more bar button items and an optional representative item that is displayed instead of the individual items when space is constrained. You can create any number of groups and configure each group with any number of items.

`UIBarItem`

`UIBarItem` is an abstract superclass for items added to a bar that appears at the bottom of the screen. Items on a bar behave in a way similar to buttons (instances of `UIButton`). They have a title, image, action, and target. You can also enable and disable an item on a bar.

`UIBezierPath`

The `UIBezierPath` class lets you define a path consisting of straight and curved line segments and render that path in your custom views. You use this class initially to specify just the geometry for your path. Paths can define simple shapes such as rectangles, ovals, and arcs or they can define complex polygons that incorporate a mixture of straight and curved line segments. After defining the shape, you can use additional methods of this class to render the path in the current drawing context.

`UIBlurEffect`

A `UIBlurEffect` object applies a blurring effect to the content layered behind a

[illegible]

`UIButton`

A `UIButton` object is a view that executes your custom code in response to user interactions.

`UICloudSharingController`

`UICollectionReusableView`

The `UICollectionReusableView` class defines the behavior for all cells and supplementary views presented by a collection view. Reusable views are so named because the collection view places them on a reuse queue rather than deleting them when they are scrolled out of the visible bounds. Such a view can then be retrieved and repurposed for a different set of content.

`UICollectionView`

The `UICollectionView` class manages an ordered collection of data items and presents them using customizable layouts. Collection views provide the same general function as table views except that a collection view is able to support more than just single-column layouts. Collection views support customizable layouts that can be used to implement multi-column grids, tiled layouts, circular layouts, and many more. You can even change the layout of a collection view dynamically if you want.

`UICollectionViewCell`

A `UICollectionViewCell` object presents the content for a single data item when that item is within the collection view's visible bounds. You can use this class as-is or subclass it to add additional properties and methods. The layout and presentation of cells is managed by the collection view and its corresponding layout object.

`UICollectionViewController`

The `UICollectionViewController` class represents a view controller whose content consists of a collection view. It implements the following behavior:

The `UICollectionViewFlowLayout` class is a concrete layout object that organizes items into a grid with optional header and footer views for each section. The items in the collection view flow from one row or column (depending on the scrolling direction) to the next, with each row comprising as many cells as will fit. Cells can be the same sizes or different sizes.

`UICollectionViewFlowLayoutInvalidationContext`

A `UICollectionViewFlowLayoutInvalidationContext` object specifies properties for determining whether to recompute the size of items or their position in the layout. The flow layout object creates instances of this class when it needs to invalidate its contents in response to changes. You can also create instances when invalidating the flow layout manually.

`UICollectionViewFocusUpdateContext`

A `UICollectionViewFocusUpdateContext` object stores information specific to a focus update in a collection view. When focus changes, the collection view delegate receives a context object with the relevant information. Your delegate methods use the information in this object to create animations or to perform other tasks related to the change in focus.

`UICollectionViewLayout`

The `UICollectionViewLayout` class is an abstract base class that you subclass and use to generate layout information for a collection view. The job of a layout object is to determine the placement of cells, supplementary views, and decoration views inside the collection view's bounds and to report that information to the collection view when asked. The collection view then applies the provided layout information to the corresponding views so that they can be presented onscreen.

`UICollectionViewLayoutAttributes`

An `UICollectionViewLayoutAttributes` object manages the layout-related attributes for a given item in a collection view. Layout objects create instances of this class when asked to do so by the collection view. In turn, the collection view uses the layout information to position cells and supplementary views inside its bounds.

A `UICollectionViewLayoutInvalidationContext` object declares which parts of your layout need to be updated when the layout is invalidated. Layout objects that are designed to support invalidation contexts can use that information to optimize their behavior during the invalidation cycle.

`UICollectionViewTransitionLayout`

The `UICollectionViewTransitionLayout` class is a special type of layout object that lets you implement behaviors when changing from one layout to another in your collection view. You can use this class as-is or subclass it to provide specialized behavior for your app. A common use for transition layouts is to create interactive transitions, such as those that are driven by gesture recognizers or touch events.

`UICollectionViewUpdateItem`

The `UICollectionViewUpdateItem` class describes a single change to make to an item in a collection view. You do not create instances of this class directly. When updating its content, the collection view object creates them and passes them to the layout object's `prepare(forCollectionViewUpdates:)` method, which can use them to prepare the layout object for the upcoming changes.

`UICollisionBehavior`

A collision behavior confers, to a specified array of dynamic items, the ability of those items to engage in collisions with each other and with the behavior's specified boundaries. A collision behavior also specifies some characteristics of its items' collisions, with other characteristics optionally specified by a `UIDynamicItemBehavior` object.

`UIColor`

A `UIColor` object stores color data and sometimes opacity (alpha value). Many methods in UIKit require you to specify color data using a `UIColor` object; when drawing you use them to set the current fill and stroke colors. Color objects are immutable and can be used safely from multiple threads in your app.

`UIControl`

[illegible] which your app might use to facilitate navigation, gather user input, or manipulate content. Controls use the Target-Action mechanism to report user interactions to your app.

`UICubicTimingParameters`

Specifies timing information for animations in the form of a cubic Bézier curve.

`UIDatePicker`

A `UIDatePicker` object is a control used for the inputting of date and time values. You can use a date picker to allow a user to enter either a point in time (calendar date, time value or both) or a time interval (for example for a timer). The date picker reports interactions to its associated target object.

`UIDevice`

The `UIDevice` class provides a Singleton instance representing the current device. From this instance you can obtain information about the device such as assigned name, device model, and operating-system name and version.

`UIDictationPhrase`

A dictation phrase object represents the textual interpretation of a spoken phrase as dictated by a user.

`UIDocument`

An abstract base class for managing the data of documents.

`UIDocumentInteractionController`

A document interaction controller, along with a delegate object, provides in-app support for managing user interactions with files in the local system. For example, an email program might use this class to allow the user to preview attachments and open them in other apps. Use this class to present an appropriate user interface for previewing, opening, copying, or printing a specified file.

a given file type and mode. You can also add your own custom menu items to this list.

`UIDocumentPickerExtensionViewController`

The `UIDocumentPickerExtensionViewController` class is the principal class for the Document Picker View Controller extension. When creating a Document Picker extension, you must subclass `UIDocumentPickerExtensionViewController` to provide the document picker's user interface. Your subclass presents a list of available documents and destinations to the user. When the user makes a selection, you trigger the file transfer and pass the selected URL back to the host app.

`UIDocumentPickerViewController`

A `UIDocumentPickerViewController` object lets the user select documents or destinations outside your app's sandbox. You must enable iCloud document support before you can use the document picker. For more information, see iCloud Design Guide.

UIDynamicAnimator

A dynamic animator provides physics-related capabilities and animations for its dynamic items, and provides the context for those animations. It does this by intermediating between the underlying iOS physics engine and dynamic items, via behavior objects you add to the animator.

UIDynamicBehavior

A dynamic behavior confers a behavioral configuration on one or more dynamic items for their participation in two-dimensional animation.

UIDynamicItemBehavior

A dynamic item behavior represents a base dynamic animation configuration for one or more dynamic items. Each of its properties overrides a corresponding default value.

UIDynamicItemGroup

A `UIDynamicItemGroup` object represents a dynamic item comprised of multiple other dynamic items. Use groups to manipulate a group of dynamic items together and treat them as a single unit for the

UIEvent

A `UIEvent` object (or, simply, an event object) represents an event in iOS. There are three general types of event: touch events, motion events, and remote-control events. Remote-control events allow a responder object to receive commands from an external accessory or headset so that it can manage manage audio and video—for example, playing a video or skipping to the next audio track. Motion events were introduced in iOS 3.0 and remote-control events in iOS 4.0.

