UIKit User Interface Catalog

Developer

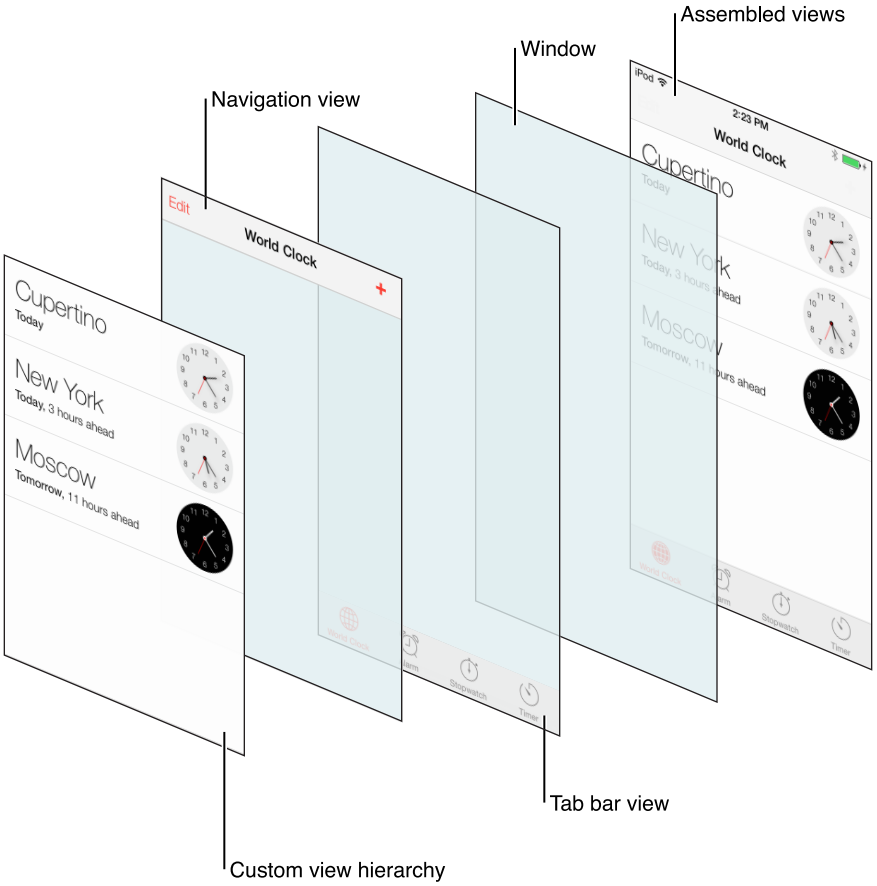
Contents

Views

About Views

Important: This is a preliminary document for an API or technology in development. Although this document has been reviewed for technical accuracy, it is not final. This Apple confidential information is for use only by registered members of the applicable Apple Developer program. Apple is supplying this confidential information to help you plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future seeds of the API or technology.

You can think of views as building blocks that you use to construct your user interface. Rather than use one view to present all of your content, you often use several views to build a view hierarchy. These views range from simple buttons and text labels to more complex views such as table views, picker views, and scroll views. Each view in the hierarchy presents a particular portion of your user interface and is generally optimized for a specific type of content.



All views in UIKit are subclasses of the base class `UIView`. For example, UIKit has views specifically for presenting images, text, and other types of content. In places where the predefined views do not provide what you need, you can also define custom views and manage the drawing and event handling yourself.

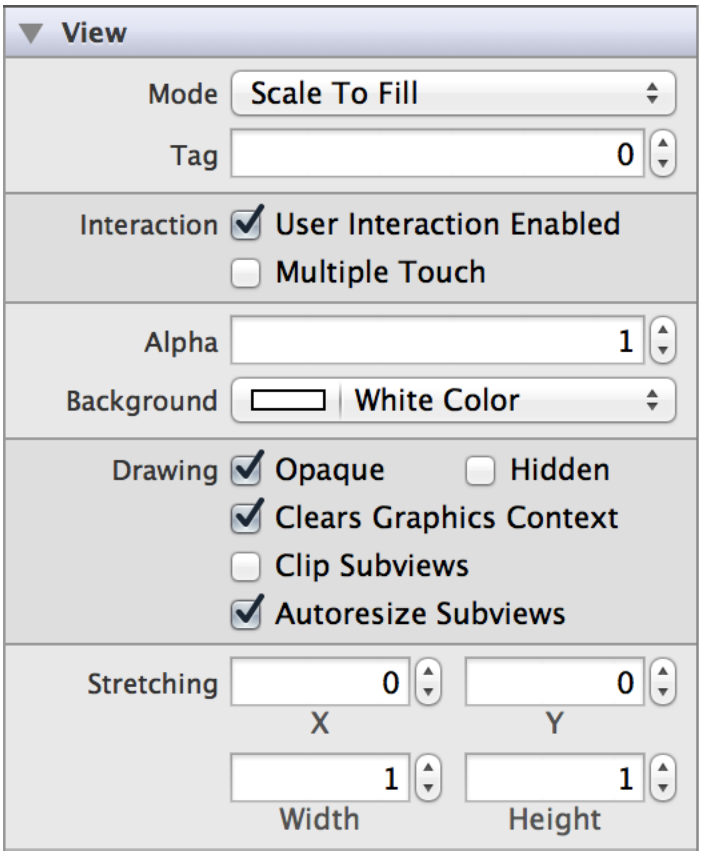
Views allow the user to:

- Experience app content
- Navigate within an app

Implementation: Views are implemented in the `UIView` class and discussed in *UIView Class Reference*.

Configuring Views

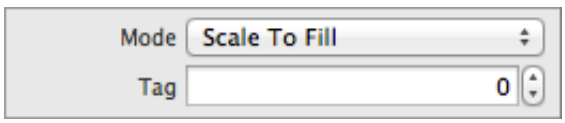
Generally, the easiest way to configure views—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



Setting View Content

Use the Mode (`contentMode`) field to specify how a view lays out its content when its bounds change. This property is often used to implement resizable controls. Instead of redrawing the contents of the view every time its bounds change, you can use this property to specify that you want to scale the contents or pin them to a particular spot on the view.

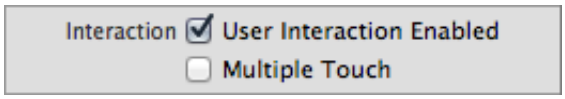
The Tag (`tag`) field serves as an integer that you can use to identify view objects in your app.



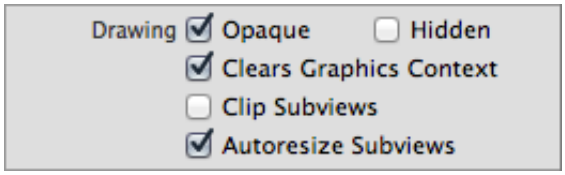
Specifying View Behavior

By default, the User Interaction Enabled (`userInteractionEnabled`) checkbox is selected, which means that user events—such as touch and keyboard—are delivered to the view normally. When the checkbox is unselected, events intended for the view are ignored and removed from the event queue.

The Multiple Touch (`multipleTouchEnabled`) checkbox is unselected by default, which means that the view receives only the first touch event in a multi-touch sequence. When selected, the view receives all touches associated with a multitouch sequence.



Views have a number of properties related to drawing behavior:



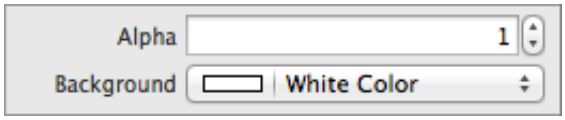
- The Opaque (`opaque`) checkbox tells the drawing system how it should treat the view. If selected, the drawing system treats the view as fully opaque, which allows the drawing system to optimize some drawing operations and improve performance. If unselected, the drawing system composites the view normally with other content. You should always disable this checkbox if your view is fully or partially transparent.
- If the Hidden (`hidden`) checkbox is selected, the view disappears from its window and does not receive input events.
- When the Clears Graphics Context (`clearsContextBeforeDrawing`) checkbox is selected, the drawing buffer is automatically cleared to transparent black before the view is drawn. This behavior ensures that there are no visual artifacts left over when the view's contents are redrawn.
- Selecting the Clip Subviews (`clipsToBounds`) checkbox causes subviews to be clipped to the bounds of the view. If unselected, subviews whose frames extend beyond the visible bounds of the view are not clipped.
- When the Autoresize Subviews (`autoresizesSubviews`) checkbox is selected, the view adjusts the size of its subviews when its bounds change.

Customizing View Appearance

Setting View Background Color and Alpha

Adjusting the Alpha (`alpha`) field changes the transparency of the view as a whole. This value can range from `0.0` (transparent) to `1.0` (opaque). Setting a view's alpha value does not have an effect on embedded subviews.

Use the Background (`backgroundColor`) field to select a color to fill in the entire frame of the view. The background color appears underneath all other content in the view.



Using Appearance Proxies

You can use an appearance proxy to set particular appearance properties for all instances of a view in your application. For example, if you want all sliders in your app to have a particular minimum track tint color, you can specify this with a single message to the slider's appearance proxy.

There are two ways to customize appearance for objects: for all instances and for instances contained within an instance of a container class.

- To customize the appearance of all instances of a class, use `appearance` to get the appearance proxy for the class.

```
[[UISlider appearance] setMinimumTrackTintColor:[UIColor greenColor]];
```

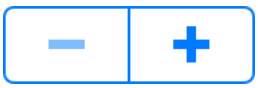
- To customize the appearances for instances of a class when contained within an instance of a container class, or instances in a hierarchy, you use `appearanceWhenContainedIn:` to get the appearance proxy for the class.

```
[[UISlider appearanceWhenContainedIn:[UIView class], nil]
            setMinimumTrackTintColor:[UIColor greenColor]];
```

Note: You cannot use the appearance proxy with the `tintColor` property on `UIView`. For more information on using `tintColor`, see "Adjusting View Tint Color" (page 19).

Adjusting View Tint Color

Views have a `tintColor` property that specifies the color of key elements within the view. Each subclass of `UIView` defines its own appearance and behavior for `tintColor`. For example, this property determines the color of the outline, divider, and glyphs on a stepper:



The `tintColor` property is a quick and simple way to skin your app with a custom color. Setting a tint color for a view automatically sets that tint color for all of its subviews. However, you can manually override this property for any of those subviews and its decendants. In other words, each view inherits its superview's tint color if its own tint color is `nil`. If the highest level view in the view heirarchy has a `nil` valur for tint color, it defaults to the system blue color.

Using Template Images

In iOS 7, you can choose to treat any of the images in your app as a template—or stencil—image. When you elect to treat an image as a template, the system ignores the image's color information and creates an image stencil based on the alpha values in the image (parts of the image with an alpha value of less than `1.0` get treated as completely transparent). This stencil can then be recolored using `tintColor` to match the rest of your user interface.

By default, an image (`UIImage`) is created with `UIImageRenderingModeAutomatic`. If you have `UIImageRenderingModeAutomatic` set on your image, it will be treated as template or original based on its context. Certain UIKit elements—including navigation bars, tab bars, toolbars, segmented controls—automatically treat their foreground images as templates, although their background images are treated as original. Other elements—such as image views and web views—treat their images as originals. If you want your image to always be treated as a template regardless of context, set `UIImageRenderingModeAlwaysTemplate`; if you want your image to always be treated as original, set `UIImageRenderingModeAlwaysOriginal`.

To specify the rendering mode of an image, first create a standard image, and then call the `imageWithRenderingMode:` method on that image.

```
UIImage *myImage = [UIImage imageNamed:@"myImageFile.png"];
myImage = [myImage imageWithRenderingMode:UIImageRenderingModeAlwaysTemplate];
```

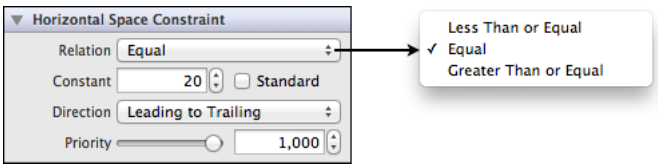
Using Auto Layout with Views

The Auto Layout system allows you to define layout constraints for user interface elements, such as views and controls. Constraints represent relationships between user interface elements. You can create Auto Layout constraints by selecting the appropriate element or group of elements and selecting an option from the menu in the bottom right corner of Xcode's Interface Builder.

Auto layout contains two menus of constraints: pin and align. The Pin menu allows you to specify constraints that define some relationship based on a particular value or range of values. Some apply to the control itself (width) while others define relationships between elements (horizontal spacing). The following tables describes what each group of constraints in the Auto Layout menu accomplishes:

Constraint Name	Purpose
Width Height	Sets the width or height of a single element.
Horizontal Spacing Vertical Spacing	Sets the horizontal or vertical spacing between exactly two elements.
Leading Space to Superview Trailing Space to Superview Top Space to Superview Bottom Space to Superview	Sets the spacing from one or more elements to the leading, trailing, top, or bottom of their container view. Leading and trailing are the same as left and right in English, but the UI flips when localized in a right-to-left environment.
Widths Equally Heights Equally	Sets the widths or heights of two or more elements equal to each other.
Left Edges Right Edges Top Edges Bottom Edges	Aligns the left, right, top, or bottom edges of two or more elements.
Horizontal Centers Vertical Centers Baselines	Aligns two or more elements according to their horizontal centers, vertical centers, or bottom baselines. Note that baselines are different from bottom edges. These values may not be defined for certain elements.
Horizontal Center in Container Vertical Center in Container	Aligns the horizontal or vertical centers of one or more elements with the horizontal or vertical center of their container view.

The “Constant” value specified for any Pin constraints (besides Widths/Heights Equally) can be part of a “Relation.” That is, you can specify whether you want the value of that constraint to be equal to, less than or equal to, or greater than or equal to the value.



For more information, see *Cocoa Auto Layout Guide*.

Making Views Accessible

To enhance the accessibility information for an item, select the object on the storyboard and open the Accessibility section of the Identity inspector.

For more information, see *Accessibility Programming Guide for iOS*.

Debugging Views

When debugging issues with views, watch for this common pitfall:

Setting conflicting opacity settings. You should not set the `opaque` property to `YES` if your view has an alpha value of less than `1.0`.

Action Sheets

Action sheets display a set of buttons representing several alternative choices to complete a task initiated by the user. For example, when the user taps the Share button in an app's toolbar, an action sheet appears offering a list of choices, such as Email, Print, and so on.

Action sheets allow the user to:

- Quickly select from a list of actions
- Confirm or cancel an action

Implementation: Action sheets are implemented in the `UIActionSheet` class and discussed in the *UIActionSheet Class Reference*.

Configuring Action Sheets

Action sheets are created, initialized, and configured in code, typically residing in a view controller implementation file. You cannot create or manipulate action sheets in Interface Builder. Rather, an action sheet appears over an existing view, interrupting its presentation, and it requires the user to dismiss it by tapping one of its buttons (or tapping outside the action sheet on iPad). The chosen action is handled by the action sheet's delegate.

Setting Action Sheet Content Programmatically

When you create an action sheet object from the `UIActionSheet` class, you can initialize its most important properties with one method, `initWithTitle:delegate:cancelButtonTitle:destructiveButtonTitle:otherButtonTitles:`. Depending on your app's needs, that single message may be enough to configure a fully functional action sheet object, as shown in the following code. Once you have created the action sheet object, send it a `show...` message, such as `showInView:` to display the action sheet.

```
UIActionSheet *actionSheet = [[UIActionSheet alloc] initWithTitle:nil
                                                         delegate:self
                                                cancelButtonTitle:@"Cancel"
                                           destructiveButtonTitle:@"Delete Note"
```

```
                                                otherButtonTitles:nil];
```

Although the first parameter of the `initWithTitle:delegate:cancelButtonTitle:destructiveButtonTitle:otherButtonTitles:` method enables you to provide a title for an action sheet, iOS human interface guidelines recommend that you do not use a title.

As described in iOS human interface guidelines, you should include a Cancel button with action sheets displayed on iPhone and with those displayed on iPad over an open popover. Otherwise on iPad, action sheets are displayed within a popover, and the user can cancel the action sheet by tapping outside the popover, in which case you do not need to include a Cancel button.

To create a Cancel button, pass a non-`nil` value for the `cancelButtonTitle:` parameter of the `initWithTitle:delegate:cancelButtonTitle:destructiveButtonTitle:otherButtonTitles:` method. A Cancel button created in this way is positioned at the bottom of the action sheet.

When your action sheet presents a potentially destructive choice, you should include a destructive button by passing a non-`nil` value for the `destructiveButtonTitle:` parameter of the `initWithTitle:delegate:cancelButtonTitle:destructiveButtonTitle:otherButtonTitles:` method. A destructive button created in this way is automatically colored red and positioned at the top of the action sheet.

Specifying Action Sheet Behavior Programmatically

You can choose to present an action sheet so that it originates from a toolbar, tab bar, button bar item, from a view, or from a rectangle within a view. On iPhone, because the action sheet slides up from the bottom of a view and covers the width of the screen, most apps use `showInView:`. On iPad, however, action sheets appear within a popover whose arrow points to the control the user tapped to invoke the choices presented by the action sheet. So, `showFromRect:inView:animated:` and `showFromBarButtonItem:animated:` are most useful on iPad.

To handle the choices presented by your action sheet, you must designate a delegate to handle the button's action, and the delegate must conform to the `UIActionSheetDelegate` protocol. You designate the delegate with the `delegate` parameter when you initialize the action sheet object. The delegate must implement the `actionSheet:clickedButtonAtIndex:` message to respond when the button is tapped. For example, the following code shows an implementation that simply logs the title of the button that was tapped.

```
- (void)actionSheet:(UIActionSheet *)actionSheet
clickedButtonAtIndex:(NSInteger)buttonIndex

{
```

```
    NSLog(@"The %@ button was tapped.", [actionSheet
buttonTitleAtIndex:buttonIndex]);
}
```

Using Auto Layout with Action Sheets

The layout of action sheets is handled for you. You cannot create Auto Layout constraints between an action sheet and another user interface element.

For general information about using Auto Layout with iOS views, see "Using Auto Layout with Views" (page 20).

Making Action Sheets Accessible

Action sheets are accessible by default.

Accessibility for action sheets primarily concerns button titles. If VoiceOver is activated, it speaks the word "alert" when an action sheet is shown, then speaks its title if set (although iOS human interface guidelines recommend against titling action sheets). As the user taps a button in the action sheet, VoiceOver speaks its title and the word "button."

For general information about making iOS views accessible, see "Making Views Accessible" (page 21).

Internationalizing Action Sheets

To internationalize an action sheet, you must provide localized translations of the button titles. Button size may change depending on the language and locale.

For more information, see *Internationalization Programming Topics*.

Debugging Action Sheets

When debugging issues with action sheets, watch for this common pitfall:

Not testing localizations. Be sure to test the action sheets in your app with the localizations you intend to ship. In particular, button titles can truncate if they are longer in localizations other than the one in which you designed your user interface. Following the HI guideline to give buttons short, logical titles helps to ameliorate this potential problem, but localization testing is also required.

Elements Similar to an Action Sheet

The following elements provide similar functionality to an action sheet:

- **Alert View.** Use an alert view to convey important information about an app or the device, interrupting the user and requiring them to stop what they're doing to choose an action or dismiss the alert. For more information, see "Alert Views" (page 30).
- **Modal View.** Use a modal view (that is, the view controller uses a modal presentation style) when users initiate a self-contained subtask in the context of their workflow or another task. For more information, see *UIViewController Class Reference*.

Activity Indicators

An activity indicator is a spinning wheel that indicates a task is in the midst of being processed. If an action takes a noticeable and indeterminate amount of time to process—such as a CPU-intensive task or connecting to a network—you should display an activity indicator to give assurance to the user that your app is not stalled or frozen.

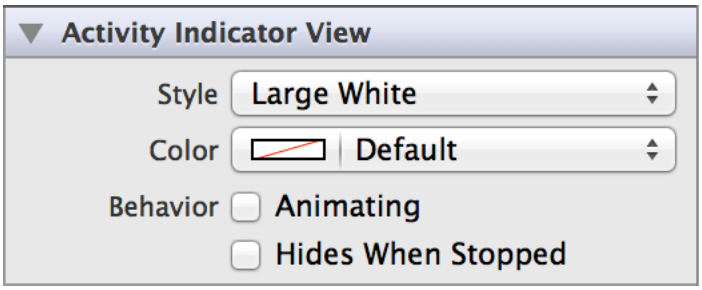
Activity indicators allow the user to:

- Receive feedback that the system is processing information

Implementation: Activity indicators are implemented in the `UIActivityIndicatorView` class and discussed in the *UIActivityIndicatorView Class Reference*.

Configuring Activity Indicators

Generally, the easiest way to configure views—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



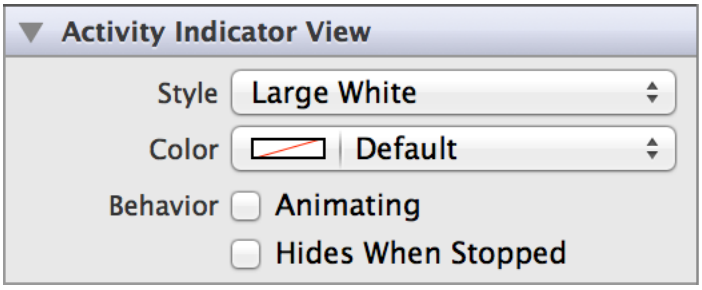
Setting Activity Indicator Values

An activity indicator is indeterminate, and has no starting or ending values associated with it.

Specifying Activity Indicator Behavior

The Animating (`isAnimating`) box is unchecked by default; checking it causes the activity indicator to start animating. This is the equivalent of calling the `startAnimating` method.

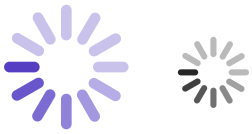
Select the Hides When Stopped (`hidesWhenStopped`) field in the Attributes Inspector for your activity indicator to disappear when the animation ends. When you call the `startAnimating` and `stopAnimating` methods, the activity indicator automatically shows and hides onscreen. This way, you won't have to worry about displaying a stationary activity indicator.



Customizing Activity Indicator Appearance

You can customize the appearance of an activity indicator by setting the properties depicted below.

`activityIndicatorViewStyle`
`currentPageIndicatorTintColor`



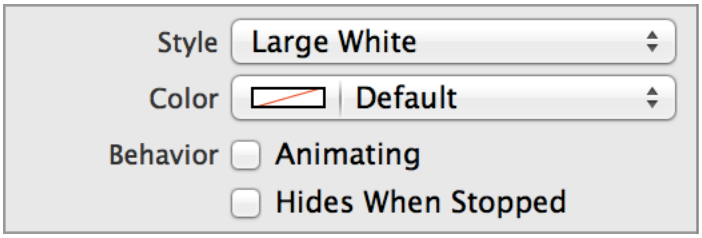
(Shown at 2x)

To customize the appearance of all activity indicators in your app, use the appearance proxy (for example, `[UIActivityIndicatorView appearance]`). For more information about appearance proxies, see "Using Appearance Proxies" (page 18).

Setting Activity Indicator Style

The Style (`activityIndicatorViewStyle`) field represents the predefined style of the activity indicator. Use the style to specify one of two default colors: white or gray. You can also select a larger size for your indicator using the "Large White" style option.

The Color (`color`) field allows you to specify a custom color for your indicator. This property takes precedence over any color set using the Style field. However, if your style is set to Large White, your activity indicator appears a larger size. Make sure your indicator is set to a different style option if you want to use the small indicator.



Using Auto Layout with Activity Indicators

You can create Auto Layout constraints between an activity indicator and other UI elements.

Typically, activity indicators appear before a label or centered within a view. To align with a label, constrain Bottom Edges and Horizontal Space to the label with the standard value. To center within a view, add the Horizontal Center in Container and Vertical Center in Container constraints.

For general information about using Auto Layout with iOS views, see "Using Auto Layout with Views" (page 20).

Making Activity Indicators Accessible

Activity indicators are accessible by default. The default accessibility trait for an activity indicator is User Interaction Enabled.

If you have a label next to your activity indicator that describes the processing task in more detail, you might want to disable its accessibility with the `isAccessibilityElement` property so VoiceOver reads the label instead. Otherwise, VoiceOver reads "In progress" while the activity indicator is animating, and "Progress halted" while it is not.

VoiceOver will read only elements that are visible onscreen. If you enable the `hidesWhenStopped` property, VoiceOver might abruptly stop speaking when the animation finishes.

For general information about making iOS views accessible, see "Making Views Accessible" (page 21).

Internationalizing Activity Indicators

Activity indicators have no special properties related to internationalization. However, if you use an activity indicator with a label, make sure you provide localized strings for the label.

For more information, see *Internationalization Programming Topics* .

Debugging Activity Indicators

When debugging issues with activity indicators, watch for this common pitfall:

Specifying conflicting appearance settings. The `color` property takes precedence over any color set using the `activityIndicatorViewStyle` property. However, if your style is set to Large White, your activity indicator appears a larger size with whatever custom color you set. Make sure your indicator is set to a different style option if you want to use the small indicator.

Elements Similar to an Activity Indicator

The following element provides similar functionality to an activity indicator:

Progress View. A class that represents a progress bar. Use this class instead of an activity indicator when your task takes a determinate amount of time. For more information, see "Progress Views" (page 65).

Alert Views

Alert views display a concise and informative alert message to the user. Alert views convey important information about an app or the device, interrupting the user and requiring them to stop what they're doing to choose an action or dismiss the alert. For example, iOS uses alerts to warn the user that battery power is running low, so they can connect a power adapter before their workflow is interrupted. An alert view appears on top of app content, and must be manually dismissed by the user before he can resume interaction with the app.

Alert views allow the user to:

- Be immediately informed of critical information
- Make a decision about a course of action

Implementation: Alert views are implemented in the `UIAlertView` class and discussed in the *UIAlertView Class Reference* .

Configuring Alert Views

Alert views are created, initialized, and configured in code, typically residing in a view controller implementation file. You cannot create or manipulate alert views in Interface Builder. Rather, an alert view floats over an existing view to interrupt its presentation, and it requires the user to dismiss it. If the alert view contains a custom button enabling the user to choose an alternative action, rather than simply dismissing the alert, that action is handled by the alert view's delegate.

Setting Alert View Content Programmatically

When setting alert view content, you can control the number of buttons, their titles, displayed text, and inclusion of one or two text fields, one of which can be a secure text-input field.

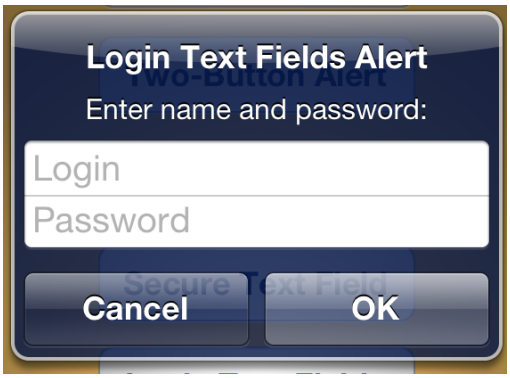
When you create an alert view object from the `UIAlertView` class, you can initialize its most important properties with one method, `initWithTitle:message:delegate:cancelButtonTitle:otherButtonTitles:`. Depending on your app's needs, that single message may be enough to configure a fully functional alert object, as shown in the following code. After you have created the alert object, send it a `show` message to display the alert.

```
UIAlertView *theAlert = [[UIAlertView alloc] initWithTitle:@"Title"
                                                   message:@"This is the message."
                                                  delegate:self
                                         cancelButtonTitle:@"OK"
                                         otherButtonTitles:nil];
[theAlert show];
```

Every alert has a Cancel button so that the user can dismiss the alert. You can add additional, custom buttons to enable the user to perform some other action related to the alert, such as rectifying the problem the alert warned about. Although you can add multiple custom buttons to an alert, *iOS Human Interface Guidelines* recommend that you limit alerts to two buttons, and consider using an action sheet instead if you need more.

To add a titled custom button to an alert, send it an `addButtonWithTitle:` message. Alternatively, you can pass the custom button title, followed by a comma and `nil` terminator, with the `otherButtonTitles:` parameter when you initialize the alert view object.

Optionally, an alert can contain one or two text fields, one of which can be a secure text-input field. You add text fields to an alert after it is created by setting its `alertViewStyle` property to one of the styles specified by the `UIAlertViewStyle` constants. The alert view styles can specify no text field (the default style), one plain text field, one secure text field (which displays a bullet character as each character is typed), or two text fields (one plain and one secure) to accommodate a login identifier and password.



Specifying Alert View Behavior Programmatically

If your alert has a custom button, you must designate a delegate to handle the button's action, and the delegate must conform to the `UIAlertViewDelegate` protocol. You designate the delegate with the `delegate` parameter when you initialize the alert view object. The delegate must implement the `alertView:clickedButtonAtIndex:` message to respond when the custom button is tapped; otherwise, your custom buttons do nothing. For example, the following code shows an implementation that simply logs the title of the button that was tapped:

```
- (void)alertView:(UIAlertView *)theAlert clickedButtonAtIndex:(NSInteger)buttonIndex
{
    NSLog(@"The %@ button was tapped.", [theAlert buttonTitleAtIndex:buttonIndex]);
}
```

In the delegate method `alertView:clickedButtonAtIndex:`, depending on which button the user tapped, you can retrieve the values of text fields in your alert view with the `textFieldAtIndex:` method.

```
if (theAlert.alertViewStyle == UIAlertViewStyleLoginAndPasswordInput) {
    NSLog(@"The login field says %@, and the password is %@.",
          [theAlert textFieldAtIndex:0].text, [theAlert textFieldAtIndex:1].text);
}
```

The alert view is automatically dismissed after the `alertView:clickedButtonAtIndex:` delegate method is invoked. Optionally, you can implement the `alertViewCancel:` method to take the appropriate action when the system cancels your alert view. An alert view can be canceled at any time by the system—for example, when the user taps the Home button. If the delegate does not implement the `alertViewCancel:` method, the default behavior is to simulate the user clicking the cancel button and closing the view.

Customizing Alert View Appearance

You cannot customize the appearance of alert views.

Using Auto Layout with Alert Views

The layout of alert views is handled for you. You cannot create Auto Layout constraints between an alert view and another user interface element.

For general information about using Auto Layout with iOS views, see "Using Auto Layout with Views" (page 20).

Making Alert Views Accessible

Alert views are accessible by default.

Accessibility for alert views pertains to the alert title, alert message, and button titles. If VoiceOver is activated, it speaks the word "alert" when an alert is shown, then speaks its title followed by its message if set. As the user taps a button, VoiceOver speaks its title and the word "button." As the user taps a text field, VoiceOver speaks its value and "text field" or "secure text field."

For general information about making iOS views accessible, see "Making Views Accessible" (page 21).

Internationalizing Alert Views

To internationalize an alert view, you must provide localized translations of the alert title, message, and button titles. Screen metrics and layout may change depending on the language and locale.

For more information, see *Internationalization Programming Topics*.

Debugging Alert Views

When debugging issues with sliders, watch for this common pitfall:

Not testing localizations. Be sure to test the alert views in your app with the localizations you intend to ship. In particular, button titles can truncate if they are longer in localizations other than the one in which you designed your user interface. Following the HI guideline to give buttons short, logical titles helps to ameliorate this potential problem, but localization testing is also required.

Elements Similar to an Alert View

The following elements provide similar functionality to an alert view:

- **Action Sheet.** Present an action sheet when users tap a button in a toolbar, giving them choices related to the action they've initiated. For more information, see "Action Sheets" (page 22).
- **Modal View.** Present a modal view (that is, the view controller uses a modal presentation style) when users initiate a subtask in the context of their workflow or another task. For more information, see *UIViewController Class Reference*.

Collection Views

A collection view displays an ordered collection of data items using standard or custom layouts. Similar to a table view, a collection view gets data from your custom data source objects and displays it using a combination of cell, layout, and supplementary views. A collection view can display items in a grid or in a custom layout that you design. Regardless of the layout style you choose, a collection view is best suited to display nonhierarchical, ordered data items.

Collection views allow the user to:

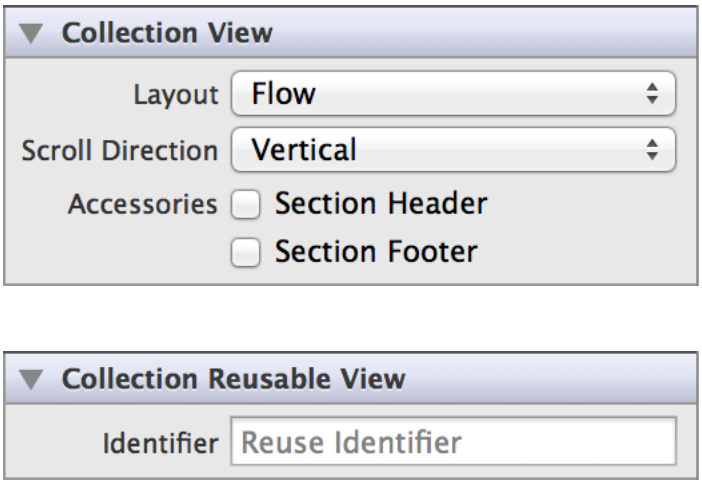
- View a catalog of variably sized items, optionally sorted into multiple sections
- Add to, rearrange, and edit a collection of items
- Choose from a frequently changing display of items

Implementation:

- Collection views are implemented in the `UICollectionView` class and discussed in the *UICollectionView Class Reference*.
- Collection view cells are implemented in the `UICollectionViewCell` class and discussed in the *UICollectionViewCell Class Reference*.
- Collection reusable views are implemented in the `UICollectionReusableView` class and discussed in the *UICollectionReusableView Class Reference*.

Configuring Collection Views

Generally, the easiest way to configure views—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



Setting Collection View Content

Cells present the main content of your collection view. The job of a cell is to present the content for a single item from your data source object. Each cell must be an instance of the `UICollectionViewCell` class, which you may subclass as needed to present your content. Cell objects provide inherent support for managing their own selection and highlight state, although some custom code must be written to actually apply a highlight to a cell. A `UICollectionViewCell` object is a specific type of reusable view that you use for your main data items.

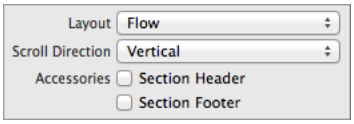
To manage the visual presentation of data, a collection view works with many related classes, such as `UICollectionViewController`, `UICollectionViewDataSource`, `UICollectionViewDelegate`, `UICollectionReusableView`, `UICollectionViewCell`, `UICollectionViewLayout`, and `UICollectionViewLayoutAttributes`.

Collection views enforce a strict separation between the data being presented and the visual elements used for presentation. Your app is solely responsible for managing the data via your custom data source objects. (To learn how to create these objects, see "Designing Your Data Objects" in *Collection View Programming Guide*

for iOS.) Your app also provides the view objects used to present that data. The collection view takes your views and—with the help of a layout object, which specifies placement and other visual attributes—does all the work of displaying them onscreen.

To display content onscreen in an efficient way, a collection view uses the following reusable view objects:

- **Cell.** Represents one data item.
- **Supplementary view.** Represents information related to the data items, such as a section header or footer.
- **Decoration view.** Represents purely decorative content that's not part of your data, such as a background image.



Because a collection view works with these and other objects to determine the visual presentation of your data, configuring a collection view in Interface Builder means that you need to configure some objects separately.

- **Items.** The number of different types of data for which you define distinct cell objects. If your app works with only one type of data item—regardless of the total number of data items you display—set this value to 1.
- **Accessories.** The existence of a header or footer view for each section (this property isn't available for custom layouts). Select Section Header or Section Footer as appropriate.

In Collection Reusable View Attributes inspector—which governs supplementary views, decoration views, and cells—you can set the Identifier (identifier) field. Enter the ID you use in your code to identify the reusable cell, decoration, or supplementary view object.

Specifying Collection View Behavior

There are several behaviors you can support in your collection view. For example, you might want to allow users to:

- Select one or more items
- Insert, delete, and reorder items or sections
- Edit an item

By default, a collection view detects when the user taps a specific cell and it updates the cell's selected or highlighted properties as appropriate. You can write code that configures a collection view to support multiple-item selection or that draws the selected or highlighted states yourself. To learn how to support multiple selection or custom selection states, see "Managing the Visual State for Selections and Highlights" in *Collection View Programming Guide for iOS*.

To support insertion, deletion, or reordering of cells in a collection view, you make changes to your data source and then tell the collection view to redisplay the content. By default, a collection view animates the insertion, deletion, or movement of a single item; if you want to animate these changes for multiple items at once, you use code blocks to batch the update. To learn how animate multiple changes to a collection view, see "Inserting, Deleting, and Moving Sections and Items" in *Collection View Programming Guide for iOS*. To let users move an item or items by dragging, you also need to incorporate a custom gesture recognizer. (To learn how to do this, see "Manipulating Cells and Views" in *Collection View Programming Guide for iOS*.)

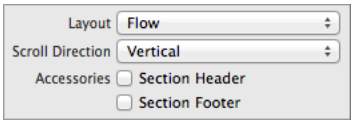
If you support editing for an item, the collection view automatically displays the Edit menu when it detects a long-press gesture on a specific cell. To learn how to support editing in a collection view, see "Showing the Edit Menu for a Cell" in *Collection View Programming Guide for iOS*.

When configuring cells and supplementary views in a storyboard, you do so by dragging the item onto your collection view and configuring it there. This creates a relationship between the collection view and the corresponding cell or view. For cells, drag a collection view cell from the object library and drop it onto your collection view. Set the custom class and the collection reusable view identifier of your cell to appropriate values.

Whether the user is selecting or deselecting a cell, the cell's selected state is always the last thing to change. Taps in a cell always result in changes to the cell's highlighted state first. Only after the tap sequence ends and any highlights applied during that sequence are removed, does the selected state of the cell change. When designing your cells, you should make sure that the visual appearance of your highlights and selected state do not conflict in unintended ways.

When the user performs a long-tap gesture on a cell, the collection view attempts to display an Edit menu for that cell. The Edit menu can be used to cut, copy, and paste cells in the collection view.

If you're working with the `UICollectionViewFlowLayout` class, you can use the Attributes inspector to set the "Scroll Direction" (`scrollDirection`) field to `Horizontal` or `Vertical`. Note that this property isn't available for custom layouts.

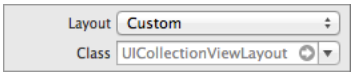


Customizing Collection View Appearance

Specifying Collection View Layout

A collection view relies on a layout object to define the layout of its cells, supplementary views, and decoration views.

The Layout field determines the layout of the cells. The default value is Flow, which refers to the layout defined by the `UICollectionViewFlowLayout` class. If you provide a custom layout class, choose Custom instead.



To learn more about creating a custom layout class, see *Collection View Programming Guide for iOS*.

Setting Collection View Background

To use a custom background for a collection view, you can specify a view that's positioned underneath all of the other content and sized automatically to fill the entire bounds of the collection view. You can set this value using the `backgroundView` property. Because this background view doesn't scroll with the collection view's content, it's not an appropriate way to display a decorative background such as the appearance of wooden shelves.

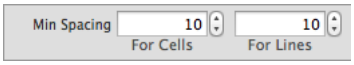
Setting Collection View Cell Background

To use a custom background for a single collection view cell, you can specify a view that's positioned behind the cell's content view and that fills the cell's bounds. You can set this value using the `backgroundView` property.

You can also specify a custom selected background by providing a view that's displayed above the cell's background view—and behind the content view—when the user selects the cell. Set this value using the `selectedBackgroundView` property.