UIFeedbackGenerator

The abstract superclass for all feedback generators.

UIFieldBehavior

A `UIFieldBehavior` object applies field-based physics to dynamic items. A field behavior defines an area in which forces such as gravity, magnetism, drag, velocity, turbulence, and others can be applied. After creating a field behavior object of the appropriate type, configure the strength of the intended force along with any other field attributes.

`UIFocusAnimationCoordinator`

The `UIFocusAnimationCoordinator` class helps coordinate focus-related animations during a focus update. `UIFocusAnimationCoordinator` instances are always created by the system and vended to your application during a focus update, and are typically discarded once the update is complete. It is not useful to instantiate `UIFocusAnimationCoordinator` yourself. The `UIFocus` header file- and its related classes and protocol- creates a single high-level software interface for controlling focus in applications that use focus-based input. This programming interface also helps to control focus behavior on the screen.

`UIFocusGuide`

The `UIFocusGuide` class is designed to expose non view areas as focusable. `UIFocusGuide`, as subclasses of `UILayoutGuide` are not views and do not define a new view or participate in the view

its related classes and its protocol- creates a single high-level software interface for controlling focus in applications that use focus-based input. This programming interface also helps to control focus behavior on the screen.

`UIFocusUpdateContext`

The `UIFocusUpdateContext` class provides information relevant to a specific focus update from one view to another. The objects of this class are ephemeral and are usually discarded after the update is finished. The `UIFocus` APIs creates a single high level software interface for controlling focus in apps using focus-based input.

`UIFont`

The `UIFont` class provides the interface for getting and setting font information. The class provides you with access to the font’s characteristics and also provides the system with access to the font’s glyph information, which is used during layout. You use font objects by passing them to methods that accept them as a parameter.

`UIFontDescriptor`

`UIFontDescriptor` objects provide a mechanism to describe a font with a dictionary of attributes. This font descriptor can be used later to create or modify a `UIFont` object. Font descriptors can be archived and unarchived. Font descriptors have a font matching capability, so that you can partially describe a font by creating a font descriptor with, for example, just a family name. You can then find all the available fonts on the system with a matching family name using `matchingFontDescriptors(withMandatoryKeys:)`.

`UIGestureRecognizer`

`UIGestureRecognizer` is an abstract base class for concrete gesture-recognizer classes. A gesture-recognizer object—or, simply, a gesture recognizer—decouples the logic for recognizing a gesture and acting on that recognition. When one of these objects recognizes a common gesture or, in some cases, a change in the gesture, it sends an action message to each designated target object.

API Reference

UIKit | Show API Changes ⌄

`UIGraphicsImageRendererContext`

`UIGraphicsImageRendererFormat`

`UIGraphicsPDFRenderer`

`UIGraphicsPDFRendererContext`

`UIGraphicsPDFRendererFormat`

`UIGraphicsRenderer`

`UIGraphicsRendererContext`

`UIGraphicsRendererFormat`

`UIGravityBehavior`

A `UIGravityBehavior` object applies a gravity-like force to all of its associated dynamic items. The magnitude and direction of the gravity force are configurable and are applied equally to all associated items. Use this behavior to modify the position of views and other dynamic items in your app's interface.

`UIImage`

An object that manages image data in your app.

`UIImageAsset`

images of the same item at different display scales.

`UIImagePickerController`

The `UIImagePickerController` class manages customizable, system-supplied user interfaces for taking pictures and movies on supported devices, and for choosing saved images and movies for use in your app. An image picker controller manages user interactions and delivers the results of those interactions to a delegate object.

`UIImageView`

A `UIImageView` object displays a single image or a sequence of animated images in your interface. Image views let you efficiently draw any image that can be specified using a `UIImage` object. For example, you can use this class to display the contents of many standard image files, such as JPEG and PNG files. You can configure image views programmatically or in your storyboard file and change the images they display at runtime. For animated images, you can also use the methods of this class to start and stop the animation and specify other animation parameters.

`UIImpactFeedbackGenerator`

A concrete `UIFeedbackGenerator` subclass that creates haptics to simulate physical impacts.

`UIInputView`

The `UIInputView` class is designed to match the appearance of the standard system keyboard when used as an input view with a responder. When defining your own custom input views or input accessory views, you can use a `UIInputView` object as the root view and add any subviews you want to create your input view. The input view and its subviews receive tinting and blur effects based on the options you specify at initialization time.

`UIInputViewController`

To create a custom keyboard, start by subclassing the `UIInputViewController` class. Add your keyboard's user interface to the `inputView` property of your subclass.

A `UIInterpolatingMotionEffect` object maps the horizontal or vertical tilt of a device to values that you specify so that UIKit can apply those values to your views. You use this class to determine the amount of tilt along a single axis to apply to a view.

`UIKeyCommand`

The `UIKeyCommand` class specifies a key presses performed on a hardware keyboard and the resulting action that should take place. Hardware keyboards allow a user to hold down the Control, Option, Command, or other modifier key and press a key in combination to initiate commands such as cut, copy, or paste. You can use instances of this class to define custom command sequences that your app recognizes and then provide an appropriate response.

`UILabel`

The `UILabel` class implements a read-only text view. You can use this class to draw one or multiple lines of static text, such as those you might use to identify other parts of your user interface. The base `UILabel` class provides support for both simple and complex styling of the label text. You can also control over aspects of appearance, such as whether the label uses a shadow or draws with a highlight. If needed, you can customize the appearance of your text further by subclassing.

`UILayoutGuide`

The `UILayoutGuide` class defines a rectangular area that can interact with Auto Layout. Use layout guides to replace the dummy views you may have created to represent inter-view spaces or encapsulation in your user interface.

`UILexicon`

A lexicon contains a read-only array of term pairs, each in a `UILexiconEntry` object, for use by a custom keyboard.

`UILexiconEntry`

A lexicon entry specifies a read-only term pair available within a `UILexicon` object, for use by a custom

`UILocalizedIndexedCollation`

The `UILocalizedIndexedCollation` class is a convenience for organizing, sorting, and localizing the data for a table view that has a section index. The table view’s data source then uses the collation object to provide the table view with input for section titles and section index titles.

~~`UILocalNotification`~~

Important:`UILocalNotification` is deprecated in iOS 10. Use `UNNotificationRequest` instead. A `UILocalNotification` object specifies a notification that an app can schedule for presentation at a specific date and time.

Deprecated

`UILongPressGestureRecognizer`

`UILongPressGestureRecognizer` is a concrete subclass of `UIGestureRecognizer` that looks for long-press gestures. The user must press one or more fingers on a view and hold them there for a minimum period of time before the action triggers. While down, the user's fingers may not move more than a specified distance; if they move beyond the specified distance, the gesture fails.

`UIManagedDocument`

`UIManagedDocument` is a concrete subclass of `UIDocument` that integrates with Core Data. When you initialize a managed document, you specify the URL for the document location. The document object then creates a Core Data stack to use to access the document's persistent store using a Managed object model from the application's main bundle. See Using Document Storage with iCloud in iCloud Programming Guide for Core Data for implementation strategies and troubleshooting steps.

`UIMarkupTextPrintFormatter`

Instances of the `UIMarkupTextPrintFormatter` class lay out HTML markup text for a multipage print job.

`UIMenuController`

`UIMenuItem`

An instance of the `UIMenuItem` class represents a custom item in the editing menu managed by the `UIMenuController` object.

`UIMotionEffect`

The `UIMotionEffect` class is an abstract superclass for defining motion-based modifiers for views. Subclasses are responsible for defining the behavior to apply to a view when motion is detected. They do this by overriding the `keyPathsAndRelativeValues(forViewerOffset:)` method and returning one or more key paths representing the view properties to modify.