Adjusting Collection View Spacing

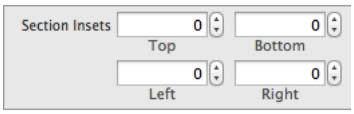
In the Collection View Flow Layout Size Inspector, you can set size values (in points) for the layout object to use when laying out cells and supplementary views.



For spacing between cells you can set the following Min Spacing values:

- **For Cells.** The minimum space to maintain between cells on one line.
- **For Lines.** The minimum space to maintain between lines of cells.

Adjusting Collection View Cell Padding



To add padding around cells so that space appears above, below, or on either side of the cells in a section, use the "Section Insets" fields in the Collection View Flow Layout Size Inspector. Specifying nonzero inset values reduces the amount of space available for laying out cells, which lets you limit the number of cells that can appear on one row or the number of rows that can appear in one section. The insets you can specify are:

- **Top.** The space to add between the bottom of the header view and the top of the first line of cells.
- **Bottom.** The space to add between the bottom last line of cells and the top of the footer
- **Left.** The space to add between the left edge of the cells and the left edge of the collection view.
- **Right.** The space to add between the right edge of the cells and the right edge of the collection view.

Using Auto Layout with Collection Views

You can create Auto Layout constraints between a collection view and other user interface elements. You can create any type of constraint for a collection view besides a baseline constraint.

For general information about using Auto Layout with iOS views, see "Using Auto Layout with Views" (page 20).

Making Collection Views Accessible

The data items in a collection view are accessible by default when they are represented by standard UIKit objects, such as `UILabel` and `UITextField`.

When a collection view changes its onscreen layout, it posts the `UIAccessibilityLayoutChangedNotification` notification.

For general information about making iOS views accessible, see "Making Views Accessible" (page 21).

Internationalizing Collection Views

For more information, see *Internationalization Programming Topics*.

Elements Similar to a Collection View

The following elements provide similar functionality to a collection view:

- **Table View.** A scrolling view that displays data items in a single-column list. For more information, see "Table Views" (page 88).
- **Scroll View.** A scrolling view that displays content without support for any specific layout or ordering scheme. For more information, see "Scroll Views" (page 69).

Image Views

An image view displays an image or an animated sequence of images. An image view lets you efficiently draw an image (such as a JPEG or PNG file) or an animated series of images onscreen, scaling the images automatically to fit within the current size of the view. Image views can optionally display a different image or series of images whenever the view is highlighted. Image views support the same file formats as the `UIImage` class—TIFF, JPEG, PNG, Windows bitmap (bmp), Windows icon (ico), Windows cursor (cur), and X Window System bitmap (xbm) formats.

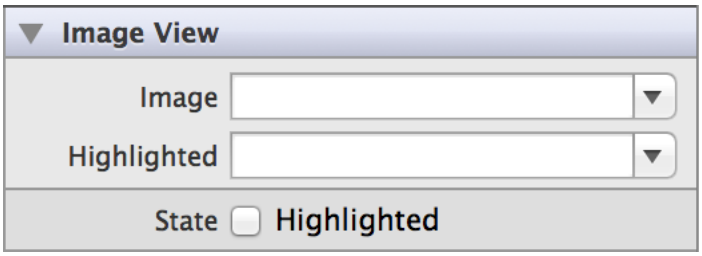
Image views allow the user to:

- View images within an app

Implementation: Image views are implemented in the `UIImageView` class and discussed in the *UIImageView Class Reference*.

Configuring Image Views

Generally, the easiest way to configure views—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



In addition, image views provide significant programmatic customization by modifying properties on the view objects and properties on whatever image object you have loaded into the view.

Setting Image View Content

If you are displaying a single image, most image views require minimal configuration beyond setting the image. If you are displaying an animated series of images, you must also configure the animation settings.

When you first use an image view object to display a single image, you can select an image to display using the Image (`image`) field in the Attributes Inspector. If you did not choose an image in the Attributes Inspector, you must set the initial image by calling `initWithImage:` or by setting the `image` property to a `UIImage` object that describes the image you want to display.

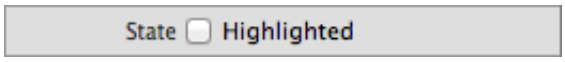
If you want to show a different image when the view is highlighted, you can specify this information in the Highlighted (`highlightedImage`) field. Alternatively, either call `initWithImage:highlightedImage:` when you initialize the image or set the `highlightedImage` property to the alternative image.

If you want your image view to display an animated sequence of images, you must do this programmatically. Because you cannot specify an array of images in the Attributes Inspector, you must write some code to tell the image view which images to use. To do this, set the `animationImages` property to an array of `UIImage` objects in the order in which they should be shown. Optionally set the `highlightedAnimationImages` property if you want to show a different animation while the view is highlighted.

Important: All images associated with a `UIImageView` object should have the same `scale` value. If your application uses images with different scales, they may render incorrectly.

Specifying Image View Behavior

Use the Highlighted (`highlighted`) checkbox to specify whether the view should show the standard or highlighted image or image sequence.



You can change the image view's state at any time.

If you are using an image sequence, you can instead configure the animation behavior programmatically:

- Set the `animationDuration` to the desired animation period (in seconds). By default, this property is computed based on the number of images at 30 frames per second.
- Set the `animationRepeatCount` to limit the number of iterations through the set of images. By default, this property has a value of zero, which means that the animation repeats forever.

You start the animation by calling `startAnimating`.

Customizing Image View Appearance

You cannot customize the appearance of an image view directly. However, you can determine how images appear in the view by setting properties at the `UIImage` and `UIView` levels.

Setting View Content Mode

The view's `contentMode` property specifies how the image should be scaled and aligned within the view. It is recommended (but not required) that you use images that are all the same size. If the images are different sizes, each will be adjusted to fit separately based on this mode.

Creating Images

The image's `capInsets`, `leftCapWidth`, and `topCapHeight` properties specify the width and height of a central portion of the image that should be scaled differently than the border areas (outside that central portion). The top and bottom border areas are tiled horizontally. The left and right border areas are tiled vertically. The corners are displayed as-is. Additionally, the image's `alignmentRectInsets` property specifies portions of the image to ignore for alignment purposes (such as shadow and glow effects).

You can create images for images views in a number of ways, including:

- Using the `imageWithAlignmentRectInsets:` method, which returns a derived image with nonzero alignment insets.
- Using the `resizableImageWithCapInsets:` or `resizableImageWithCapInsets:resizingMode:` methods, which return a derived *static* image with nonzero cap insets. The image's `resizingMode` property indicates whether the image should be scaled or tiled.
- Using the `animatedResizableImageNamed:capInsets:duration:` or `animatedResizableImageNamed:capInsets:resizingMode:duration:` methods, which return a derived *animated* image with nonzero cap insets.

These methods cannot be set after the image is created or loaded.

Customizing Image Transparency and Alpha Blending

Transparency of an image view is defined by properties of both the underlying image and the view as follows:

- If the view's Opaque (`opaque`) flag is set, the image is alpha blended with the background color of the view, and the view itself is opaque. The view's Alpha (`alpha`) setting is ignored.
- If the view's Opaque (`opaque`) flag is not set, the alpha channel for each pixel (or `1.0` if the image has no alpha channel) is multiplied by the view's Alpha (`alpha`) setting, and the resulting value determines the transparency for that pixel.

Important: It is computationally expensive to perform alpha compositing of non-opaque image views containing images with alpha channels. If you are not intentionally using the image alpha channel or the view's Alpha setting, you should set the Opaque flag to improve performance. See the last bullet point of "Debugging Image Views" (page 46) for more information.

Using Auto Layout with Image Views

You can create Auto Layout constraints between a image view and other user interface elements. You can create any type of constraint for a image view besides a baseline constraint.

You generally want the image view to fill the full width of your screen. To ensure that this happens correctly on all devices and orientations, you can create Leading Space to Superview and Trailing Space to Superview constraints, and set both values equal to `0`. This will ensure that the image view remains pinned to the edges of the device screen.

For general information about using Auto Layout with iOS views, see "Using Auto Layout with Views" (page 20).

Making Image Views Accessible

Image views are accessible by default. The default accessibility traits for a image view are Image and User Interaction Enabled.

For general information about making iOS views accessible, see "Making Views Accessible" (page 21).

Internationalizing Image Views

Internationalization of image views is automatic if your view displays only static images loaded from your app bundle. If you are loading images programmatically, you are at least partially responsible for loading the correct image.

- For resources in your app bundle, you do this by specifying the name in the attributes inspector or by calling the `imageNamed:` class method on `UIImage` to obtain the localized version of each image.
- For images that are not in your app bundle, your code must do the following:
  1. Determine which image to load in a manner specific to your app, such as providing a localized string that contains the URL.
  2. Load that image by passing the URL or data for the correct image to an appropriate `UIImage` class method, such as `imageWithData:` or `imageWithContentsOfFile:`.

Tip: Screen metrics and layout may also change depending on the language and locale, particularly if the internationalized versions of your images have different dimensions. Where possible, you should try to make minimize dimension differences in internationalized versions of image resources.

For more information, see *Internationalization Programming Topics*.

Debugging Image Views

When debugging issues with image views, watch for these common pitfalls:

- **Not loading your image with the correct method.** If you are loading an image from your app bundle, use `imageNamed:`. If you are loading an image from a file (with a full path or URL), use `imageWithContentsOfFile:`.
- **Not making animated image frames the same size.** This helps you avoid having to deal with scaling, tiling, or positioning differences between frames.
- **Not using a consistent scale value for all animated image frames.** Mixing images with different scale factors may produce undefined behavior.
- **Doing custom drawing in a subclass of an image view.** The `UIImageView` class is optimized to draw its images to the display. `UIImageView` does not call the `drawRect:` method of its subclasses. If your subclass needs to include custom drawing code, you should subclass the `UIView` class instead.
- **Not enabling event handling in subclasses if you need it.** New image view objects are configured to disregard user events by default. If you want to handle events in a custom subclass of `UIImageView`, you must explicitly change the value of the `userInteractionEnabled` property to `YES` after initializing the object.
- **Not providing prescaled images where possible.** For example, if you expect certain large images to be frequently displayed in a scaled-down thumbnail view, you might consider keeping the scaled-down images in a thumbnail cache. Scaling the image is a relatively expensive operation.
- **Not limiting image size.** Consider prescaling or tiling large images. The *MVCNetworking* sample code project (`QImageScrollView.m`) demonstrates how to determine what model of iOS device your software is running on. You can then use that information to help you determine the image dimension thresholds to use when scaling or tiling.
- **Not disabling alpha blending except where needed.** Unless you are intentionally working with images that contain transparency (drawing UI elements, for example), you should generally mark the view as opaque by selecting the Opaque checkbox in the Attributes Inspector, or setting the `opaque` property on the view itself.

  For views that are not opaque, the device must perform a lot of unnecessary computation if alpha blending is enabled and the image contains an alpha channel. This performance impact is further magnified if you are using Core Animation shadows, because the shape of the shadow is then based on the contents of the view, and must be dynamically computed.

  To learn more about how alpha blending works, see "Customizing Image Transparency and Alpha Blending" (page 45).

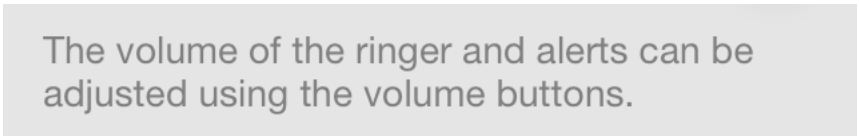
Elements Similar to an Image View

The following elements provide similar functionality to a web view:

- **Button.** You can set the background image of a button control (of type `UIButtonTypeCustom`). For more information, see "Buttons" (page 123).
- **Scroll View.** An image view typically scales content up or down to fit the dimensions of the view. If you need to display an image with user-controlled zooming and scaling, you should place that image view inside a scroll view. For more information, see "Scroll Views" (page 69).
- **Custom Views.** If you create a custom subclass of `UIView`, you can programmatically draw images inside its `drawRect:` method. (For maximum performance, you should do this only when absolutely necessary.) For more information, see "About Views" (page 14).

Labels

A label displays static text. Labels are often used in conjunction with controls to describe their intended purpose, such as explaining which value a button or slider affects.



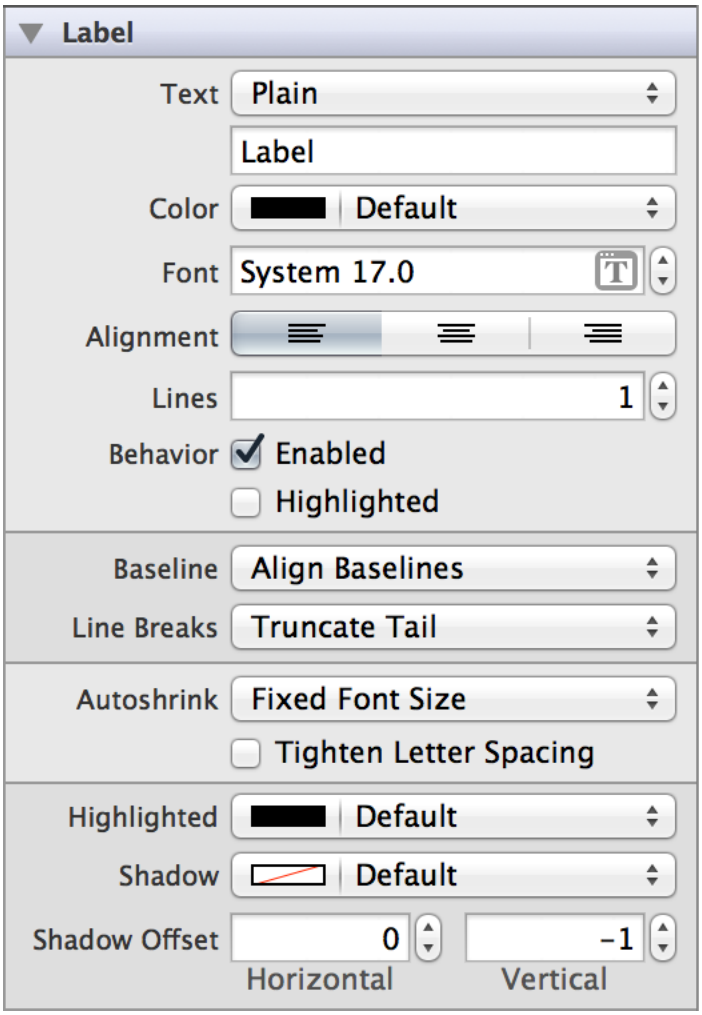
Labels allow the user to:

- Understand the purpose of controls in an app
- Receive instructions or context in an app

Implementation: Labels are implemented in the `UILabel` class and discussed in the *UILabel Class Reference*.

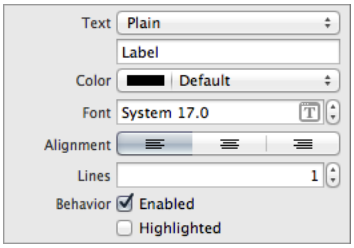
Configuring Labels

Generally, the easiest way to configure views—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



Setting Label Content

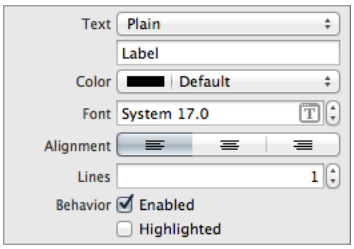
Set label content using the Label Text (`text` and `attributedText`) field in the Attributes Inspector. Both properties get set whether you specified the value of the Text field to be plain or attributed. For more information about attributed text, see "Specifying Text Appearance" (page 52).



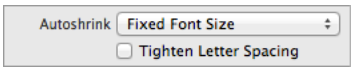
By default, a label is a single line. To create a multiline label, increase the value of the Lines (`numberOfLines`) field.

Specifying Label Behavior

You can specify whether a label is enabled or highlighted using the Enabled (`enabled`) and Highlighted (`highlighted`) checkboxes in the Attributes Inspector.

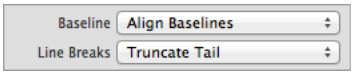


The Autoshrink (`adjustsFontSizeToFitWidth`) field is used to specify the manner in which font size will be reduced with the label's bounding rectangle.



The Fixed Font Size option is the equivalent of setting `adjustsFontSizeToFitWidth` to `NO`, meaning that font size will not adjust. Select the Minimum Font Scale (`minimumScaleFactor`) option to specify the smallest multiplier for the current font size that the font can scale down to, and the Minimum Font Size (`minimumFontSize`) option to specify the smallest font size that the font can scale down to.

Select the Tighten Letter Spacing (`adjustsLetterSpacingToFitWidth`) checkbox if you want the spacing between letters to be adjusted to fit the string within the label's bounding rectangle.

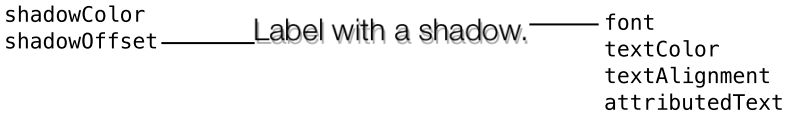


The Baselines (`baselineAdjustment`) field determines how to adjust the position of text in cases when the text must be drawn using a different font size than the one originally specified. For example, with the Align Baselines option, the position of the baseline remains fixed at its initial location while the text appears to move toward that baseline. Similarly, selecting the None option makes it appear as if the text is moving upwards toward the top-left corner of the bounding box.

Use the Line Breaks (`lineBreakMode`) field to specify the technique to use for wrapping and truncating the label's text if it exceeds a single line. Note that if this property is set to a value that causes text to wrap to another line, do not set the `adjustsFontSizeToFitWidth` or `adjustsLetterSpacingToFitWidth` property to `YES`.

Customizing Label Appearance

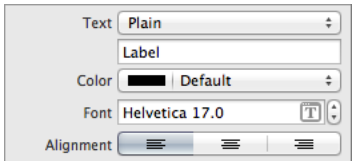
You can customize the appearance of a label by setting the properties depicted below.

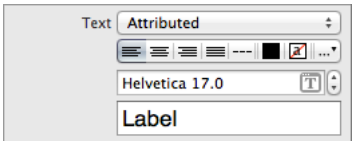


To customize the appearance of all labels in your app, use the appearance proxy (for example, `[UILabel appearance]`). For more information about appearance proxies, see "Using Appearance Proxies" (page 18).

Specifying Text Appearance

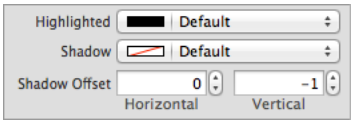
Labels can have one of two types of text: plain or attributed. Plain text supports a single set of formatting attributes—font, size, color, and so on—for the entire string. On the other hand, attributed text supports multiple sets of attributes that apply to individual characters or ranges of characters in the string.





Specifying Highlighted Appearance

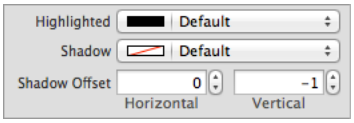
By default, the highlighted appearance of a label is no different than that of its normal control state.



However, you can create a different look for your label when it's in the `UIControlStateHighlighted` state by modifying its color in the Highlighted (`highlightedTextColor`) field.

Setting a Text Shadow

You can set a color for your label's shadow using the Shadow (`shadowColor`) field in the Attributes Inspector.



If you want to change how far the shadow is drawn from the button text, you can adjust the shadow offset. You can customize the offset for both dimensions using the Shadow Offset (`shadowOffset`) fields.

Using Auto Layout with Labels

You can create Auto Layout constraints between a label and other user interface elements. You can create any type of constraint for a label.

For general information about using Auto Layout with iOS views, see "Using Auto Layout with Views" (page 20).

Making Labels Accessible

Labels are accessible by default. The default accessibility trait for a label are Static Text and User Interaction Enabled.