`UIMotionEffectGroup`

The UIMotionEffectGroup class manages a collection of motion effects that you want to apply to a view at the same time. This class behaves similarly to the CAAnimationGroup class in Core Animation. The key paths and values returned by each motion effect object are applied simultaneously and with the same timing.

UIMutableApplicationShortcutItem

A mutable application shortcut item, also called, verbosely, a *mutable Home screen dynamic quick action*, specifies a configurable user-initiated action for your app. This class is a convenience subclass of UIApplicationShortcutItem, helping you work with registered, and therefore immutable, quick actions.

~~UIMutableUserNotificationAction~~

Important: UIMutableUserNotificationAction is deprecated in iOS 10. Use UNNotificationAction instead. A UIMutableUserNotificationAction object represents a modifiable version of the UIUserNotificationAction class.

Deprecated

Important: UIMutableUserNotificationCategory is deprecated in iOS 10. Use UNNotificationCategory instead. A UIMutableUserNotificationCategory object encapsulates information about custom actions that your app can perform in response to a local or push notification. Use instances of this class to customize the actions included in an alert when space onscreen is constrained.

Deprecated

UINavigationBar

Provides a control for navigating hierarchical content in an app.

UINavigationController

The UINavigationController class implements a specialized view controller that manages the navigation of hierarchical content. This **navigation interface** makes it possible to present your data efficiently and makes it easier for the user to navigate that content. You generally use this class as-is but you may also subclass to customize the class behavior.

`UINavigationItem`

A `UINavigationItem` object manages the buttons and views to be displayed in a `UINavigationBar` object. When building a navigation interface, each view controller pushed onto the navigation stack must have a `UINavigationItem` object that contains the buttons and views it wants displayed in the navigation bar. The managing `UINavigationController` object uses the navigation items of the topmost two view controllers to populate the navigation bar with content.

`UINib`

Instances of the `UINib` class serve as object wrappers, or containers, for Interface Builder nib files.

`UINotificationFeedbackGenerator`

A concrete `UIFeedbackGenerator` subclass that creates haptics to communicate successes, failures, and warnings.

A control that displays a horizontal series of dots, each of which corresponds to a page in the application’s document (or other data-model entity).

`UIPageViewController`

A page view controller lets the user navigate between pages of content, where each page is managed by its own view controller object. Navigation can be controlled programmatically by your app or directly by the user using gestures. When navigating from page to page, the page view controller uses the transition that you specify to animate the change.

`UIPanGestureRecognizer`

`UIPanGestureRecognizer` is a concrete subclass of `UIGestureRecognizer` that looks for panning (dragging) gestures. The user must be pressing one or more fingers on a view while they pan it. Clients implementing the action method for this gesture recognizer can ask it for the current translation and velocity of the gesture.

UIPasteboard

Use the UIPasteboard class to let a user to share data from one place to another within your app, and from your app to other apps. For sharing data with any other app, use the systemwide general pasteboard; for sharing data with another app from your team—that has the same team ID as the app to share from—use named pasteboards.

UIPercentDrivenInteractiveTransition

A percent-driven interactive transition object drives the custom animation between the disappearance of one view controller and the appearance of another. It relies on a transition animator delegate—a custom object that adopts the UIViewControllerAnimatedTransitioning protocol—to set up and perform the animations.

UIPickerView

UIPinchGestureRecognizer

UIPinchGestureRecognizer is a concrete subclass of UIGestureRecognizer that looks for pinching gestures involving two touches. When the user moves the two fingers toward each other, the conventional meaning is zoom-out; when the user moves the two fingers away from each other, the conventional meaning is zoom-in.

UIPopoverBackgroundView

The UIPopoverBackgroundView class provides the background appearance for a popover. This class must be subclassed before it can be used. The implementation of your subclass is responsible for providing the border decoration and arrow for the popover. Subclasses must override all declared properties and methods to provide information about where to lay out the corresponding popover content and arrow. Subclasses must also provide implementations for all methods of the UIPopoverBackgroundViewMethods protocol.

~~UIPopoverController~~

Important: UIPopoverController is deprecated in iOS 9. In iOS 9 and later, a popover is implemented as a UIViewController presentation. To create a popover, use an instance of UIPopoverPresentationController and specify the popover style.

Deprecated

UIPopoverPresentationController

A UIPopoverPresentationController object manages the display of content in a popover. From the time a popover is presented until the time it is dismissed, UIKit uses an instance of this class to manage the presentation behavior. You use instances of this class as-is to configure aspects of the popover appearance and behavior for view controllers whose presentation style is set to popover.

UIPresentationController

[illegible]

UIKit uses a presentation controller to manage various aspects of the presentation process for that view controller. The presentation controller can add its own animations on top of those provided by animator objects, it can respond to size changes, and it can manage other aspects of how the view controller is presented onscreen.

UIPress

A UIPress object represents the presence or movement of a button press on the screen for a particular event. The press specifically encapsulates the pressing of some physically actuated button. All of the press types represent actual physical buttons on one of a variety of remotes. You access UIPress objects through UIEvent objects passed into responder objects for event handling. The gestureRecognizers property returns the gesture recognizers—that is, instances of a concrete subclass of UIGestureRecognizer—that are currently handling the given button press.

UIPressesEvent

A UIPressesEvent object is an event that describes the state of a set of physical buttons that are available to the device, such as those on an associated remote or game controller.

UIPreviewAction

A preview action, or *peek quick action*, is displayed below a peek when a user swipes the peek upward. A peek quick action typically selects a deep link to your app and has a title, a style, and a handler.

UIPreviewActionGroup

A preview quick action group contains one or more child quick actions, each an instance of the UIPreviewAction class.

UIPreviewInteraction

UIPrinter

A UIPrinter object contains the attributes of a printer on the network. You use a printer object to

UIPrinterPickerController

A UIPrinterPickerController object displays the system interface for selecting a printer. You can use a printer picker controller to display a list of printers to the user prior to printing a document, photo, or other content. Printer pickers display all pickers normally but you can filter out printers by assigning an appropriate delegate object to the picker before displaying it.

UIPrintFormatter

UIPrintFormatter is an abstract base class for print formatters: objects that lay out custom printable content that can cross page boundaries. Given a print formatter, the printing system can automate the printing of the type of content associated with the print formatter.

`UIPrintInfo`

A `UIPrintInfo` object encapsulates information about a print job, including printer identifier, job name, output type (photo, normal, grayscale), orientation (portrait or landscape), and any selected duplex mode. This information is used by the printing system when it prints.

`UIPrintInteractionController`

The shared instance of the `UIPrintInteractionController` class presents a printing user interface and manages the printing of documents, images, and other printable content in iOS.

`UIPrintPageRenderer`

A `UIPrintPageRenderer` object draws pages of content that are to be printed, with or without the assistance of print formatters.

`UIPrintPaper`

An instance of the `UIPrintPaper` class encapsulates the size of paper used for a print job and the rectangle in which content can be printed.

You use the `UIProgressView` class to depict the progress of a task over time. An example of a progress bar is the one shown at the bottom of the Mail application when it’s downloading messages.

`UIPushBehavior`

A push behavior applies a continuous or instantaneous force to one or more dynamic items, causing those items to change position accordingly.

`UIReferenceLibraryViewController`

A `UIReferenceLibraryViewController` object provides a dictionary service to look up the definition of a word or term from within an app. It should not be used to display wordlists, create a standalone dictionary app, or republish the content in any form.

`UIRefreshControl`

A `UIRefreshControl` object provides a standard control that can be used to initiate the refreshing of a table view’s contents. You link a refresh control to a table through an associated table view controller object. The table view controller handles the work of adding the control to the table’s visual appearance and managing the display of that control in response to appropriate user gestures.

`UIRegion`

A `UIRegion` object defines shape for use in UIKit Dynamics. When creating animations, you use regions to define the effective area of a field behavior such as a magnetic or gravitational force. Most regions are rectangular or elliptical in shape, but you can use the methods of this class to create more complex shapes by adding, subtracting, and intersecting other regions.

`UIResponder`

The `UIResponder` class defines an interface for objects that respond to and handle events. It is the superclass of `UIApplication`, `UIView` and its subclasses (which include `UIWindow`). Instances of these classes are sometimes referred to as responder objects or, simply, responders.

`UIRotationGestureRecognizer` is a concrete subclass of `UIGestureRecognizer` that looks for rotation gestures involving two touches. When the user moves the fingers opposite each other in a circular motion, the underlying view should rotate in a corresponding direction and speed.

`UIScreen`

A `UIScreen` object defines the properties associated with a hardware-based display. iOS devices have a main screen and zero or more attached screens. A tvOS device has a main screen for the television connected to the device. Use this class to obtain screen objects for each display attached to the device. Each screen object defines the bounds rectangle for the associated display and other interesting properties such as its brightness.

`UIScreenEdgePanGestureRecognizer`

A `UIScreenEdgePanGestureRecognizer` looks for panning (dragging) gestures that start near an edge of the screen. The system uses screen edge gestures in some cases to initiate view controller transitions. You can use this class to replicate the same gesture behavior for your own actions.

`UIScreenMode`

A `UIScreenMode` object represents a possible set of attributes that can be applied to a `UIScreen` object. The object encapsulates information about the size of the screen's underlying display buffer and the aspect ratio it uses for individual pixels.

`UIScrollView`

The `UIScrollView` class provides support for displaying content that is larger than the size of the application's window. It enables users to scroll within that content by making swiping gestures, and to zoom in and back from portions of the content by making pinching gestures.

`UISearchBar`

The `UISearchBar` class implements a text field control for text-based searches. The control provides a text field for entering text, a search button, a bookmark button, and a cancel button. The `UISearchBar`

[illegible]
clicked.

`UISearchContainerViewController`

A `UISearchContainerViewController` object is a wrapper for search results that you want to embed in a container view controller. Although you can present a `UISearchController` object modally, you should never push one onto a navigation controller's stack or use one as a child of another container view controller. Instead, embed an instance of this class and let it manage the presentation of the search controller's content.

`UISearchController`

A `UISearchController` object manages the display of search results based on interactions with a search bar. You use a search controller in tandem with your existing view controllers. When you have a view controller with searchable content, incorporate the search bar of a `UISearchController` object into your view controller's interface. When the user interacts with that search bar, the search controller automatically displays a new view controller with the search results that you specify.

~~`UISearchDisplayController`~~

Important: `UISearchDisplayController` is deprecated in iOS 8. (Note that `UISearchDisplayDelegate` is also deprecated.) To manage the presentation of a search bar and display search results in iOS 8 and later, instead use `UISearchController`.

Deprecated

`UISegmentedControl`

A `UISegmentedControl` object is a horizontal control made of multiple segments, each segment functioning as a discrete button. A segmented control affords a compact means to group together a number of controls.

`UISelectionFeedbackGenerator`

[illegible]

`UISimpleTextPrintFormatter`

Instances of the `UISimpleTextPrintFormatter` class lay out plain text for printing, possibly over multiple pages. The class allows you to specify global font, color, and text alignment properties for the printed text.

`UISlider`

A visual control used to select a single value from a continuous range of values.

`UISnapBehavior`

A snap behavior defines a dynamic item's movement to a specified point; the movement proceeds with a spring-like effect, ending with an oscillation whose amount you can set.

`UISplitViewController`

The `UISplitViewController` class is a container view controller that presents a master-detail interface. In a master-detail interface, changes in the primary view controller (the master) drive changes in a secondary view controller (the detail). The two view controllers can be arranged so that they are side-by-side, so that only one at a time is visible, or so that one only partially hides the other. In iOS 8 and later, you can use the `UISplitViewController` class on all iOS devices; in previous versions of iOS, the class is available only on iPad.

`UISpringTimingParameters`

Specifies timing information for animations that mimics the behavior of a spring.

`UIStackView`

The `UIStackView` class provides a streamlined interface for laying out a collection of views in either a column or a row. Stack views let you leverage the power of Auto Layout, creating user interfaces that can dynamically adapt to the device's orientation, screen size, and any changes in the available space. 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. The exact

`UIStepper`

A stepper control provides a user interface for incrementing or decrementing a value. A stepper displays two buttons, one with a minus ("–") symbol and one with a plus ("+") symbol.

`UIStoryboard`

A `UIStoryboard` object encapsulates the view controller graph stored in an Interface Builder storyboard resource file. This view controller graph represents the view controllers for all or part of your application's user interface. Normally, view controllers in a storyboard are instantiated and created automatically in response to actions defined within the storyboard itself. However, you can use a storyboard object to instantiate the initial view controller in a storyboard file or instantiate other view controllers that you want to present programmatically.

~~UIStoryboardPopoverSegue~~

The UIStoryboardPopoverSegue class defines a specific type of segue for presenting content in a popover. For popover segues, the destination view controller contains the content to be displayed in the popover. This class provides an additional popoverController property so that your custom code has access to the popover controller object. For example, you might want to store the popover controller elsewhere in your code so that you can dismiss the popover programmatically.

Deprecated

UIStoryboardSegue

Prepares for and performs the visual transition between two view controllers.

UIStoryboardUnwindSegueSource

A UIStoryboardUnwindSegueSource object encapsulates information about an unwind segue. The information includes the view controller being dismissed by the unwind segue and the action method responsible for the dismissal.

gestures in one or more directions. A swipe is a discrete gesture, and thus the associated action message is sent only once per gesture.

UISwitch

You use the UISwitch class to create and manage the On/Off buttons used, for example, in the Settings app for options such as Airplane Mode and Bluetooth. These objects are known as switches.

UITabBar

A UITabBar object is a control for selecting between different subtasks, views, or modes in an app. Normally, you use tab bars in conjunction with a UITabBarController object, but you can also use them as standalone controls in your app. Tab bars always appear across the bottom edge of the screen and display the contents of one or more UITabBarItem objects. A tab bar's appearance can be customized with a background image or tint color to suit the needs of your interface. Tapping an item selects and highlights that item, and you use the selection of the item to enable the corresponding mode for your app.

`UITabBarController`

The `UITabBarController` class implements a specialized view controller that manages a radio-style selection interface. This *tab bar interface* displays tabs at the bottom of the window for selecting between the different modes and for displaying the views for that mode. This class is generally used as-is but may be subclassed in iOS 6 and later.

`UITabBarItem`

The `UITabBarItem` class implements an item on a `UITabBar` object. A tab bar operates strictly in radio mode, where one item is selected at a time—tapping a tab bar item toggles the view above the tab bar. You can also specify a badge value on the tab bar item for adding additional visual information—for example, the Messages app uses a badge on the item to show the number of new messages. This class also provides a number of system defaults for creating items.

`UITableView`

`UITableViewCell`

The `UITableViewCell` class defines the attributes and behavior of the cells that appear in `UITableView` objects. This class includes properties and methods for setting and managing cell content and background (including text, images, and custom views), managing the cell selection and highlight state, managing accessory views, and initiating the editing of the cell contents.

`UITableViewController`

The `UITableViewController` class creates a controller object that manages a table view. It implements the following behavior:

`UITableViewFocusUpdateContext`

The `UITableViewFocusUpdateContext` is a subclass of the `UIFocusUpdateContext` and provide information relevant to a specific focus update from one view to another. These subclasses provide extra information that is only relevant to focus updates involving tableviews. Instances of this class are ephemeral and are usually discarded after the update is finished.