For general information about making iOS views accessible, see "Making Views Accessible" (page 21).

Internationalizing Labels

For more information, see *Internationalization Programming Topics* .

Debugging Labels

When debugging issues with labels, watch for this common pitfall:

Specifying conflicting text wrapping and font adjustment settings. The `lineBreakMode` property describes how text should wrap or truncate within the label. If you set a value for this property that causes text to wrap to another line, do not set the `adjustsFontSizeToFitWidth` and `adjustsLetterSpacingToFitWidth` properties to `YES`, those fields ar used to scale the font size to fit into the label without adding line breaks.

Elements Similar to a Label

The following element provides similar functionality to a label:

Text Field. Text fields allows the user to input a single line of text into an app. You typically use text fields to gather small amounts of text from the user and perform some immediate action, such as a search operation, based on that text. For more information, see "Text Fields" (page 165).

Navigation Bars

Navigation bars allow you to present your app's content in an organized and intuitive way. A navigation bar is displayed at the top of the screen, and contains buttons for navigating through a hierarchy of screens. A navigation bar generally has a back button, a title, and a right button. The most common way to use a navigation bar is with a navigation controller. You can also use a navigation bar as a standalone object in your app.

Navigation bars allow the user to:

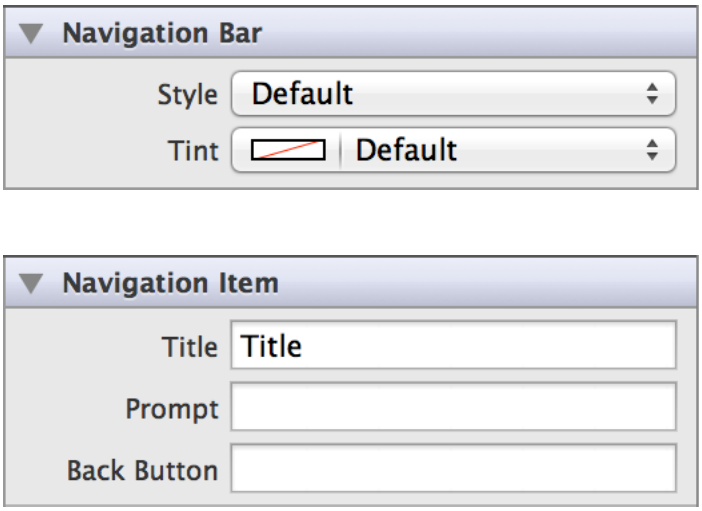
- Navigate to the previous view
- Transition to a new view

Implementation:

- Navigation bars are implemented in the `UINavigationBar` class and discussed in the *UINavigationBar Class Reference*.
- Navigation items are implemented in the `UINavigationItem` class and discussed in the *UINavigationItem Class Reference*.
- Bar button items are implemented in the `UIBarButtonItem` class and discussed in the *UIBarButtonItem Class Reference*.
- Bar items are implemented in the `UIBarItem` class and discussed in the *UIBarItem Class Reference*.

Configuring Navigation Bars

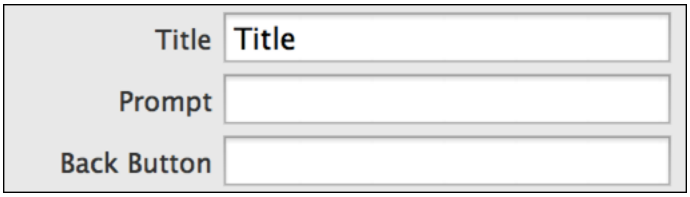
Generally, the easiest way to configure controls—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



Setting Navigation Bar Content

After you create a navigation bar, either in conjunction with a navigation controller or as a standalone object, you need to add content to the bar. A navigation bar can display a left button, title, prompt string, and right button.

The navigation bar displays information from a stack of `UINavigationItem` objects. At any given time, the `UINavigationItem` that is currently the `topItem` of the stack determines the title and other optional information in the navigation bar, such as the right button and prompt. The `UINavigationItem` that is immediately below the `topItem` is the `backItem`, which determines the appearance of the left or back button.



You can also add bar button items to a `UINavigationItem`. A `UIBarButtonItem` generally has a title and either a custom image or one of the system-supplied images listed in `UIBarButtonSystemItem`. It's common to have a right bar button, but you can also use a left bar button in the place of a back button.

To add any of these elements to a navigation bar, select the desired item from the Object library in Interface Builder and drag it to your storyboard. Then, you customize the contents in the Attributes Inspector as described in "Setting Navigation Bar Images" (page 59).

For more information about the elements that you add to a navigation bar, see *UINavigationItem Class Reference* and *UIBarButtonItem Class Reference*.

Specifying Navigation Bar Behavior

The most common way to use a navigation bar is with a `UINavigationController` object. A navigation controller manages the navigation between different screens of content for you. It also creates the navigation bar automatically, and pushes and pops navigation items as appropriate.

You can add a navigation controller to your app in Interface Builder or programmatically. To use Interface Builder to create a navigation controller, see "Creating a Navigation Interface Using a Storyboard" in *View Controller Catalog for iOS*. To create a navigation controller programmatically, see "Creating a Navigation Interface Programmatically" in *View Controller Catalog for iOS*.

A navigation controller automatically assigns itself as the delegate of its navigation bar object. Attempting to change the delegate raises an exception. For more information about using a navigation bar with navigation controller, see "Navigation Controllers".

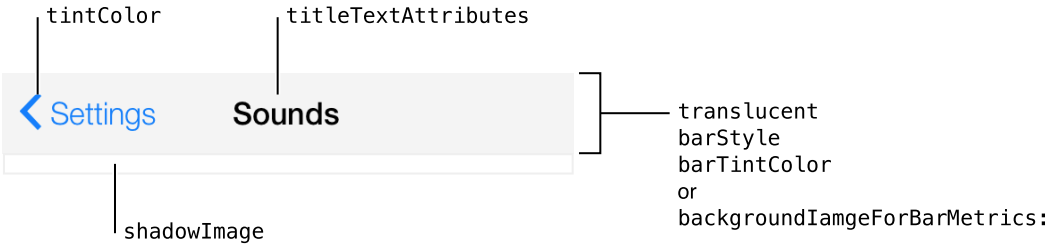
When you use a navigation bar as a standalone object, you set the initial appearance of the navigation bar by creating the appropriate `UINavigationItem` objects and adding them to the navigation bar object stack. When you create your standalone navigation bar in Interface Builder, Xcode creates the corresponding `UINavigationItem` objects for the elements you drag to the navigation bar.

You are also responsible for managing the stack of `UINavigationItem` objects when you use a navigation bar as a standalone object. You push new navigation items onto the stack using the `pushNavigationItem:animated:` method and pop items off the stack using the `popNavigationItemAnimated:` method. In addition to pushing and popping items, you can also set the contents of the stack directly using either the `items` property or the `setItems:animated:` method. You might use these methods at launch time to restore your interface to its previous state or to push or pop more than one navigation item at a time.

Assign a custom delegate object to the `delegate` property and use that object to intercept messages sent by the navigation bar. Delegate objects must conform to the `UINavigationBarDelegate` protocol. The delegate notifications let you track when navigation items are pushed or popped from the stack. You use these notifications to update the rest of your app's user interface. For more information about implementing a delegate object, see *UINavigationBarDelegate Protocol Reference*.

Customizing Navigation Bar Appearance

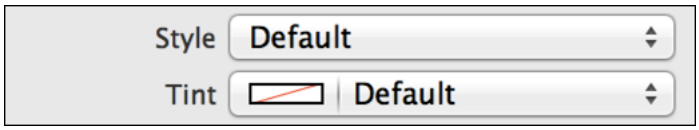
You can customize the appearance of a navigation bar by setting the properties depicted below.



To customize the appearance of all navigation bars in your app, use the appearance proxy (for example, `[UINavigationBar appearance]`). For more information about appearance proxies, see "Using Appearance Proxies" (page 18).

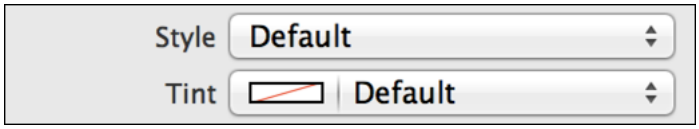
Specifying Navigation Bar Style

Navigation bars have two standard appearance styles: translucent white with dark text (default) or translucent black with light text. Use the Style (`barStyle`) field to select one of these standard styles.



Adjusting Navigation Bar Tint

You can specify a custom tint color for the navigation bar background using the Tint (`barTintColor`) field. The default background tint color is white.



Additionally, you can set a custom tint color for the interactive elements within a navigation bar—including button images and titles—programmatically using the `tintColor` property. The navigation bar will inherit its superview's tint color if a custom one is set, or show the default system blue color if none is set. For more information, see "Adjusting View Tint Color" (page 19).

Setting Navigation Bar Images

You can set a custom background image for your navigation bar. You can do this using `setBackgroundImage:forBarMetrics:`. Note that you must specify bar metrics because navigation bars have different dimensions on different devices and orientations.

You can also use a custom shadow image for the navigation bar using the `shadowImage` property. To show a custom shadow image, you must also set a custom background image.

Setting Navigation Bar Translucency

Navigation bars are translucent by default in iOS 7. Additionally, there is a system blur applied to all navigation bars. This allows your content to show through underneath the bar.

These settings automatically apply when you set any style for `barStyle` or any custom color for `barTintColor`. If you prefer, you can make the navigation bar opaque by setting the `translucent` property to `NO` programmatically. In this case, the bar draws an opaque background using black if the navigation bar has `UIBarStyleBlack` style, white if the navigation bar has `UIBarStyleDefault`, or the navigation bar's `barTintColor` if a custom value is defined.

If the navigation bar has a custom background image, the default translucency is automatically inferred from the average alpha values of the image. If the average alpha is less than 1.0, the navigation bar will be translucent by default; if the average alpha is 1.0, the search bar will be opaque by default. If you set the `translucent` property to `YES` on a navigation bar with an opaque custom background image, the navigation bar makes the image translucent. If you set the `translucent` property to `NO` on a navigation bar with a translucent custom background image, the navigation bar provides an opaque background for the image using black if the navigation bar has `UIBarStyleBlack` style, white if the navigation bar has `UIBarStyleDefault`, or the navigation bar's `barTintColor` if a custom value is defined.

Formatting Navigation Bar Title

The `titleTextAttributes` property specifies the attributes for displaying the bar's title text. You can specify the font, text color, text shadow color, and text shadow offset for the title in the text attributes dictionary, using the text attribute keys described in *NSString UIKit Additions Reference*.

You can adjust the vertical position of a navigation bar's title using the `setTitleVerticalPositionAdjustment:forBarMetrics:` method. Note that you must specify bar metrics because navigation bars have different dimensions for different devices and screen orientations.

Setting Bar Button Item Glyphs

Any bar button in a navigation bar can have a custom image instead of text. You can provide this image to your bar button item during initialization. Note that a bar button image will be automatically rendered as a template image within a navigation bar, unless you explicitly set its rendering mode to `UIImageRenderingModeAlwaysOriginal`. For more information, see "Using Template Images" (page 19).

Using Auto Layout with Navigation Bars

You can create Auto Layout constraints between a navigation bar and other user interface elements. You can create any type of constraint for a navigation bar besides a baseline constraint.

For general information about using Auto Layout with iOS views, see "Using Auto Layout with Views" (page 20).

Making Navigation Bars Accessible

Navigation bars are accessible by default. The default accessibility trait for a navigation bar is User Interaction Enabled.

With VoiceOver enabled on an iOS device, after the user navigates to a new view in the hierarchy, VoiceOver reads the navigation bar's title, followed by the name of the left bar button item. When the user taps on an element in a navigation bar, VoiceOver reads the name and the type of the element, such as, "General back button," "Keyboard heading," and "Edit button."

For general information about making iOS views accessible, see "Making Views Accessible" (page 21).

Internationalizing Navigation Bars

For more information, see *Internationalization Programming Topics*.

Debugging Navigation Bars

When debugging issues with navigation bars, watch for this common pitfall:

Specifying conflicting appearance settings. When customizing navigation bar appearance with a style or color, use one option or the other, but not both. Conflicting settings for navigation bar appearance will be resolved in favor of the most recently set value. For example, setting a new style clears any custom tint color you have set. Similarly, setting a custom tint color overrides any style you have set.

Elements Similar to a Navigation Bar

The following classes provide similar functionality to a navigation bar:

- **Toolbar.** A navigation controller can also manage a toolbar. On iPhone, this toolbar always appears at the bottom edge of the screen, but on iPad a toolbar can appear at the top of the screen. You can create a toolbar with a navigation controller, or as a standalone object. Unlike a navigation bar, which contains controls for navigating through a hierarchy of screens, a toolbar contains controls that perform actions related to the contents of the screen. For example, a toolbar might contain Share button and a Search button. For more information about toolbars, see "Toolbars" (page 104).
- **Tab Bar.** Similar to a navigation bar, a tab bar allows the user to switch between different views. However, a tab bar is persistent, which means that the user can select any tab from any other tab. By contrast, a navigation bar presents a linear path through various screens. For more information about tab bars, see "Tab Bars" (page 83).

Picker Views

A picker view lets the user choose between certain options by spinning a wheel on the screen. Picker views are well suited for choosing things like dates and times (as the date picker does) that have a moderate number of discrete options. Other examples include picking which armor to wear in a game and picking a font for text in a word processor. The list of options should be ordered in some logical way to make scanning easier.

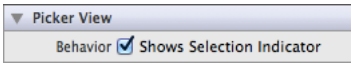
Picker views allow the user to:

- Quickly choose between a set of distinct options

Implementation: Date pickers are implemented in the `UIDatePicker` class and discussed in the *UIDatePicker Class Reference*.

Configuring Picker Views

Generally, the easiest way to configure views—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



Setting Picker View Content Programmatically

Populating a picker requires both a data source and a delegate. It is not possible to populate a picker in Interface Builder; you must do this programmatically.

Picker views need a delegate to display data and appropriately handle user interaction. The delegate adopts the `UIPickerViewDelegate` protocol and provides the content for each component's row, either as an attributed string, a plain string, or a view, and it typically responds to new selections or deselections. It also implements methods to return the drawing rectangle for rows in each component—these optional methods are only needed if the delegate returns a view as part of the picker's content.

Additionally, picker views require a data source. The data source adopts the `UIPickerViewDataSource` protocol and implements the required methods to return the number of components (columns) and the number of rows in each component. Note that the actual contents of each row comes from the delegate, not the data source.

For information about delegates and data sources, see "Delegates and Data Sources".

After setting the data source and delegate, set the initial selection by calling the `selectRow:inComponent:animated:` without animation. Typically this is done in a the `viewDidLoad` method of the view's view controller.

If the picker is visible, use animation when you update the selection.

You can dynamically change the rows of a component by calling the `reloadComponent:` method, or dynamically change the rows of all components by calling the `reloadAllComponents` method. When you call either of these methods, the picker view asks the delegate for new component and row data, and asks the data source for new component and row counts. Reload a picker view when a selected value in one component should change the set of values in another component. For example, changing a row value from February to March in one component should change a related component representing the days of the month.

Specifying Picker View Behavior

You cannot customize the picker view's selection indicator on iOS 7. The selection indicator is always shown, so toggling the Shows Selection Indicator (`showsSelectionIndicator`) box has no effect.

Customizing Picker View Appearance

You cannot customize the appearance of picker views.

Using Auto Layout with Picker Views

You can create an Auto Layout constraint between a picker view and other user interface elements. You can create any type of constraint for a picker view besides a baseline constraint.

Picker views usually reside at the bottom of the screen in all device orientations. Select Bottom Space to Superview and set the relation equal to 0 for the date picker to pin to the bottom of the screen in all device orientations.

For general information about using Auto Layout with iOS views, see "Using Auto Layout with Views" (page 20).

Making Picker Views Accessible

Picker views are accessible by default. The default accessibility trait for a picker view is Adjustable.

For general information about making iOS views accessible, see "Making Views Accessible" (page 21).

Internationalizing Picker Views

To internationalize picker view, you must provide localized translations of each string in the picker.

For more information, see *Internationalization Programming Topics*.

Debugging Picker Views

When debugging issues with picker views, watch for this common pitfall:

Not testing localizations. Be sure to test the pickers in your app with the localizations you intend to ship.

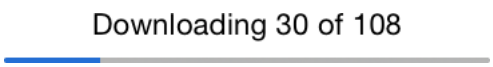
Elements Similar to a Picker View

The following elements provide similar functionality to a picker view:

- **Date Picker.** Uses a picker to let the user pick a date and time. For more information, see "Date Pickers" (page 131).
- **Stepper.** Lets the user increment and decrement a value. For more information, see "Steppers" (page 156).

Progress Views

A progress view is used to illustrate the progress of a task over time in an app. You use a progress view to let the user know how long until an operation completes, such as a download. The Mail app uses progress views in several different situations, including when it's downloading new messages or sending an outgoing message.



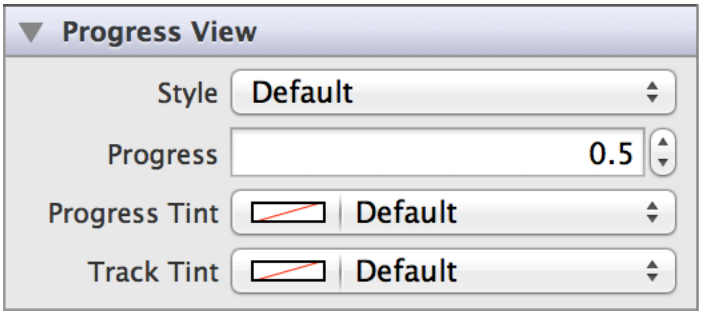
Progress views allow the user to:

- Receive feedback on a loading operation.
- See an estimate of how much time is left until a task finishes.

Implementation: Progress views are implemented in the `UIProgressView` class and discussed in the *UIProgressView Class Reference*.

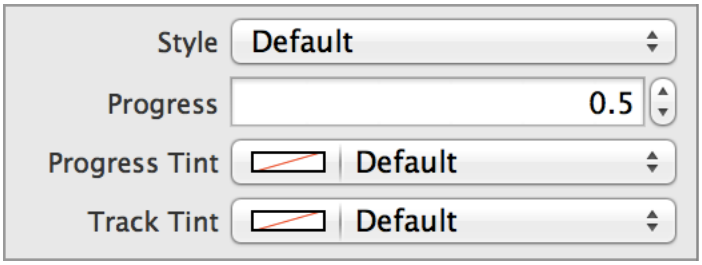
Configuring Progress Views

Generally, the easiest way to configure views—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



Setting Progress View Value

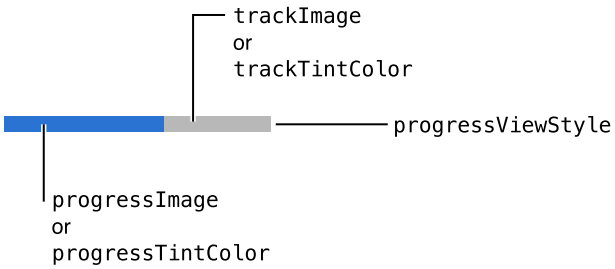
You can set the initial progress as a float between 0 and 1 by using the Progress (`progress`) field in the Attributes Inspector. You can also do this programmatically using the `setProgress:animated:` method without animation. This is typically done in the `viewDidLoad` method of the view controller.



If the progress view is visible, use animation when you update the progress.

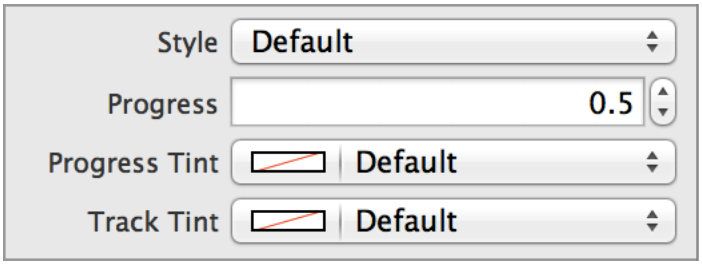
Customizing Progress View Appearance

You can customize the appearance of a progress view by setting the properties depicted below.