`UITableViewHeaderFooterView`

The `UITableViewHeaderFooterView` class implements a reusable view that can be placed at the top or bottom of a table section. You use headers and footers to display additional information for that section.

`UITableViewRowAction`

A `UITableViewRowAction` object defines a single action to present when the user swipes horizontally in a table row. In an editable table, performing a horizontal swipe in a row reveals a button to delete the row by default. This class lets you define one or more custom actions to display for a given row in your table. Each instance of this class represents a single action to perform and includes the text, formatting information, and behavior for the corresponding button.

`UITapGestureRecognizer`

specified number of times.

`UITextChecker`

You use instances of the `UITextChecker` class to check a string (usually the text of a document) for misspelled words.

`UITextField`

A `UITextField` object displays an editable text area in your interface. You use text fields to gather text-based input from the user using the onscreen keyboard. The keyboard is configurable for many different types of input such as plain text, emails, numbers, and so on. Text fields use the target-action mechanism and a delegate object to report changes made during the course of editing.

UITextInputAssistantItem

A UITextInputAssistantItem object manages bar button items displayed in the shortcuts bar above the keyboard on iPad. Use this object to add app-specific actions to the shortcuts bar. The center of the shortcuts bar displays typing suggestions for the user. You can install custom bar button items that lead or trail the typing suggestions.

UITextInputMode

An instance of the UITextInputMode class represents the current text-input mode. You can use this object to determine the primary language currently being used for text input.

UITextInputStringTokenizer

The UITextInputStringTokenizer class is a base implementation of the UITextInputTokenizer protocol provided by the UIKit framework.

UITextPosition

A UITextPosition object represents a position in a text container; in other words, it is an index into the

UITextRange

A UITextRange object represents a range of characters in a text container; in other words, it identifies a starting index and an ending index in string backing a text-entry object.

UITextSelectionRect

The UITextSelectionRect class encapsulates information about a selected range of text in a document. This class is an abstract class and must be subclassed to be used. The system text input views provide their own concrete implementations of this class.

UITextView

The UITextView class implements the behavior for a scrollable, multiline text region. The class supports the display of text using custom style information and also supports text editing. You typically use a text view to display multiple lines of text, such as when displaying the body of a large text document.

UIToolbar

A toolbar is a control that displays one or more buttons, called toolbar items. A toolbar momentarily highlights or does not change the appearance of an item when tapped.

UITouch

A UITouch object represents the location, size, movement, and force of a finger on the screen for a particular event. The force of a touch is available starting in iOS 9 on devices that support 3D Touch or Apple Pencil.

UITraitCollection

A trait collection describes the iOS interface environment for your app, including traits such as horizontal and vertical size class, display scale, and user interface idiom. To create an adaptive interface, write code to adjust your app's layout according to changes in these traits.

~~UIUserNotificationAction~~

response to a remote or local notification.

Deprecated

~~UIUserNotificationCategory~~

Important:UIUserNotificationCategory is deprecated in iOS 10. Use UNNotificationCategory instead. A UIUserNotificationCategory object encapsulates information about custom actions that your app can perform in response to a local or push notification.

Deprecated

~~UIUserNotificationSettings~~

Important: UIUserNotificationSettings is deprecated in iOS 10. Use UNNotificationSettings instead. A UIUserNotificationSettings object encapsulates the types of notifications that can be displayed to the user by your app.

Deprecated

UIVibrancyEffect

A `UIVibrancyEffect` object amplifies and adjusts the color of the content layered behind a `UIVisualEffectView` object, allowing the content placed inside of the `contentView` to become more vivid.

UIVideoEditorController

A `UIVideoEditorController` object, or video editor, manages the system-supplied user interface for trimming video frames from the start and end of a previously recorded movie as well as reencoding to lower quality. The object manages user interactions and provides the filesystem path of the edited movie to your delegate object (see `UIVideoEditorControllerDelegate`). The features of the `UIVideoEditorController` class are available only on devices that support video recording.

UIView

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

UIViewController

The `UIViewController` class provides the infrastructure for managing the views of your iOS apps. A view controller manages a set of views that make up a portion of your app's user interface. It is responsible for loading and disposing of those views, for managing interactions with those views, and for coordinating responses with any appropriate data objects. View controllers also coordinate their efforts with other controller objects—including other view controllers—and help manage your app's overall interface.

UIViewPrintFormatter

An instance of the `UIViewPrintFormatter` class lays out the drawn content of a view for printing. The view's content can span multiple pages.

UIViewPropertyAnimator

Animates changes to views and allows the dynamic modification of those animations.

`UIVisualEffect`

This class contains no methods and is intended as a way to initialize a `UIVisualEffectView` with `UIBlurEffect` and `UIVibrancyEffect` objects.

`UIVisualEffectView`

A `UIVisualEffectView` object gives you an easy way implement some complex visual effects. Depending on the desired effect, the effect may affect content layered behind the view or content added to the visual effect view's `contentView`.

`UIWebView`

You can use the `UIWebView` class to embed web content in your app. To do so, create a `UIWebView` object, attach it to a window, and send it a request to load web content. You can also use this class to move back and forward in the history of webpages, and you can even set some web content properties programmatically.

API Reference

UIKit | Show API Changes

A `UIWindow` object provides the backdrop for your app's user interface and provides important event-handling behaviors. Windows do not have any visual appearance of their own, but they are crucial to the presentation of your app's views. Every view that appears onscreen is enclosed by a window, and each window is independent of the other windows in your app. Events received by your app are initially routed to the appropriate window object, which in turn forwards those events to the appropriate view. Windows work with your view controllers to implement orientation changes and to perform many other tasks that are fundamental to your app's operation.

Protocols

`NSLayoutManagerDelegate`

The `NSLayoutManagerDelegate` protocol defines the optional methods implemented by delegates of `NSLayoutManager` objects.

`NSTextAttachmentContainer`

The `NSTextAttachmentContainer` protocol defines the interface to text attachment objects from `NSLayoutManager`.

`NSTextLayoutOrientationProvider`

The `NSTextLayoutOrientationProvider` protocol defines the orientation of text for an object. In macOS, the `NSTextContainer` and `NSTextView` classes adopt this protocol; in iOS, only the `NSTextContainer` class implements it. A `NSTextContainer` object returns the value from its associated text view when present; otherwise, it returns `horizontal` by default. If you define a custom `NSTextContainer` object, you can override this method and return `vertical` to support laying out text vertically.

`NSTextStorageDelegate`

The `NSTextStorageDelegate` protocol defines the optional methods implemented by delegates of `NSTextStorage` objects.

data from the system. Implementation of this method is optional, but expected.

UIAccessibility

The `UIAccessibility` informal protocol provides accessibility information about an application’s user interface elements. Assistive applications, such as VoiceOver, convey this information to users with disabilities to help them use the application.

UIAccessibilityAction

The `UIAccessibilityAction` informal protocol provides a way for accessibility elements to support specific actions, such as selecting values in a range or scrolling through information on the screen. For example, to respond to a scrolling gesture, you implement the `accessibilityScroll(_:)` method and post `UIAccessibilityPageScrolledNotification` with the new page status (such as “Page 3 of 9”). Or, to make an element such as a slider or picker view accessible, you first need to characterize it by including the `UIAccessibilityTraitAdjustable` trait. Then, you must implement the `accessibilityIncrement()` and `accessibilityDecrement()` methods. When you do this, assistive technology users can adjust the element using gestures specific to the assistive technology.

UIAccessibilityContainer

The `UIAccessibilityContainer` informal protocol provides a way for `UIView` subclasses to make selected components accessible as separate elements. For example, a view might contain icons or drawn text that, to end users, appear and function as separate items. But because these components are not implemented as instances of `UIView`, they are not automatically accessible to users with disabilities. Therefore, such a container view should implement the `UIAccessibilityContainer` methods to supply accessibility information about these components to assistive applications such as VoiceOver.

UIAccessibilityFocus

The `UIAccessibilityFocus` informal protocol provides a way to find out whether an assistive technology, such as VoiceOver, is focused on an accessible element.

in your user interface. You can use the identifiers you define in UI Automation scripts because the value of `accessibilityIdentifier` corresponds to the return value of the `name` method of *UIAElement*.

`UIAccessibilityReadingContent`

The `UIAccessibilityReadingContent` protocol can be implemented on an object that represents content that is intended to be read by users, such as a book or an article. To give VoiceOver users a superior, continuous reading experience, you can implement this protocol on such an element, characterize it with the `UIAccessibilityTraitCausesPageTurn` trait, and use the `UIAccessibilityScrollDirectionNext` and `UIAccessibilityScrollDirectionPrevious` constants to enable page turning.

`UIActionSheetDelegate`

Important: UIActionSheetDelegate is deprecated in iOS 8. (Note that UIActionSheet is also deprecated.) To create and manage action sheets in iOS 8 and later, use UIAlertController.

UIActivityItemSource

The UIActivityItemSource protocol defines the methods used by a UIActivityViewController object to retrieve the data items to act on. You can use this protocol in situations where you want to provide the data from one of your app's existing objects instead of creating a separate UIActivityItemProvider object. When implementing this protocol, your object becomes the data provider, providing the view controller with access to the items.

UIAdaptivePresentationControllerDelegate

An object that conforms to the UIAdaptivePresentationControllerDelegate protocol works with a presentation controller to determine how to respond to trait changes in your app. Your delegate can suggest a new presentation style or an entirely new view controller for displaying content.

UIAlertViewDelegate

For apps that run in versions of iOS prior to iOS 8, the UIAlertViewDelegate protocol defines the

[illegible]

UIAppearance

Use the UIAppearance protocol to get the appearance proxy for a class. You can customize the appearance of instances of a class by sending appearance modification messages to the class's appearance proxy.

UIAppearanceContainer

A class must adopt the UIAppearance protocol to allow appearance customization using the UIAppearance API.

UIApplicationDelegate

The UIApplicationDelegate protocol defines methods that are called by the singleton UIApplication object in response to important events in the lifetime of your app.

UIBarPositioning

The UIBarPositioning protocol defines the ways that bars can be positioned on iOS devices. Bars can be positioned at the bottom of their enclosing view, at the top of their enclosing view, or at both the top of their enclosing view and also the top of the screen. In this last case, the bar will abut the status bar displayed by the system. Bars in this position need to have their background extend above their own frame to the top of the screen. This allows the background to show through the status bar.

UIBarPositioningDelegate

The UIBarPositioningDelegate protocol supports the positioning of a bar that conforms to the UIBarPositioning protocol. Navigation bars, toolbars, and search bars all have delegates that support this protocol. The delegate can use the method of this protocol to specify the bar's position when that bar is moved to a window.

UICloudSharingControllerDelegate

UICollectionViewDataSource

An object that adopts the UICollectionViewDataSource protocol is responsible for providing the data and views required by a collection view. A data source object represents your app's data model and vends information to the collection view as needed. It also handles the creation and configuration of cells and supplementary views used by the collection view to display your data.

UICollectionViewDataSourcePrefetching

A protocol that provides advance warning of the data requirements for a collection view, allowing the triggering of asynchronous data load operations.

UICollectionViewDelegate

The UICollectionViewDelegate protocol defines methods that allow you to manage the selection and highlighting of items in a collection view and to perform actions on those items. The methods of this protocol are all optional.

`UICollectionViewDelegateFlowLayout`

The `UICollectionViewDelegateFlowLayout` protocol defines methods that let you coordinate with a `UICollectionViewFlowLayout` object to implement a grid-based layout. The methods of this protocol define the size of items and the spacing between items in the grid.

`UICollisionBehaviorDelegate`

To respond to UIKit dynamic item collisions, configure a custom class to adopt the `UICollisionBehaviorDelegate` protocol. Then, in a collision behavior (an instance of the `UICollisionBehavior` class), set the delegate to be an instance of your custom class.

`UIContentContainer`

The methods of the `UIContentContainer` protocol help you adapt the contents of your view controllers to size and trait changes. All `UIViewController` and `UIPresentationController` objects provide default implementations for the methods of this protocol. When creating your own custom view

`UIContentSizeCategoryAdjusting`

`UICoordinateSpace`

The `UICoordinateSpace` protocol defines methods for converting between different frames of reference on a screen. The `UIView` class adopts this protocol so that you can convert easily between most coordinate spaces in your app. The `UIScreen` class includes the `coordinateSpace` and `fixedCoordinateSpace` properties, which give you access to the screen's coordinate spaces. You can adopt this protocol in your own classes to convert between your custom coordinate spaces and the coordinate spaces of your app's views and screens.

`UIDataSourceModelAssociation`

The `UIDataSourceModelAssociation` protocol defines an interface for providing persistent references to data objects in your app. Your data source objects can adopt this protocol in order to assist a corresponding table or collection view during the state restoration process. Those classes use the methods of this protocol to ensure that the same data objects (and not just the same row indexes) are scrolled into view and selected.

`UIDocumentInteractionControllerDelegate`

The `UIDocumentInteractionControllerDelegate` protocol includes methods you can implement to respond to messages from a document interaction controller. Use this protocol to participate when document previews are displayed and when a document is about to be opened by another application. You can also use this protocol to respond to commands (such as "copy" and "print") from a document interaction controller's options menu.

`UIDocumentMenuDelegate`

The `UIDocumentMenuDelegate` protocol defines the methods you must implement to track user interactions with a document menu view controller. The document menu calls these methods when the user selects a document picker or dismisses the menu. If the user selects a document picker, set the

`UIDocumentPickerDelegate`

The `UIDocumentPickerDelegate` protocol defines the methods you must implement to track when the user selects a document or destination, or to track when the document picker is canceled.

`UIDynamicAnimatorDelegate`

To respond to the pausing or resumption of UIKit dynamic animation, configure a custom class to adopt the `UIDynamicAnimatorDelegate` protocol. Then, in a dynamic animator (an instance of the `UIDynamicAnimator` class), set the delegate to be an instance of your custom class.

`UIDynamicItem`

To make a custom object eligible to participate in UIKit Dynamics, adopt the `UIDynamicItem` protocol in the object's class.

`UIFocusEnvironment`

A protocol that defines the focus behavior for a branch of the view hierarchy.

`UIFocusItem`

A protocol—not intended for conformance by third-party classes—that lets an item declare its ability to participate in focus.

`UIGestureRecognizerDelegate`

Delegates of a gesture recognizer—that is, an instance of a concrete subclass of `UIGestureRecognizer` —adopt the `UIGestureRecognizerDelegate` protocol to fine-tune an app's gesture-recognition behavior. The delegates receive messages from a gesture recognizer, and their responses to these messages enable them to affect the operation of the gesture recognizer or to specify a relationship between it and another gesture recognizer, such as allowing simultaneous recognition or setting up a dynamic failure requirement.

`UIGuidedAccessRestrictionDelegate`

`UIImagePickerControllerDelegate`

The `UIImagePickerControllerDelegate` protocol defines methods that your delegate object must implement to interact with the image picker interface. The methods of this protocol notify your delegate when the user either picks an image or movie, or cancels the picker operation.