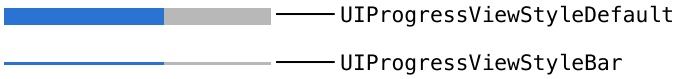


To customize the appearance of all progress views in your app, use the appearance proxy (for example, `[UIProgressView appearance]`). For more information about appearance proxies, see "Using Appearance Proxies" (page 18).

Selecting Progress View Style

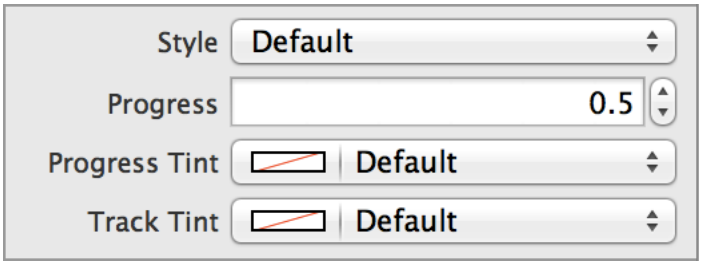


With progress views, you have two style options: default and bar. You can specify either value by using the Style (`progressViewStyle`) field. There is no difference in functionality, only in appearance. The `Default` style has blue-tinted progress and a gray track; the `Bar` style is a thinner version of the default.



Adjusting Progress View Tint

You can adjust the tint of two parts of the progress view: the track and the progress. The track tint applies to the area of the track that is not filled, while the progress tint applies to the portion of the progress bar that is filled. Use the Track Tint (`trackTintColor`) and Progress Tint (`progressTintColor`) fields to set custom colors for the track and progress. You can adjust the tint for either style of the progress view.



Using Auto Layout with Progress Views

You can create Auto Layout constraints between a progress view and other user interface elements. You can create any type of constraint for a progress view besides a baseline constraint.

To keep a progress view centered and adjust its width according to device orientation or screen size, you can use Auto Layout to pin it to its superview. Using the Auto Layout Pin menu, create Leading Space to Superview and Trailing Space to Superview constraints and set their values equal to each other. Doing this ensures that the endpoints of your progress view are a specified distance from the edges of its superview. With these constraints, the progress view stays centered and its width adjusts automatically for different device orientations and screen sizes.

For general information about using Auto Layout with iOS views, see "Using Auto Layout with Views" (page 20).

Making Progress Views Accessible

Progress views are accessible by default. The default accessibility traits for a progress view are Updates Frequently and User Interaction Enabled. The Updates Frequently accessibility trait means that the progress view doesn't send update notifications when its state changes. This trait tells an assistive app that it should poll for changes in the progress view when necessary.

For general information about making iOS views accessible, see "Making Views Accessible" (page 21).

Internationalizing Progress Views

Progress views have no special properties related to internationalization. However, if you use a progress view with a label, make sure you provide localized strings for the label.

For more information, see *Internationalization Programming Topics*.

Elements Similar to a Progress View

The following element provides similar functionality to a progress view:

Activity Indicator View. For an indeterminate progress indicator—or, informally, a "spinner"—use an activity indicator view. For more information, see "Activity Indicators" (page 26).

Scroll Views

A scroll view allows users to see content that is larger than the scroll view's boundaries. When a scroll view first appears—or when users interact with it—vertical or horizontal scroll indicators flash briefly to show users that there is more content they can reveal. Other than the transient scroll indicators, a scroll view has no predefined appearance.

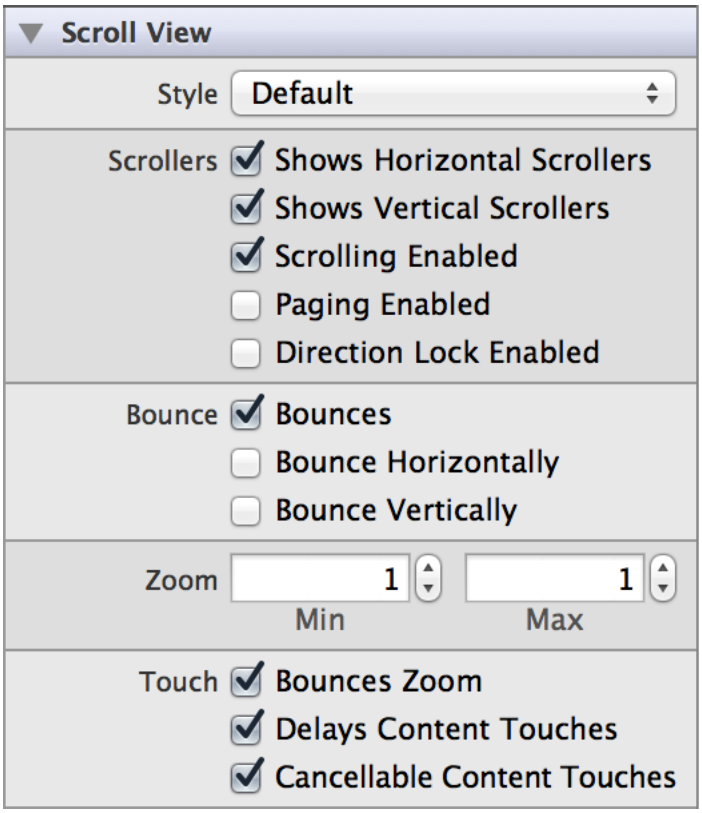
Scroll views allow the user to:

- View content that does not fit on the screen of the device

Implementation: Scroll views are implemented in the `UIScrollView` class and discussed in the *UIScrollView Class Reference*

Configuring Scroll Views

Generally, the easiest way to configure views—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



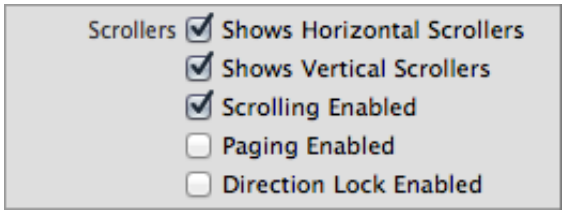
Setting Scroll View Content

You set scroll view content programmatically by adding subviews to its content view with the `addSubview:` method.

Specifying Scroll View Behavior

Scroll views need a delegate to handle scrolling, dragging, and zooming. By assigning a view controller as the scroll view's delegate and implementing any or all of the `UIScrollViewDelegate` methods, you can define these behaviors.

A scroll view responds to the speed and direction of gestures to reveal content in a way that feels natural to people. When users drag content in a scroll view, the content follows the touch; when users flick content, the scroll view reveals the content quickly and stops scrolling when the user touches the screen or when the end of the content is reached. A scroll view can also operate in paging mode, in which each drag or flick gesture reveals one app-defined page of content.

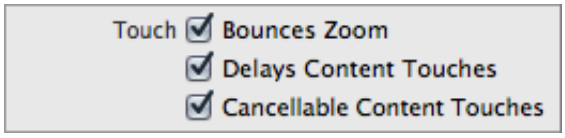


Use the Shows Horizontal Scrollers (`showsHorizontalScrollIndicator`) and Shows Vertical Scrollers (`showsVerticalScrollIndicator`) boxes to specify whether the corresponding scroll indicator should be visible during tracking and fades out after tracking. These options are enabled by default; toggle off is you do not want the scroller to be shown.

You can specify whether scrolling is enabled or disabled in the scroll view using the Scrolling Enabled (`scrollEnabled`) checkbox. Scrolling is enabled by default. When scrolling is disabled, the scroll view does not accept touch events; it forwards them up the responder chain.

If you check the Paging Enabled (`pagingEnabled`) box, the the scroll view stops on multiples of the scroll view's bounds when the user scrolls, giving the effect of scrolling through a single page at a time.

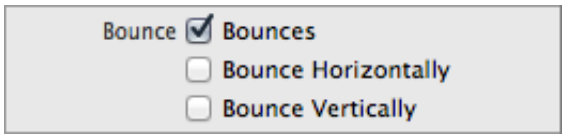
If you turn on directional lock by checking the Direction Lock Enabled (`directionalLockEnabled`) box, a user will only be able to scroll in one direction at a time. By default, a user can scroll in both directions, or diagonally.



If the Bounces Zoom (`bouncesZoom`) option is enabled, when zooming exceeds either the maximum or minimum limits for scaling, the scroll view temporarily animates the content scaling just past these limits before returning to them. If this option is disabled, zooming stops immediately at one a scaling limits.

Using the Delays Content Touches (`delaysContentTouches`) checkbox, you can specify whether the scroll view delays the handling of touch-down gestures. When enabled, the view delays handling the touch-down gesture until it can determine if scrolling is the intent.

You can indicate whether touches in the content view always lead to tracking by using the Cancellable Content Touches (`canCancelContentTouches`) checkbox. When enabled, if a user drags their finger enough to initiate a scroll within a view in the content that has begun tracking a finger touching it, that view receives a `touchesCancelled:withEvent:` message and the scroll view handles the touch as a scroll. When disabled, the scroll view does not scroll regardless of finger movement once the content view starts tracking.

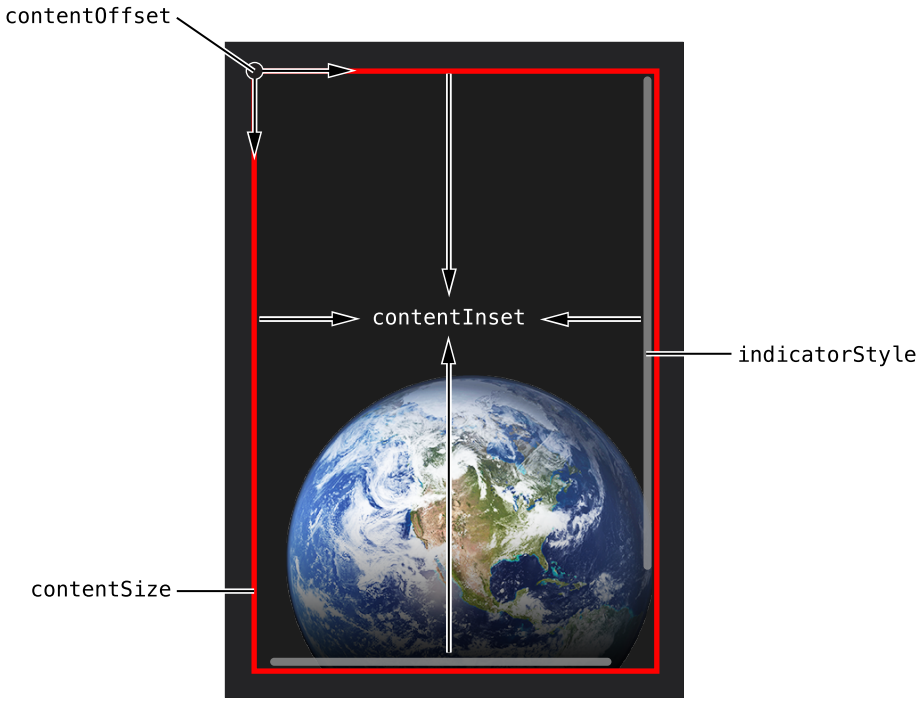


Use the Bounces (`bounces`) checkbox to indicate whether the scroll view bounces past the edge of content and back again. Enable Bounce Horizontally (`alwaysBounceHorizontal`) if you want content to bounce when scrolled horizontally, and Bounce Vertically (`alwaysBounceVertical`) if you want content to bounce when scrolled vertically.

You can use the Min Zoom (`minimumZoomScale`) and Max Zoom (`maximumZoomScale`) fields to specify how much the scroll view's content can be zoomed. The maximum zoom scale must be greater than the minimum zoom scale for zooming to be enabled. The default value is 1.0.

Customizing Scroll View Appearance

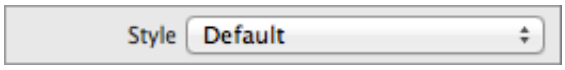
You can customize the appearance of a scroll view by setting the properties depicted below.



To customize the appearance of all scroll views in your app, use the appearance proxy (for example, `[UIScrollView appearance]`). For more information about appearance proxies, see "Using Appearance Proxies" (page 18).

Setting Scroll View Style

The only way to customize the appearance of a scroll view is by setting the style of the scroll indicators. There are three different choices for indicator style: default (black with a white border), white, or black. You can set the indicator style using the "Style" (`indicatorStyle`) field in the Attributes Inspector.



Specifying Scroll View Content Layout

Scroll views have a number of options that dictate how their content is laid out. You specify the size of the content using the `contentSize` property, which is initially set to zero. You can use the `contentInset` property to specify a content inset, which is the distance that the content is padded or inset from the enclosing scroll view. Additionally, you can use the `contentOffset` property or the `setContentOffset:animated:` method to set the point at which the origin of the content view is offset from the origin of the scroll view.

Using Auto Layout with Scroll Views

You can create Auto Layout constraints between a scroll view and other user interface elements. You can create any type of constraint for a scroll view besides a baseline constraint.

For general information about using Auto Layout with iOS views, see "Using Auto Layout with Views" (page 20).

Making Scroll Views Accessible

Scroll views are accessible by default. The default accessibility trait for a scroll view is "User Interaction Enabled."

For general information about making iOS views accessible, see "Making Views Accessible" (page 21).

Internationalizing Scroll Views

For more information, see *Internationalization Programming Topics*.

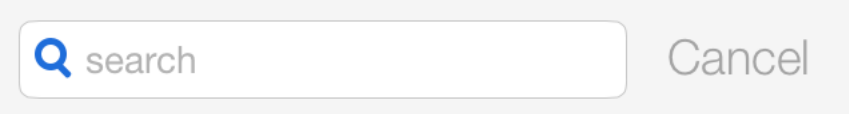
Elements Similar to a Scroll View

The following elements provide similar functionality to a scroll view:

- **Table View.** A scrolling view that displays data items in a single-column list. For more information, see "Table Views" (page 88).
- **Collection View.** A scrollable view that displays an ordered collection of data items using standard or custom layouts. Similar to a table view, a collection view gets data from your custom data source objects and displays it using a combination of cell, layout, and supplementary views. For more information, see "Collection Views" (page 35).

Search Bars

A search bar provides an interface for text-based searches with a text box and buttons such as search and cancel. A search bar accepts text from users, which can be used as input for a search (shown here with placeholder text). A scope bar, which is available only in conjunction with a search bar—allows users to define the scope of a search (shown here below a search bar).



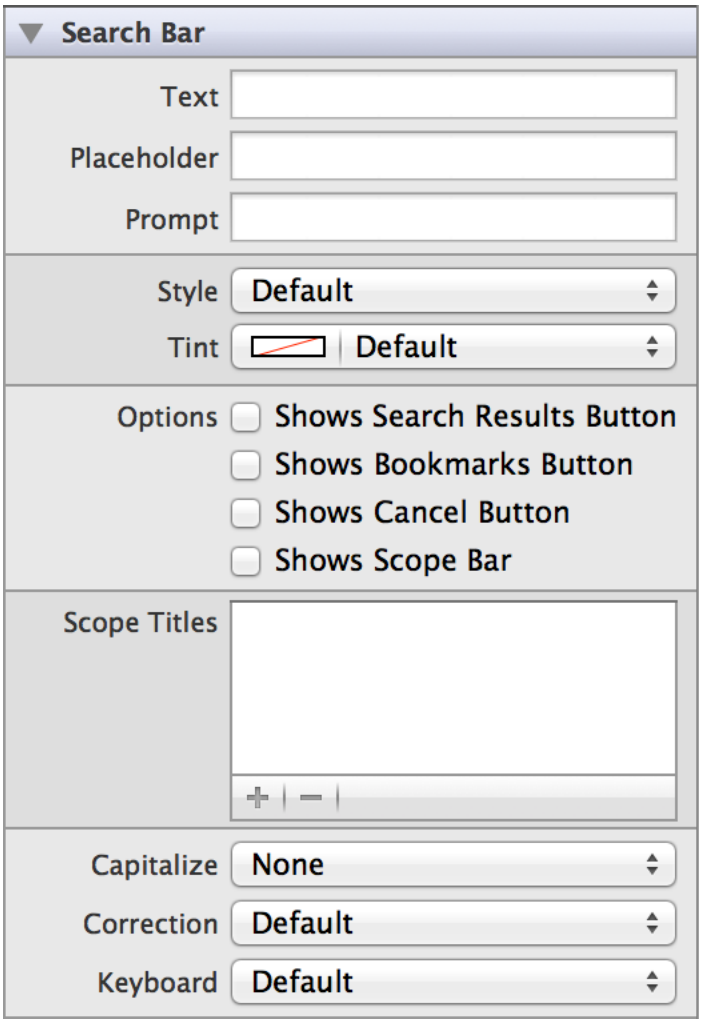
Search bars allow the user to:

- Quickly find a value in a large collection
- Create a scope filter

Implementation: Search bars are implemented in the `UISearchBar` class and discussed in the *UISearchBar Class Reference*.

Configuring Search Bars

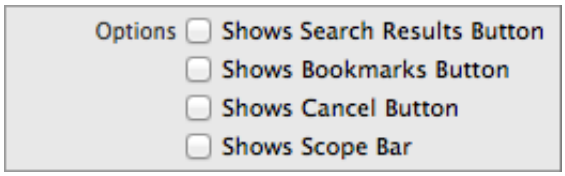
Generally, the easiest way to configure views—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



Setting Search Bar Content

The Text (`text`) field contains the current search text; you can use it to set an initial search. Don't use it to provide a description of the search; use placeholder text instead. Placeholder text is specified in the Placeholder (`placeholder`) field, and is visible only when there is no other text in the search field. Placeholder text is styled differently to communicate its different meaning to the user and it is automatically cleared when the user starts typing. It is suitable for very short descriptions of what the user should enter in the search field.

The prompt text is specified in the Prompt (`prompt`) field. It appears directly above the search bar. Unlike the placeholder text, the prompt text is visible whether or not the user has entered text in the search field, so it is suitable for longer descriptions or directions.



Search bars can display a number of different buttons. The Cancel button is intended to terminate a search operation; you can display this button by selecting the Shows Cancel Button checkbox. The Search Results and Bookmarks buttons appear in the right side of search bar, and can be toggled to display those respective views. You can display one of these buttons by selecting either the Shows Search Results Button (`showsSearchResultsButton`) or Shows Bookmarks Button (`showsBookmarkButton`) checkbox. Note that you cannot display both of these buttons simultaneously; if both properties are enabled, only the Search Results button is visible.

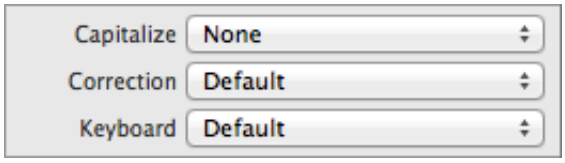
> **Note:** These buttons are merely user interface elements and have no functionality. You must implement the appropriate functionality yourself using the corresponding `UISearchBarDelegate` methods.

A search bar can also display a scope bar, which lets users limit the scope of a search. For example, when searching in an email app, the user could restrict the search to the Inbox or to a particular folder. To display a scope bar, check the Shows Scope Bar (`showsScopeBar`) box and add an array of scope bar titles as strings to the Scope Titles (`scopeButtonTitles`) field.

Specifying Search Bar Behavior

Search bars need a delegate to handle user interaction. You implement the `UISearchBarDelegate` protocol on a delegate object to respond to user actions—for example, performing the search. Every search bar needs a delegate object that implements the `UISearchBarDelegate` protocol. The delegate is responsible for taking actions in response to user input such as editing the search text, starting or canceling a search, and tapping in the scope bar. At the very minimum, the delegate needs to perform a search after text is entered in the text field.

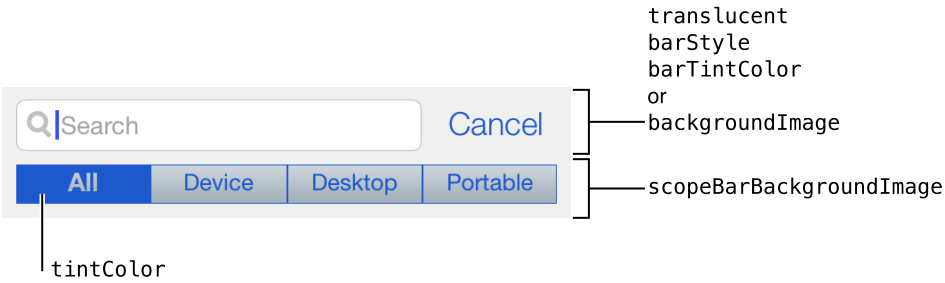
A user types content into a search bar using a keyboard, which has a number of customization options:



- **Keyboard layout.** The Keyboard field allows you to select from a number of different keyboard layouts. Match the keyboard layout to the purpose and scope of the search bar. The default keyboard layout is an alphanumeric keyboard in the device's default language. For a list of possible keyboard types, see `UIKeyboardType`.
- **Capitalization scheme.** The Capitalization field specifies how text should be capitalized in the search bar: no capitalization, every word, every sentence, or every character. The no capitalization scheme is selected by default.
- **Auto-correction.** The Correction field simply disables or enables auto-correct in the search bar.

Customizing Search Bar Appearance

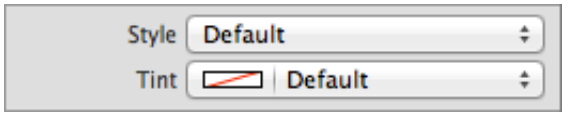
You can customize the appearance of a search bar by setting the following properties:



To customize the appearance of all search bars in your app, the appearance proxy (for example, `[UISearchBar appearance]`). For more information about appearance proxies, see "Using Appearance Proxies" (page 18).

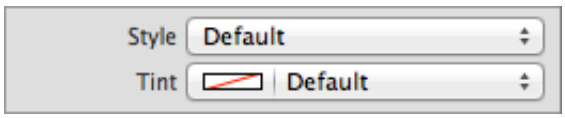
Specifying Search Bar Style

Bars have two standard appearance styles: translucent white with dark text (default) or translucent black with light text. Use the Style (`barStyle`) field to select one of these standard styles.



Adjusting Search Bar Tint

You can specify a custom tint color for the search bar background using the Tint (`barTintColor`) field. The default background tint color is white.



Additionally, you can set a custom tint color for the interactive elements within a search bar—including the scope bar, cancel button, and cursor—programmatically using the `tintColor` property. The search bar will inherit its superview's tint color if a custom one is set, or show the default system blue color if none is set. For more information, see "Adjusting View Tint Color" (page 19).

Specifying Search Bar Background Images

A search bar can have a background image that covers the area behind the search field. Use the `backgroundImage` property to set a background image for your search bar. You can also set the background image for a search bar's scope bar using the `scopeBarBackgroundImage` property. Single-pixel images or stretchable images are stretched; otherwise, the image is tiled. If you set one of these background image properties, you should also set the other to give your app interface a consistent look.

Setting Search Bar Translucency

Search bars are translucent by default on iOS 7. Additionally, there is a system blur applied to all search bars. This allows your content to show through underneath the bar.

These settings automatically apply when you set any style for `barStyle` or any custom color for `barTintColor`. If you prefer, you can make the search bar opaque by setting the `translucent` property to `NO` programmatically. In this case, the bar draws an opaque background using black if the search bar has `UIBarStyleBlack` style, white if the search bar has `UIBarStyleDefault`, or the search bar's `barTintColor` if a custom value is defined.

If the search bar has a custom background image, the default translucency is automatically inferred from the average alpha values of the image. If the average alpha is less than 1.0, the search bar will be translucent by default; if the average alpha is 1.0, the search bar will be opaque by default. If you set the `translucent` property to `YES` on a search bar with an opaque custom background image, the search bar makes the image translucent. If you set the `translucent` property to `NO` on a search bar with a translucent custom background image, the search bar provides an opaque background for the image using black if the search bar has `UIBarStyleBlack` style, white if the search bar has `UIBarStyleDefault`, or the search bar's `barTintColor` if a custom value is defined.

Specifying Search Bar Layout

You can also control certain aspects of the search bar's layout by providing position adjustments: for icons using the `positionAdjustmentForSearchBarIcon:` method, for the background image using the `searchFieldBackgroundPositionAdjustment` property, and for search text using the `searchTextPositionAdjustment` property.

Using Auto Layout with Search Bars

You can create Auto Layout constraints between a search bar and other user interface elements. You can create any type of constraint for a search bar besides a baseline constraint.

For general information about using Auto Layout with iOS views, see "Using Auto Layout with Views" (page 20).

Making Search Bars Accessible

Search bars are accessible by default.

For general information about making iOS views accessible, see "Making Views Accessible" (page 21).

Internationalizing Search Bars

To internationalize a search bar, you must provide localized strings for the following properties:

- `placeholder`
- `prompt`
- `text`
- `scopeButtonTitles`

For more information, see *Internationalization Programming Topics*.

Debugging Navigation Bars

When debugging issues with navigation bars, watch for these common pitfalls:

- **Specifying conflicting appearance settings.** When customizing search bar appearance with a style or color, use one option or the other, but not both. Conflicting settings for search bar appearance will be resolved in favor of the most recently set value. For example, setting a new style clears any custom tint color you have set. Similarly, setting a custom tint color overrides any style you have set.
- **Performance issues.** If search operations can be carried out very rapidly, it is possible to update the search results as the user is typing by implementing the `searchBar:textDidChange:` method on the delegate object. However, if a search operation takes more time, you should wait until the user taps the Search button before beginning the search in the `searchBarSearchButtonClicked:` method. Always perform search operations a background thread to avoid blocking the main thread. This keeps your app responsive to the user while the search is running and provides a better user experience.

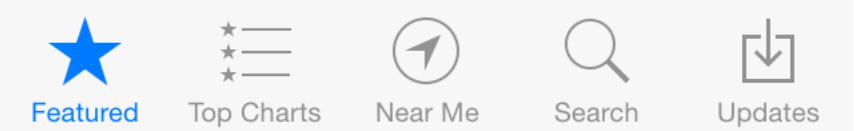
Elements Similar to a Search Bar

The following element provides similar functionality to a search bar:

Toolbar. A toolbar object contains controls that allow the user to perform actions related to objects onscreen. For more information, see "Toolbars" (page 104).

Tab Bars

A tab bar provides easy access to different views in an app. Use a tab bar to organize information in your app by subtask. The most common way to use a tab bar is with a tab bar controller. You can also use a tab bar as a standalone object in your app.



Tab bars allow the user to:

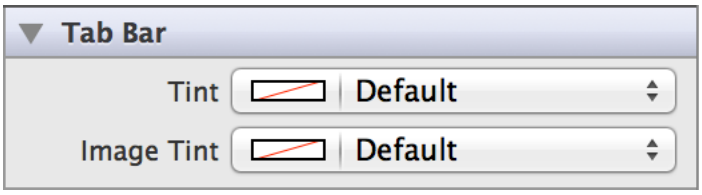
- Quickly navigate within an app
- Get an understanding of the app's layout

Implementation:

- Tab bars are implemented in the `UITabBar` class and discussed in the *UITabBar Class Reference*.
- Tab bar items are implemented in the `UITabBarItem` class and discussed in the *UITabBarItem Class Reference*.

Configuring Tab Bars

Generally, the easiest way to configure views—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



Specifying Tab Bar Content

Each tab on a tab bar is represented as a `UITabBarItem`, and you use the `UITabBarItem` class methods to create a tab bar item. Each tab bar item has a title, selected image, unselected image, and a badge value.

After you create your tab bar items, add them to your tab bar with the `items` property, which is an array of `UITabBarItem` objects. If you want to animate changes to your tab bar items array, use the `setItems:animated:` method instead.

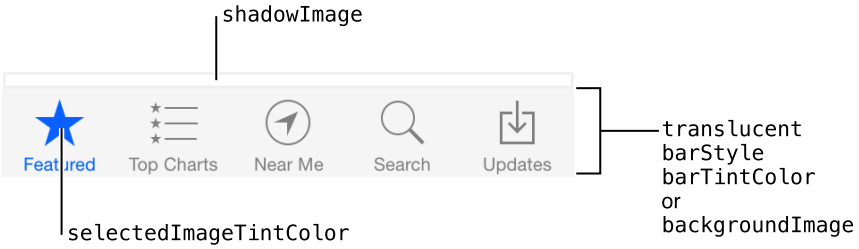
Specifying Tab Bar Behavior

You can change the contents of a tab bar at runtime, and allow users to add, remove, or reorder tabs. To present a modal view that allows users to customize a tab bar, use the `beginCustomizingItems:` method. You can also add a `UITabBarDelegate` object to your app. The tab bar delegate receives messages when the user customizes the tab bar.

The most common way to use a tab bar is in conjunction with a tab bar controller. A `UITabBarController` object manages the various tab views and view controllers, and the tab bar itself. If you use a tab bar controller, you should not use the `UITabBar` methods or properties to modify the tab bar. If you do, the system throws an exception. For more information about how to create a tab bar interface with an associated tab bar controller, see "Tab Bar Controllers".

Customizing Tab Bar Appearance

You can customize the appearance of a tab bar by setting the properties depicted below.



To customize the appearance of all tab bars in your app, use the appearance proxy (for example, `[UITabBar appearance]`). For more information, see "Using Appearance Proxies" (page 18).

Specifying Tab Bar Style

Tab bars have two standard appearance styles: translucent white with dark text (default) or translucent black with light text. Use the `barStyle` property to programmatically select one of these standard styles.

Adjusting Tab Bar Tint

You can specify a custom tint color for the bar background using the Tint (`barTintColor`) field. The default background tint color is white.

Use the Image Tint (`selectedImageTintColor`) field to specify the bar item's tint color when that tab is selected. By default, that color is blue.

Setting Tab Bar Images

The selection indicator image is shown when a tab is selected. It is drawn on top of the tab bar, behind the bar item icon. By default, there is no selection indicator image, but you can set a custom one using the `selectionIndicatorImage` property.

By default, there is no divider image between tabs on a tab bar. You can set custom divider images for each combination of left and right tab control states using the `setDividerImage:forLeftState:rightState:` method. If you use custom dividers, make sure to set divider images all combinations of tabs states: left selected, right selected, or both unselected.

You can also set a custom background image for your entire tab bar using the `backgroundImage` property. If you set this property with a stretchable image, the image is stretched. If you use a non-stretchable image, the image is tiled.

If you want to use custom shadow image for the tab bar, set the `shadowImage` property. To show a custom shadow image, you must also set a custom background image with `backgroundImage`.

Setting Tab Bar Translucency

Tab bars are translucent by default on iOS 7. Additionally, there is a system blur applied to all tab bars. This allows your content to show through underneath the bar.

These settings automatically apply when you set any style for `barStyle` or any custom color for `barTintColor`. If you prefer, you can make the tab bar opaque by setting the `translucent` property to `NO` programmatically. In this case, the bar draws an opaque background using black if the tab bar has `UIBarStyleBlack` style, white if the tab bar has `UIBarStyleDefault`, or the tab bar's `barTintColor` if a custom value is defined.

If the tab bar has a custom background image, the default translucency is automatically inferred from the average alpha values of the image. If the average alpha is less than 1.0, the tab bar will be translucent by default; if the average alpha is 1.0, the tab bar will be opaque by default. If you set the `translucent` property to `YES` on a tab bar with an opaque custom background image, the tab bar makes the image translucent. If

you set the `translucent` property to `NO` on a tab bar with a translucent custom background image, the tab bar provides an opaque background for the image using black if the tab bar has `UIBarStyleBlack` style, white if the tab bar has `UIBarStyleDefault`, or the tab bar's `barTintColor` if a custom value is defined.

Setting Tab Bar Item Glyphs

Each item in a tab bar can have a custom selected image and unselected image. You can specify these images when you initialize a tab bar item using the `initWithTitle:image:selectedImage:` method. Note that a tab bar item image will be automatically rendered as a template image within a tab bar, unless you explicitly set its rendering mode to `UIImageRenderingModeAlwaysOriginal`. For more information, see "Using Template Images" (page 19).

Using Auto Layout with Tab Bars

You can create Auto Layout constraints between a tab bar and other user interface elements. You can create any type of constraint for a tab bar besides a baseline constraint.

For general information about using Auto Layout with iOS views, see "Using Auto Layout with Views" (page 20).

Making Tab Bars Accessible

Tab bars are accessible by default.

With VoiceOver enabled on an iOS device, when a user touches a tab in a tab bar, VoiceOver reads the title of the tab, its position in the bar, and whether it is selected. For example in the iTunes app on iPad, you might hear "Selected, Audiobooks, four of seven" or "Genius, six of seven."

For general information about making iOS views accessible, see "Making Views Accessible" (page 21).

Internationalizing Tab Bars

To internationalize a tab bar, you must provide localized strings for the tab bar item titles.

For more information, see *Internationalization Programming Topics*.

Elements Similar to a Tab Bar

The following classes provide similar functionality to a tab bar:

- **Segmented Control.** Similar to a tab bar, a segmented control functions as a button that shows different views. If you want to provide functionality similar to a tab bar, but don't want those controls to be persistent, consider using a segmented control. Remember that a tab bar should be accessible from every location in an app. For more information, see "Segmented Controls" (page 142).
- **Navigation Bar.** A navigation bar also allows users to navigate through different content views, but it offers a linear path. With a tab bar, a user can view any other tab at any given time. For more information, see "Navigation Bars" (page 55).
- **Toolbar.** Both a tab bar and a toolbar are always visible onscreen. However, unlike a tab bar, which switches between views, a `UIToolbar` object contains controls that allow the user to perform actions related to objects onscreen. For more information, see "Toolbars" (page 104).

Table Views

A table view presents data in a scrollable list of multiple rows that may be divided into sections. It presents data in a single-column list of multiple rows and is a means for displaying and editing hierarchical lists of information. For instance, the Mail application uses a table view to display email messages in a user's inbox.

In normal mode, selecting a message allows the user to read it. In editing mode, selecting a message allows the user to delete it from the inbox. Table views provide a simple yet versatile interface for managing and interacting with collections of data.

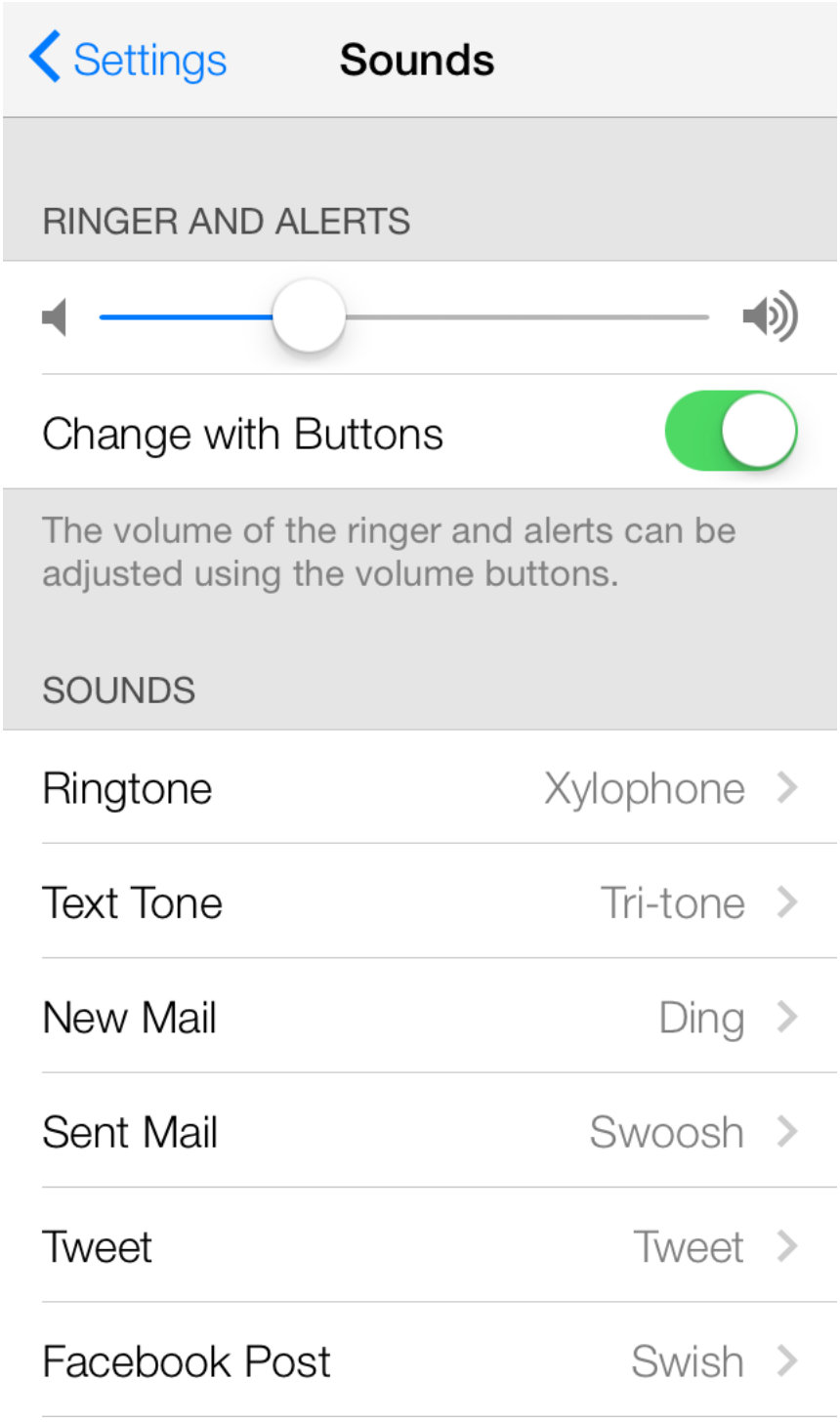


Table views allow the user to:

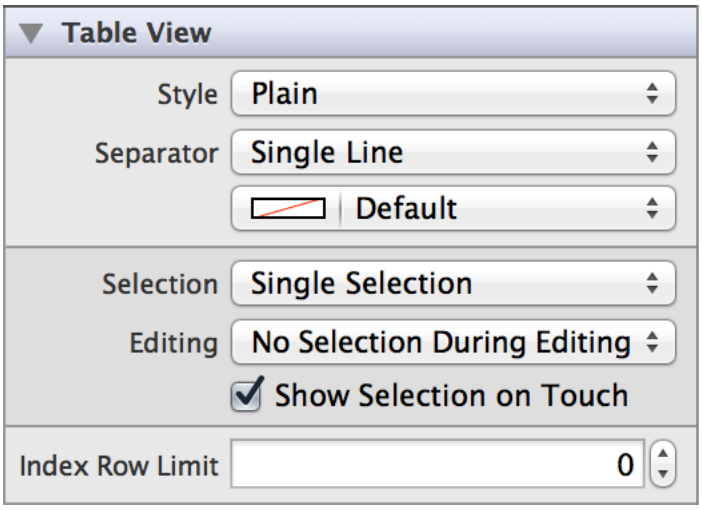
- Navigate through hierarchically structured data
- View an indexed list of items
- See detail information and controls in visually distinct groupings
- Interact with a selectable list of options

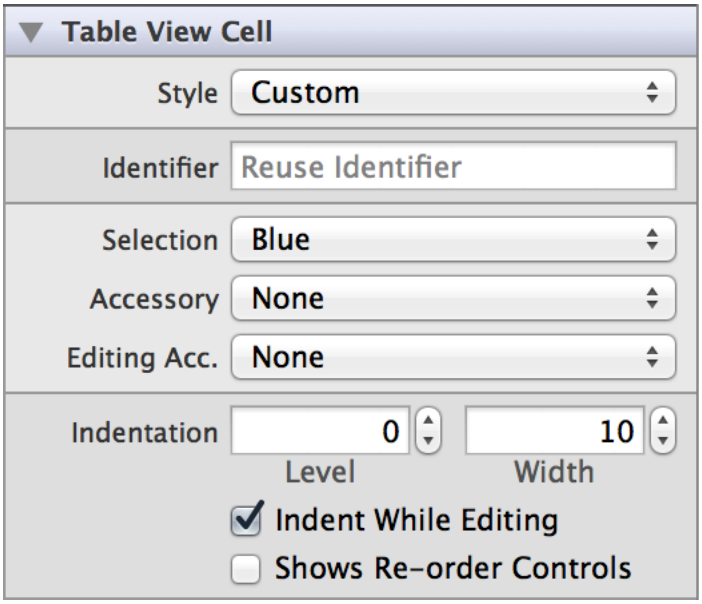
Implementation:

- Table views are implemented in the `UITableView` class and discussed in *UITableView Class Reference*.
- Individual table cells are implemented in the `UITableViewCell` class and discussed in *UITableViewCell Class Reference*.
- Table headers and footers are implemented in the `UITableViewHeaderFooterView` class and discussed in *UITableViewHeaderFooterView Class Reference*.