`UIInputViewAudioFeedback`

The `UIInputViewAudioFeedback` protocol defines a single property for enabling a custom input or keyboard accessory view to play standard keyboard input clicks.

`UIKeyInput`

A subclass of `UIResponder` can adopt this protocol to implement simple text entry. When instances of this subclass are the first responder, the system keyboard is displayed.

`UILayoutSupport`

This protocol is implemented by the `UIViewController` properties `topLayoutGuide` and `bottomLayoutGuide` to support using Auto Layout with a view controller's view, starting in iOS 7. You can use layout guides as layout items in the `NSLayoutConstraint` factory methods.

`UINavigationBarDelegate`

The `UINavigationBarDelegate` protocol defines optional methods that a `UINavigationBar` delegate should implement to update its views when items are pushed and popped from the stack. The navigation bar represents only the bar at the top of the screen, not the view below. It's the application's responsibility to implement the behavior when the top item changes.

`UINavigationControllerDelegate`

Use a navigation controller delegate (a custom object that implements this protocol) to modify behavior when a view controller is pushed or popped from the navigation stack of a `UINavigationController` object.

The `UIObjectRestoration` protocol should be adopted by classes that act as "restoration classes" for objects during the state restoration process. The method in this protocol should be used to return the object if it already exists or create it if needed.

`UIPageViewControllerDataSource`

The `UIPageViewControllerDataSource` protocol is adopted by an object that provides view controllers to the page view controller on an as-needed basis, in response to navigation gestures.

`UIPageViewControllerDelegate`

The delegate of a page view controller must adopt the `UIPageViewControllerDelegate` protocol. These methods allow the delegate to receive a notification when the device orientation changes and when the user navigates to a new page. For page-curl style transitions, the delegate can provide a different spine location in response to a change in the interface orientation.

`UIPickerViewAccessibilityDelegate`

The `UIPickerViewAccessibilityDelegate` protocol defines methods you can implement to provide accessibility information for individual components of a picker view.

`UIPickerViewDataSource`

The `UIPickerViewDataSource` protocol must be adopted by an object that mediates between a `UIPickerView` object and your application's data model for that picker view. The data source provides the picker view with the number of components, and the number of rows in each component, for displaying the picker view data. Both methods in this protocol are required.

`UIPickerViewDelegate`

The delegate of a `UIPickerView` object must adopt this protocol and implement at least some of its methods to provide the picker view with the data it needs to construct itself.

`UIPopoverBackgroundViewMethods`

The `UIPopoverBackgroundViewMethods` protocol defines methods that `UIPopoverBackgroundView`

`UIPopoverControllerDelegate`

The `UIPopoverControllerDelegate` protocol defines the methods you can implement for the delegate of a `UIPopoverController` object. Popover controllers notify their delegate whenever user interactions would cause the dismissal of the popover and, in some cases, give the user a chance to prevent that dismissal.

`UIPopoverPresentationControllerDelegate`

The methods of the `UIPopoverPresentationControllerDelegate` protocol let you customize the behavior of a popover-based presentation. A popover presentation controller notifies your delegate at appropriate points during the presentation process. You can use the delegate methods to customize this process and respond to changes dynamically.

UIPreviewActionItem

The `UIPreviewActionItem` protocol is adopted by the `UIPreviewAction` and `UIPreviewActionGroup` classes.

UIPreviewInteractionDelegate

UIPrinterPickerControllerDelegate

The `UIPrinterPickerControllerDelegate` protocol defines methods for managing the presentation and dismissal of a printer picker interface. You also use the methods of this protocol to influence the content displayed in the picker and to respond when the user selects a printer.

UIPrintInteractionControllerDelegate

The `UIPrintInteractionControllerDelegate` protocol is implemented by the delegate of the `UIPrintInteractionController` shared instance to perform a number of optional tasks.

UIResponderStandardEditActions

UIScrollViewAccessibilityDelegate

The `UIScrollViewAccessibilityDelegate` protocol defines methods you can implement to provide accessibility information for a scrollview.

UIScrollViewDelegate

The methods declared by the `UIScrollViewDelegate` protocol allow the adopting delegate to respond to messages from the `UIScrollView` class and thus respond to, and in some affect, operations such as scrolling, zooming, deceleration of scrolled content, and scrolling animations.

UISearchBarDelegate

The `UISearchBarDelegate` protocol defines the optional methods you implement to make a `UISearchBar` control functional. A `UISearchBar` object provides the user interface for a search field on a bar, but it's the application's responsibility to implement the actions when buttons are tapped. At a minimum, the delegate needs to perform the actual search when text is entered in the text field.

`UISearchControllerDelegate`

This protocol defines delegate methods for `UISearchController` objects.

`UISearchDisplayDelegate`

Important: `UISearchDisplayDelegate` is deprecated in iOS 8. (Note that `UISearchDisplayController` is also deprecated.) To manage the presentation of a search bar and display search results in iOS 8 and later, instead use `UISearchControllerDelegate`.

`UISearchResultsUpdating`

Use the `UISearchResultsUpdating` protocol to update search results based on information the user enters into the search bar.

`UISplitViewControllerDelegate`

mode and to the current interface orientation. When the split view interface collapses and expands, or when a new view controller is added to the interface, you can also use these methods to configure the child view controllers appropriately.

`UIStateRestoring`

The `UIStateRestoring` protocol lets you include any object in your state restoration archives. You can add state restoring objects to an archive directly or by referencing them from another object that is preserved, such as a view controller. The methods of the protocol let you save enough information about the object to find or recreate it during the next launch cycle.

`UITabBarControllerDelegate`

You use the `UITabBarControllerDelegate` protocol when you want to augment the behavior of a tab bar. In particular, you can use it to determine whether specific tabs should be selected, to perform actions after a tab is selected, or to perform actions before or after the user customizes the order of the tabs. After implementing these methods in your custom object, you should then assign that object to the `delegate` property of the corresponding `UITabBarController` object.

`UITabBarDelegate`

The `UITabBarDelegate` protocol defines optional methods for a delegate of a `UITabBar` object. The `UITabBar` class provides the ability for the user to reorder, remove, and add items to the tab bar; this process is referred to as customizing the tab bar. The tab bar delegate receives messages when customizing occurs.

`UITableViewDataSource`

The `UITableViewDataSource` protocol is adopted by an object that mediates the application’s data model for a `UITableView` object. The data source provides the table-view object with the information it needs to construct and modify a table view.

`UITableViewDataSourcePrefetching`

`UITableViewDelegate`

The delegate of a `UITableView` object must adopt the `UITableViewDelegate` protocol. Optional methods of the protocol allow the delegate to manage selections, configure section headings and footers, help to delete and reorder cells, and perform other actions.

`UITextDocumentProxy`

A text document proxy provides textual context to a custom keyboard (which is based on the `UIInputViewController` class) by way of the keyboard’s `textDocumentProxy` property.

`UITextFieldDelegate`

The `UITextFieldDelegate` protocol defines methods that you use to manage the editing and validation of text in a `UITextField` object. All of the methods of this protocol are optional.

`UITextInput`

The protocol you implement to interact with the text input system and enable features such as autocorrection and multistage text input in documents.

`UITextInputDelegate`

The text input delegate acts as an intermediary between a document and the text input system, conveying notifications of pending or transpired changes in text and selection in the document.

`UITextInputTokenizer`

An instance of a class that adopts the `UITextInputTokenizer` protocol is a tokenizer; a tokenizer allows the text input system to evaluate text units of different granularities. Granularities of text units are always evaluated with reference to a storage or reference direction.

`UITextInputTraits`

The `UITextInputTraits` protocol defines features associated with keyboard input to a text object.

related messages for `UITextView` objects. All of the methods in this protocol are optional. You can use them in situations where you might want to adjust the text being edited (such as in the case of a spell checker program) or modify the intended insertion point.