Configuring Table Views

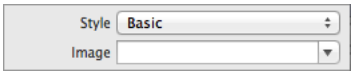
Generally, the easiest way to configure views—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.





Setting Table Content

To display content, a table view must have a data source. The data source mediates between the app's data model and the table view. A table view's data source must conform to the `UITableViewDataSource` protocol. For more information about the data source, see "Data Source and Delegate" in *Table View Programming Guide for iOS*.



Each individual table cell can display a variety of content. Cells that use the default basic style can display an image and text label, and cells that use one of the other three standard styles can display an image, text label, and detail text label in a particular pre-defined layout. You can set a cell's image programmatically using the "Image" (image) field in the Attributes Inspector, which appears when the cell is in one of the four standard styles. However, you must set the `textLabel` and `detailTextLabel` properties programmatically. To learn more about table cell content, see "A Closer Look at Table View Cells".

A cell's content—image, text, and any custom views—resides in its content view. If you want to customize your table cell beyond the standard cell styles, you can set the cell style to custom and add your custom views to the cell's `contentView` property programmatically.

Each table—and each section within that table—can have a header and a footer that displays text or custom content. You use headers and footers to display additional information about the table or its sections. The `UITableViewHeaderFooterView` class implements a reusable view that you can place at the top or bottom of a table or table section.

Headers and footers can either display a text label and optional detail text label, or custom content. You can set the `textLabel` and `detailTextLabel` properties programmatically. Alternatively, you can add your custom views to the header or footer's `contentView` property programmatically. If you are using any custom content in a header or footer, do not use the standard `textLabel` and `detailTextLabel` properties; instead, add your own labels to the content view. For more information about headers and footers, see "Grouped Table Views" in *Table View Programming Guide for iOS* and *UITableViewHeaderFooterView Class Reference*.

Specifying Table View Behavior

Table views need a delegate to manages the appearance and behavior. By assigning a view controller as the table view's delegate and implementing any or all of the `UITableViewDelegate` methods, you can allow the delegate to manage selections, configure section headings and footers, help to delete and reorder cells, and perform other actions.

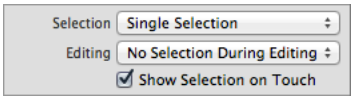
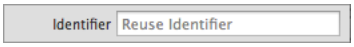
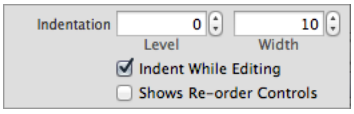


Table selection style controls the number of cells a user can select at a given time. There are three types of selection available for individual cells in a table view: single, multiple, or none. Tables can have different types of selection in normal mode and editing mode. For example, you can allow users to select multiple items in normal mode, but only delete one item at a time in editing mode. In Interface Builder, you can specify selection style for normal mode using the Selection field, and for editing mode using the Editing field. You can also choose whether a cell is visually highlighted upon selection by checking the Show Selection on Touch box.

Index Row Limit (`sectionIndexMinimumDisplayRowCount`) allows you to specify the minimum number of rows required in the table for the index to be shown. Note that this applies to plain style tables only.



A reuse identifier is a string used to identify a cell that can be reused for multiple rows of a table view (for performance purposes). You can set this property using the Identifier (`reuseIdentifier`) field in the Attributes Inspector. You can also set a reuse identifier programmatically during cell initialization.



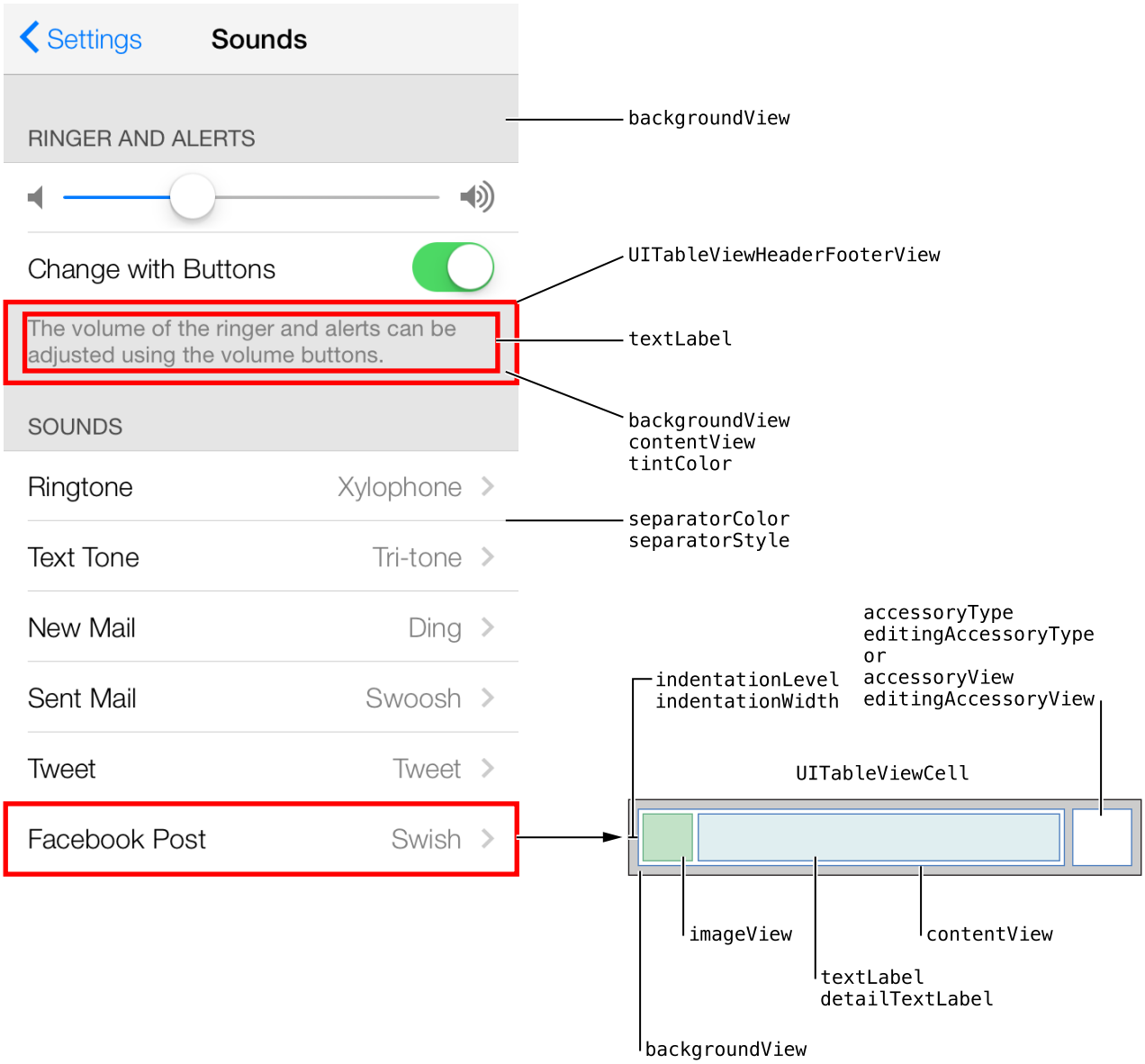
You can set indentation values for cell content through the Level (`indentationLevel`) and Width (`indentationWidth`) fields. The width is the value for each level of indentation. You can indicate whether to indent cell content in editing mode by checking the Indent While Editing (`shouldIndentWhileEditing`) box.

Checking the Shows Re-order Controls (`showsReorderControl`) box will cause the cell to display a control that allows it to be reordered within the table in editing mode. However, you must also implement the `UITableViewDataSource` method `tableView:moveRowAtIndexPath:toIndexPath:` and set `tableView:canMoveRowAtIndexPath:` to return `YES` to get the reordering control to appear in a particular cell. This part must be done programmatically.

To make the table view aware of a header or footer view, you need to register it. You do this using the `registerNib:forCellReuseIdentifier:` or `registerClass:forCellReuseIdentifier:` method. Similar to that of a table cell, the header or footer's reuse identifier is a string used to identify a header or footer view that can be reused for multiple headers or footers within a table. It is set during initialization using the `initWithReuseIdentifier:` method.

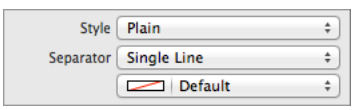
Customizing Table View Appearance

You can customize the appearance of a table view by setting the properties depicted below.



To customize the appearance of all table views in your app, use the appearance proxy (for example, `[UITableView appearance]`). For more information about appearance proxies, see "Using Appearance Proxies" (page 18).

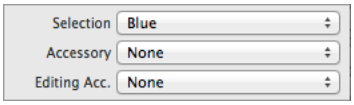
Specifying Table View Style



Separator style dictates how a table's cells are separated visually. You can customize the style and color of table cell separators using the Separator (`separatorStyle`, `separatorColor`) fields in the Attributes Inspector. There are three available styles: single line, etched line, or none.

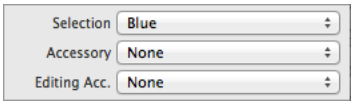
Table style affects the appearance of a table's sections. Tables have two styles: plain and grouped. Certain appearance properties only apply when a table view is displayed in one particular style. You can select the style of a table to be plain or grouped using the Style (`style`) attribute.

Setting Cell Selection Style



Cell selection style specifies which color the outline of a selected cell will appear: gray, blue, or none. Use the Selection (`selectionStyle`) field to set cell selection style.

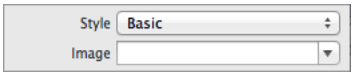
Choosing Accessory Types



You can also set accessory types for normal and editing modes through the Accessory (`accessoryType`) and Editing Acc. (`editingAccessoryType`) fields in the Attributes Inspector. For a list of standard accessory types, see `Cell Accessory Type`.

Alternatively, you can use custom views by setting the `accessoryView` and `editingAccessoryView` properties programmatically. Custom views have precedence over the standard accessory types, so if you set the accessory view properties, your cell ignores the value of the `accessoryType` and `editingAccessoryType` properties.

Specifying Cell Layout



The cell style attribute determines how content is visually laid out in a cell. You can select one of four existing styles—or create a custom one—in the Style field of a table cell's Attributes Inspector.

Specifying Header and Footer Appearance

You can customize the appearance of your header or footer by setting a custom background view or tint color. The background view is placed behind the `contentView` and used to display static background content behind the header or footer. For example, you might assign an image view to this property and use it to display a custom background image. Alternatively, you can set a custom tint for the header or footer view by setting its `tintColor` property. Avoid setting both a custom background view and a custom tint.

Using Auto Layout with Table Views

You can create Auto Layout constraints between a table view and other user interface elements. You can create any type of constraint for a table view besides a baseline constraint.

For general information about using Auto Layout with iOS views, see "Using Auto Layout with Views" (page 20).

Making Table Views Accessible

Table views are accessible by default. Accessibility for tables is handled at the table cell level.

For general information about making iOS views accessible, see "Making Views Accessible" (page 21).

Internationalizing Table Views

To internationalize a table view, you must provide localized strings for text labels and detail text labels of all table cells, headers, and footers.

For more information, see *Internationalization Programming Topics*.

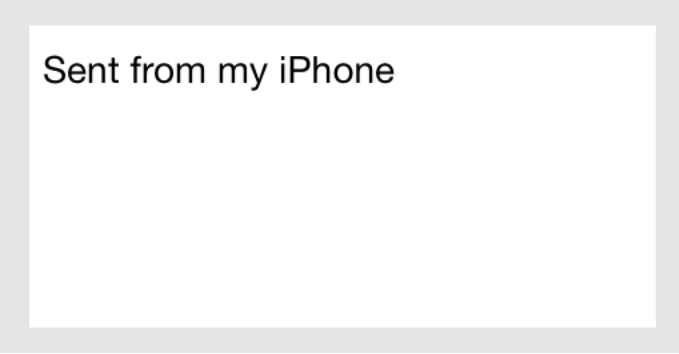
Elements Similar to a Table View

The following element provides similar functionality to a table view:

Scroll View. A class that provides support for displaying content that is larger than the size of the app's window. Use this class when your app contains too much information to display on an iOS device screen at one time. For more information, see "Scroll Views" (page 69).

Text Views

A text view accepts and displays multiple lines of text. Text views support scrolling and text editing. You typically use a text view to display a large amount of text, such as the body of an email message.



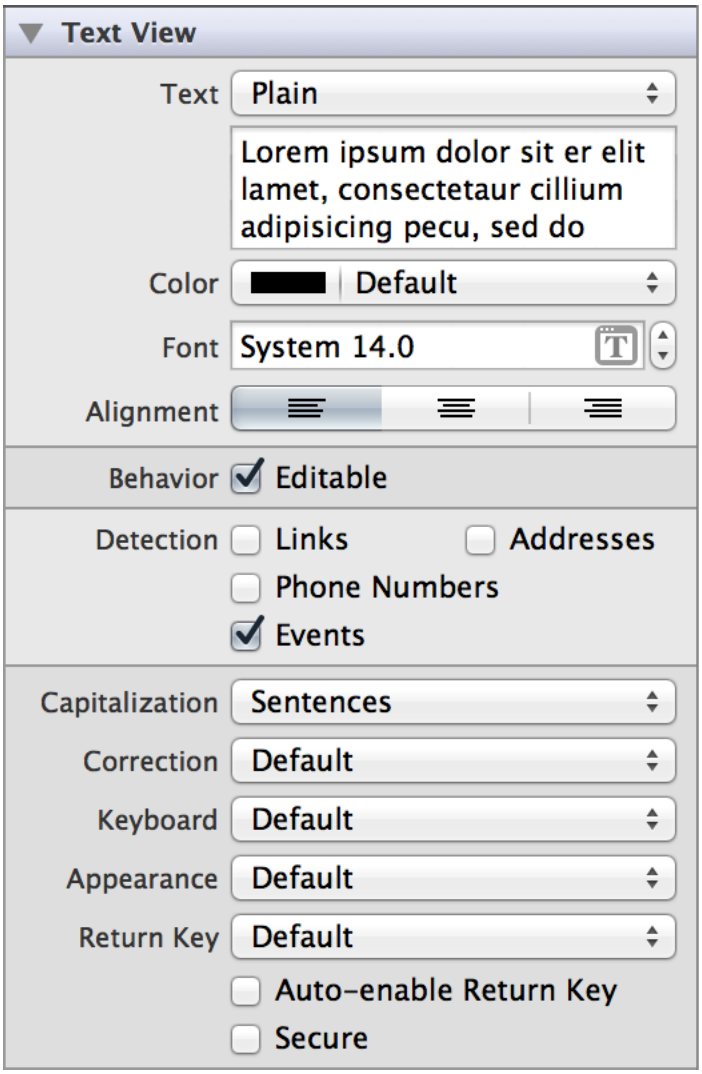
Text views allow the user to:

- Input user content into an app

Implementation: Text views are implemented in the `UITextView` class and discussed in the *UITextView Class Reference* .

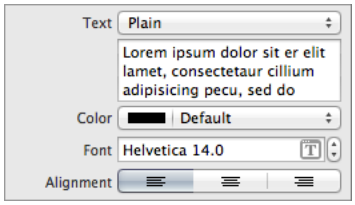
Configuring Text Views

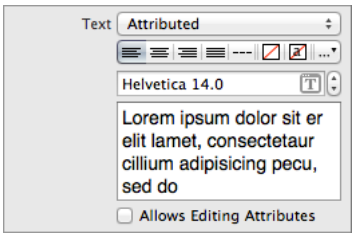
Generally, the easiest way to configure views—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



Setting Text View Content

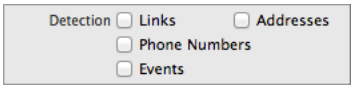
Set text view content using the Text (`text` and `attributedText`) field. Both properties get set whether you specified the value of the field to be plain or attributed. Plain text supports a single set of formatting attributes—font, size, color, and so on—for the entire string. On the other hand, attributed text supports multiple sets of attributes that apply to individual characters or ranges of characters in the string.





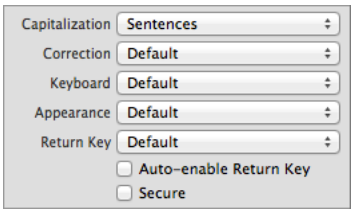
Specifying Text View Behavior

By default, users can add, remove, or change text within a text view. To disable these behaviors, uncheck the Editable (`editable`) checkbox in the Attributes Inspector.



A text view is capable of recognizing when text is formatted as a link, address, phone number, or event. If you enable the corresponding Detection (`dataDetectorTypes`) checkboxes, users will be able to trigger the appropriate action for each type of text by clicking it in the text view. For example, they will be able to call a phone number or add an event to their calendar.

A user types content into a text view using a keyboard, which has a number of customization options:



- **Keyboard layout.** The Keyboard field allows you to select from a number of different keyboard layouts. Match the keyboard layout to the purpose of the text view. If the user will be entering a web address, select the URL keyboard. The default keyboard layout is an alphanumeric keyboard in the device's default language. For a list of possible keyboard types, see `UIKeyboardType`. You cannot customize the appearance of the keyboard on iOS 7.
- **Return key.** The return key, which appears in the bottom right of the keyboard, allows the user to notify the system when they are finished editing the text field. You can select one of several standard return key types by using the Return Key field. The return key is disabled by default, and only becomes enabled when a user types something into the text view. If you want your user to be able to press the return key any

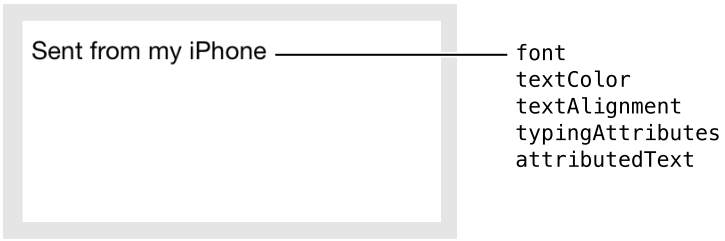
time the keyboard is open, even if the input is empty or incomplete, you can select the Auto-enable Return Key checkbox. Different return keys are intended to provide the user with an understanding of what action hitting the key will trigger. Note that simply selecting a different return key appearance does not provide you with the functionality intended by that key; you must implement the action yourself. Unlike text field delegates, a text view delegate does not provide a method that gets called when the return key is pressed. However, you can implement custom return key functionality in the text view delegate's `textView:shouldChangeTextInRange:replacementText:` method, which gets called after every keystroke.