`UITimingCurveProvider`

Provides the timing information needed to perform animations.

`UIToolbarDelegate`

The `UIToolbarDelegate` protocol defines the interface that toolbar delegate objects implement to manage the toolbar behavior. This protocol declares no methods of its own but conforms to the UIBarPositioningDelegate protocol to support the positioning of a toolbar when it is moved to a window.

`UITraitEnvironment`

The iOS interface environment, which includes traits such as horizontal and vertical size class, display scale, and user interface idiom, is available to apps through the `UITraitEnvironment` protocol. The following interface classes adopt this protocol: `UIScreen`, `UIWindow`, `UIViewController`, `UIPresentationController`, and `UIView`.

`UIVideoEditorControllerDelegate`

The `UIVideoEditorControllerDelegate` protocol defines methods that your delegate object must implement to respond to the video editor. The methods of this protocol notify your delegate when the system has saved an edited movie or the user has cancelled editing to discard any changes. There is also a method for responding to errors encountered by the video editor.

`UIViewAnimating`

Defines methods for implementing custom animator objects.

`UIViewControllerAnimatedTransitioning`

animator object, which creates the animations for transitioning a view controller on or off screen in a fixed amount of time. The animations you create using this protocol must not be interactive. To create interactive transitions, you must combine your animator object with another object that controls the timing of your animations.

`UIViewControllerContextTransitioning`

The `UIViewControllerContextTransitioning` protocol's methods provide contextual information for transition animations between view controllers. Do not adopt this protocol in your own classes, nor should you directly create objects that adopt this protocol. During a transition, the animator objects involved in that transition receive a fully configured context object from UIKit. Custom animator objects —objects that adopt the `UIViewControllerAnimatedTransitioning` or `UIViewControllerInteractiveTransitioning` protocol—should simply retrieve the information they need from the provided object.

`UIViewControllerInteractiveTransitioning`

To enable an object (such as a navigation controller) to drive a view controller transition, configure a custom class to adopt the `UIViewControllerInteractiveTransitioning` protocol. An object that supports this protocol is called an *interactive transition delegate*.

`UIViewControllerPreviewing`

This protocol defines the interface for configuring a previewing view controller on devices that support 3D Touch.

`UIViewControllerPreviewingDelegate`

Implement the methods of this protocol to respond, with a preview view controller and a commit view controller, to the user pressing a view object on the screen of a device that supports 3D Touch.

`UIViewControllerRestoration`

The `UIViewControllerRestoration` protocol should be adopted by classes that act as "restoration classes" for view controllers during the state restoration process. The method in this protocol should be

`UIViewControllerTransitionCoordinator`

An object that adopts the `UIViewControllerTransitionCoordinator` protocol provides support for animations associated with a view controller transition. Typically, you do not adopt this protocol in your own classes. When you present or dismiss a view controller, UIKit creates a transition coordinator object automatically and assigns it to the view controller's `transitionCoordinator` property. That transition coordinator object is ephemeral and lasts for the duration of the transition animation.

`UIViewControllerTransitionCoordinatorContext`

An object that conforms to the `UIViewControllerTransitionCoordinatorContext` protocol provides information about an in-progress view controller transition. Do not adopt this protocol in your own classes. UIKit creates an object that adopts this protocol and makes it available to your code when you animate changes using a transition coordinator object.

`UIViewControllerTransitioningDelegate`

An object that implements the `UIViewControllerTransitioningDelegate` protocol vends the objects used to manage a fixed-length or interactive transition between view controllers. When you want to present a view controller using a custom modal presentation type, set its `modalPresentationStyle` property to `custom` and assign an object that conforms to this protocol to its `transitioningDelegate` property. When you present that view controller, UIKit queries your transitioning delegate for the objects to use when animating the view controller into position.

`UIViewImplicitlyAnimating`

Provides methods for modifying an animation while it is running.

`UIWebViewDelegate`

The `UIWebViewDelegate` protocol defines methods that a delegate of a `UIWebView` object can optionally implement to intervene when web content is loaded.

API Reference

UIKit | Show API Changes

UIKit Constants

UIKit Data Types

UIKit Structures

UIKit Enumerations

UIKit Operators

Structures

`NSStringDrawingOptions`

The following constants are provided as rendering options for a string when it is drawn.

`UIContentSizeCategory`

`UIFontDescriptorSymbolicTraits`

`UIFontDescriptorSymbolicTraits` symbolically describes stylistic aspects of a font. The lower 16 bits represent the typeface, and the upper 16 bits describe appearance of the font.

`UIRectCorner`

The corners of a rectangle.

Extended Types

`CIColor`

The component values defining a color in a specific color space

`CIImage`

A representation of an image to be processed or produced by Core Image filters.

`IndexPath`

`IndexPath` represents the path to a specific node in a tree of nested array collections.

`NSAttributedString`

An `NSAttributedString` object manages character strings and associated sets of attributes (for example, font and kerning) that apply to individual characters or ranges of characters in the string. An association of characters and their attributes is called an attributed string. The cluster’s two public classes, `NSAttributedString` and `NSMutableAttributedString`, declare the programmatic interface for read-only attributed strings and modifiable attributed strings, respectively.

`Bundle`

An `NSBundle` object helps you access the code and resources in a bundle directory on disk. Apple uses bundles to represent apps, frameworks, plug-ins, and many other specific types of content. Bundles organize their contained resources into well-defined subdirectories, and bundle structures vary depending on the platform and the type of the bundle. By using a bundle object, you do not have to know the structure of a bundle to access its resources. The bundle object provides a single interface for locating items, taking into account the bundle structure, user preferences, available localizations, and other relevant factors.

`NSCoder`

The `NSCoder` abstract class declares the interface used by concrete subclasses to transfer objects and other values between memory and some other format. This capability provides the basis for archiving (where objects and data items are stored on disk) and distribution (where objects and data items are copied between different processes or threads). The concrete subclasses provided by Foundation for these purposes are `NSArchiver`, `NSUnarchiver`, `NSKeyedArchiver`, `NSKeyedUnarchiver`, and `NSPortCoder`. Concrete subclasses of `NSCoder` are referred to in general as coder classes, and instances of these classes as coder objects (or simply coders). A coder object that can only encode values is referred

`NSExceptionName`

`NSIndexPath`

The `NSIndexPath` class represents the path to a specific node in a tree of nested array collections. This path is known as an **index path**.

`NSMutableAttributedString`

The `NSMutableAttributedString` class declares additional methods for mutating the content of an attributed string. You can add and remove characters (raw strings) and attributes separately or together as attributed strings. See the class description for `NSAttributedString` for more information about attributed strings.

`NSNotification.Name`

The type used for the name of a notification.

`NSObject`

`NSObject` is the root class of most Objective-C class hierarchies. Through `NSObject`, objects inherit a basic interface to the runtime system and the ability to behave as Objective-C objects.

`RunLoopMode`

`NSString`

The `NSString` class and its mutable subclass, `NSMutableString`, provide an extensive set of APIs for working with strings, including methods for comparing, searching, and modifying strings. `NSString` objects are used throughout Foundation and other Cocoa frameworks, serving as the basis for all textual and linguistic functionality on the platform.

`NSValue`

this class to work with such data types in collections (such as `NSArray` and `NSSet`), Key-value coding, and other APIs that require Objective-C objects. `NSValue` objects are always immutable.

`URLResourceValues`

URLs to file system resources support the properties defined below. Note that not all property values will exist for all file system URLs. For example, if a file is located on a volume that does not support creation dates, it is valid to request the creation date property, but the returned value will be nil, and no error will be generated.