- **Capitalization scheme.** The Capitalization field specifies how text should be capitalized in the text view: no capitalization, every word, every sentence, or every character. The sentence capitalization scheme is selected by default.
- **Auto-correction.** The Correction field simply disables or enables auto-correct in the text view.
- **Secure content**. The Secure checkbox is unselected by default. Selecting it causes the text view to obscure text once it is typed, allowing the user to safely enter secure content—such as a password—into the view.

You can use the text view delegate methods to handle custom keyboard dismissal.

Customizing Text View Appearance

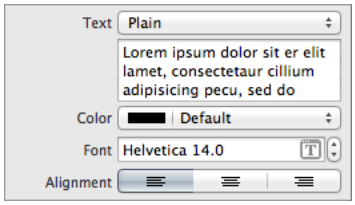
You can customize the appearance of a text view by setting the properties depicted below.

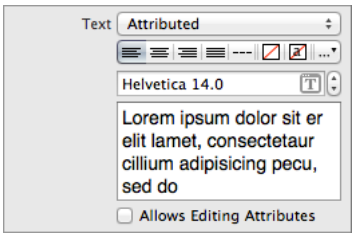


To customize the appearance of all text views in your app, use the appearance proxy (for example, `[UITextView appearance]`). For more information about appearance proxies, see "Using Appearance Proxies" (page 18).

Specifying Text Appearance

Text views can have one of two types of text: plain or attributed. Plain text supports a single set of formatting attributes—font, size, color, and so on—for the entire string. On the other hand, attributed text supports multiple sets of attributes that apply to individual characters or ranges of characters in the string.





Using Auto Layout with Text Views

You can create Auto Layout constraints between a text view and other user interface elements. You can create any type of constraint for a text view besides a baseline constraint.

You generally want the text view to fill the full width of your screen. To ensure that this happens correctly on all devices and orientations, you can create “Leading Space to Superview” and “Trailing Space to Superview” constraints, and set both values equal to 0. This will ensure that the text view remains pinned to the edges of the device screen.

For general information about using Auto Layout with iOS views, see “Using Auto Layout with Views” (page 20).

Making Text Views Accessible

Text views are accessible by default. The default accessibility trait for a text view is User Interaction Enabled.

For general information about making iOS views accessible, see “Making Views Accessible” (page 21).

Internationalizing Text Views

For more information, see *Internationalization Programming Topics*.

Debugging Text Views

When debugging issues with text views, watch for this common pitfall:

Placing a text view inside of a scroll view. Text views handle their own scrolling. You should not embed text view objects in scroll views. If you do so, unexpected behavior can result because touch events for the two objects can be mixed up and wrongly handled.

Elements Similar to a Text View

The following element provides similar functionality to a text view:

Scroll View. Use a scroll view for scrollable content. For more information, see "Scroll Views" (page 69).

Toolbars

A toolbar usually appears at the bottom of a screen, and displays one or more buttons called toolbar items. Generally, these buttons provide some sort of tool that is relevant to the screen's current content. A toolbar is often used in conjunction with a navigation controller, which manages both the navigation bar and the toolbar.

Toolbars allow the user to:

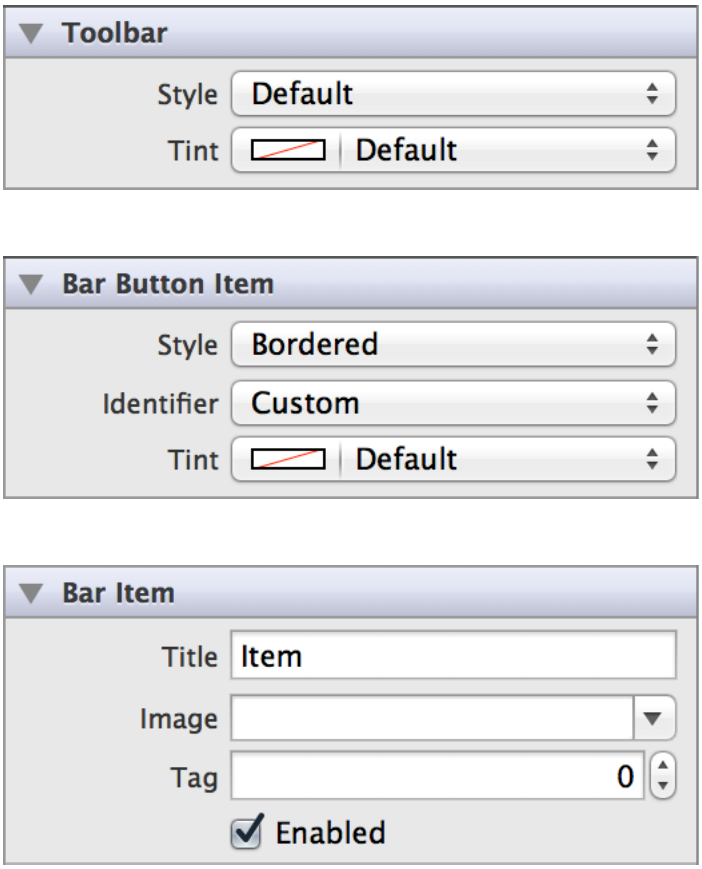
- Select one of a set of performable actions within a given view

Implementation:

- Toolbars are implemented in the `UIToolbar` class and discussed in *UIToolbar Class Reference*.
- Bar button items are implemented in the `UIBarButtonItem` class and discussed in *UIBarButtonItem Class Reference*.
- Bar items are implemented in the `UIBarItem` class and discussed in *UIBarItem Class Reference*.

Configuring Toolbars

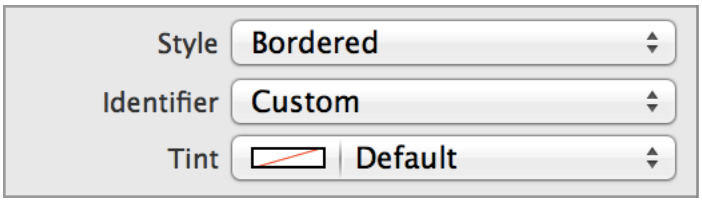
Generally, the easiest way to configure views—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



Setting Toolbar Content

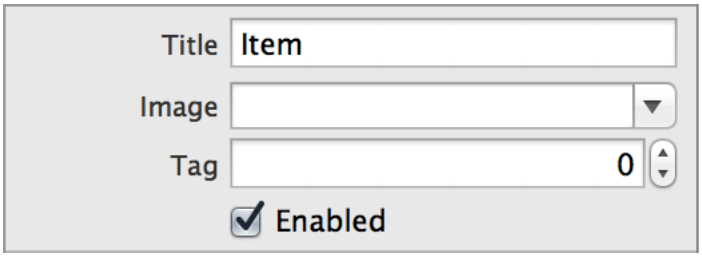
After you create a toolbar, you need to add items to the bar. Each item is a `UIBarButtonItem` object, which you can add to the toolbar directly in Interface Builder or in code using the `items` property. If you want to animate changes to your toolbar items array, use the `setItems:animated:` method.

You can specify the content of a particular bar button item by selecting its identifier. The identifier can either be custom or take on the value of well-know system buttons such as Edit or Done. For a list of system identifiers, see `UIBarButtonSystemItem`.



If you are using a bar button item with the custom identifier, you can set some of its properties at the `UIBarItem` level. For example, you can specify either a custom title or image using the Title (`title`) or Image (`image`) fields.

You can assign a tag to your bar button item using the Tag (`tag`) field. This is intended to be a unique identifier for your button so you can access it in code.



Specifying Toolbar Behavior

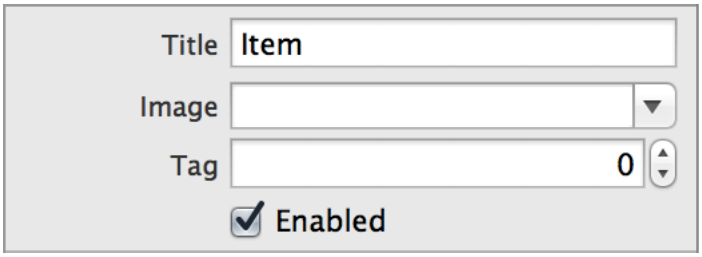
Toolbars do not need a delegate to function properly; their parent view controller can define their behavior without implementing any delegate protocols.

When a user clicks a particular button on the toolbar, you can respond by performing some corresponding action in your app, such as deleting an email. You register the target-action method for a bar button item as shown below.

```
self.myBarButtonItem.target = self;
self.myBarButtonItem.action = @selector(myAction:);
```

Alternatively, you can Control-drag the bar button item's selector from the Connections Inspector to the action method. For more information, see "Understanding the Target-Action Mechanism" (page 118).

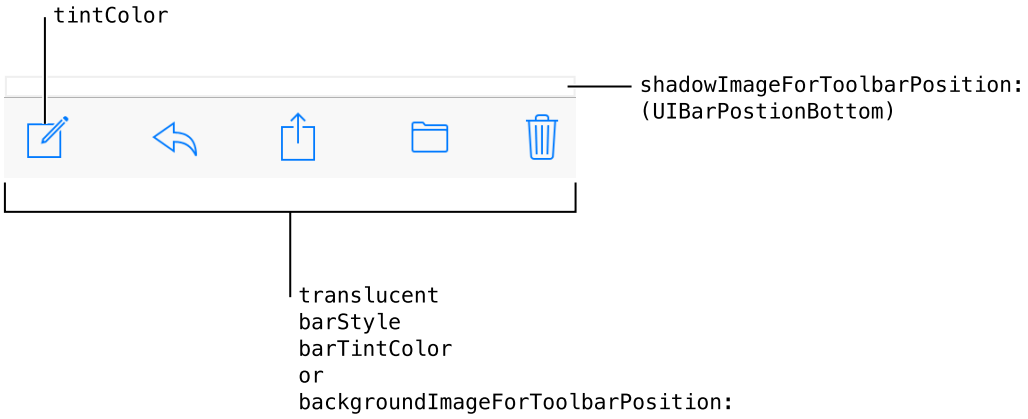
You can disable or enable a given button on the toolbar by selecting the button in Interface Builder and toggling its Enabled (`enabled`) box.



A common way to create and manage a toolbar is in conjunction with a navigation controller. The navigation controller displays the toolbar and populates it with items from the currently visible view controller. Using a navigation controller is ideal for an app design where you want to change the contents of the toolbar dynamically. However, you should not use a navigation controller if your app does not have or need a navigation bar. For more information, see "Displaying a Navigation Toolbar" in *View Controller Catalog for iOS*.

Customizing Toolbar Appearance

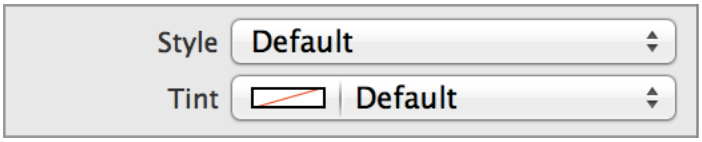
You can customize the appearance of a toolbar by setting the properties depicted below.



To customize the appearance of all toolbars in your app, use the appearance proxy (for example, `[UIToolbar appearance]`). For more information about appearance proxies, see "Using Appearance Proxies" (page 18).

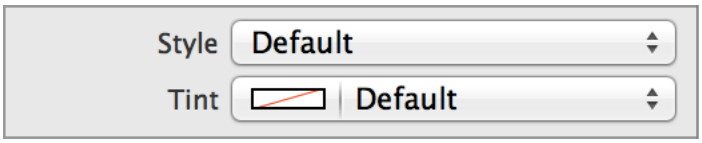
Specifying Toolbar Style

Toolbar have two standard appearance styles: translucent white with dark text (default) or translucent black with light text. Use the Style (`barStyle`) field to select one of these standard styles.



Adjusting Toolbar Tint

You can specify a custom tint color for the bar background using the Tint (`barTintColor`) field. The default background tint color is white.



Additionally, you can set a custom tint color for the interactive elements within a toolbar bar—including the button images and text—programmatically using the `tintColor` property. The toolbar bar will inherit its superview's tint color if a custom one is set, or show the default system blue color if none is set. For more information, see "Adjusting View Tint Color" (page 19).

Setting Toolbar Background Images

You can set a custom background image for your toolbar using the `backgroundImageForToolbarPosition:barMetrics:` method. The image must be the correct dimensions in order to cover the area of the toolbar correctly. Remember to set custom images for different sets of bar metrics.

If you want to use custom shadow image for the toolbar, use the `setShadowImage:forToolbarPosition:` method. To show a custom shadow image, you must also set a custom background image with `backgroundImageForToolbarPosition:barMetrics:`.

Setting Toolbar Translucency

Toolbars are translucent by default on iOS 7. Additionally, there is a system blur applied to all toolbars. This allows your content to show through underneath the bar.

These settings automatically apply when you set any style for `barStyle` or any custom color for `barTintColor`. If you prefer, you can make the toolbar opaque by setting the `translucent` property to `NO` programmatically. In this case, the bar draws an opaque background using black if the toolbar has `UIBarStyleBlack` style, white if the toolbar has `UIBarStyleDefault`, or the toolbar's `barTintColor` if a custom value is defined.

If the toolbar has a custom background image, the default translucency is automatically inferred from the average alpha values of the image. If the average alpha is less than 1.0, the toolbar will be translucent by default; if the average alpha is 1.0, the toolbar will be opaque by default. If you set the `translucent` property to `YES` on a toolbar with an opaque custom background image, the toolbar makes the image translucent. If you set the `translucent` property to `NO` on a toolbar with a translucent custom background image, the toolbar provides an opaque background for the image using black if the toolbar has `UIBarStyleBlack` style, white if the toolbar has `UIBarStyleDefault`, or the toolbar's `barTintColor` if a custom value is defined.

Setting Bar Button Item Glyphs

Any bar button in a toolbar can have a custom image instead of text. You can provide this image to your bar button item during initialization. Note that a bar button image will be automatically rendered as a template image within a toolbar, unless you explicitly set its rendering mode to `UIImageRenderingModeAlwaysOriginal`. For more information, see "Using Template Images" (page 19).

Using Auto Layout with Toolbars

You can create Auto Layout constraints between a toolbar and other user interface elements. You can create any type of constraint for a toolbar besides a baseline constraint.

You cannot create Auto Layout constraints for individual bar button items. However, you can use bar button items with the Fixed Space and Flexible Space identifiers to determine the spacing of buttons on your toolbar.

For general information about using Auto Layout with iOS views, see "Using Auto Layout with Views" (page 20).

Making Toolbars Accessible

Toolbars are accessible by default.

For general information about making iOS views accessible, see "Making Views Accessible" (page 21).

Internationalizing Toolbars

To internationalize a toolbar, you must provide localized strings for all button titles. Remember to test all localizations, as button size may change unexpectedly when using localized strings.

For more information, see *Internationalization Programming Topics*.

Debugging Toolbars

When debugging issues with toolbars, watch for this common pitfall:

Trying to customize the content of a non-custom bar button item. If you try to set a custom title or image—at the `UIBarItem` level—for a bar button item with a non-custom identifier, the bar button item's identifier will automatically switch to the custom type in Interface Builder.

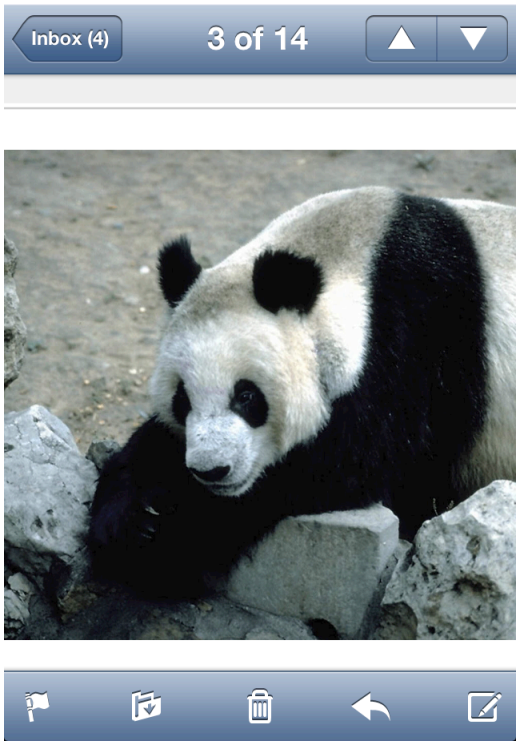
Elements Similar to a Toolbar

The following elements provide similar functionality to a toolbar:

- **Tab Bar.** A toolbar is most similar to a tab bar—both can appear at the bottom of the screen. Use a toolbar to display controls that perform specific functions, and use a tab bar to allow the user to switch between different views or subtasks. For more information, see "Tab Bars" (page 83).
- **Navigation Bar.** For more information, see "Navigation Bars" (page 55).

Web Views

A web view is a region that can display rich HTML content (shown here between the navigation bar and toolbar in Mail on iPhone).



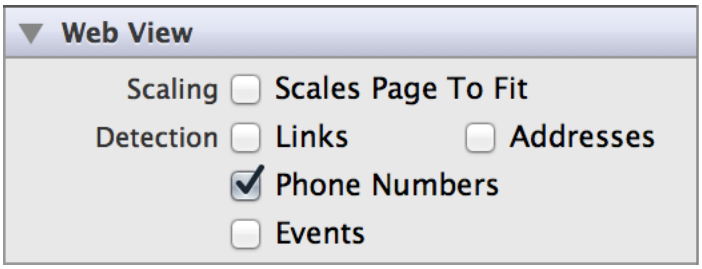
Web views allow the user to:

- View web content within an app

Implementation: Web views are implemented in the `UIWebView` class and discussed in the *UIWebView Class Reference* .

Configuring Web Views

Generally, the easiest way to configure views—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



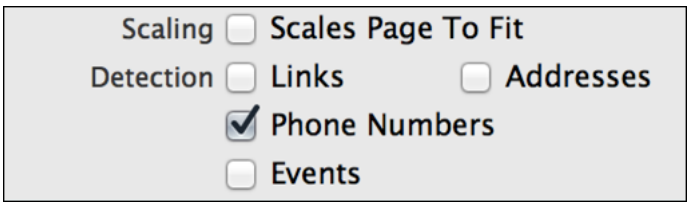
Setting Web View Content Programmatically

To get your web view to display content, you simply create a `UIWebView` object, attach it to a window, and send it a request to load web content. Use the `loadRequest:` method to begin loading web content, the `stopLoading` method to stop loading, and the `loading` property to find out if a web view is in the process of loading. You can create the web view object in code or in Interface Builder, but you can load content in code only.

```
[self.myWebView loadRequest:[NSURLRequest requestWithURL:[NSURL
URLWithString:@"http://www.apple.com/"]]];
```

Specifying Web View Behavior

You can set your web view to automatically scale web content to fit on the screen of the user's device. By default, this behavior is disabled, but you can enable it by checking the Scales Page To Fit (`scalesPageToFit`) box in Attributes Inspector. Enabling this property also allows the user to zoom in and out in the web view.



A web view is capable of recognizing when web text is formatted as a link, address, phone number, or event. If you enable the corresponding Detection (`dataDetectorTypes`) checkboxes, users will be able to trigger the appropriate action for each type of text by clicking it in the web view. For example, they will be able to call a phone number or add an event to their calendar.

Customizing Web View Appearance

You cannot customize the appearance of a web view.

Using Auto Layout with Web Views

You can create Auto Layout constraints between a web view and other user interface elements. You can create any type of constraint for a web view besides a baseline constraint.

You generally want the web view to fill the full width of your screen. To ensure that this happens correctly on all devices and orientations, you can create “Leading Space to Superview” and “Trailing Space to Superview” constraints, and set both values equal to 0. This will ensure that the web view remains pinned to the edges of the device screen.

For general information about using Auto Layout with iOS views, see “Using Auto Layout with Views” (page 20).

Making Web Views Accessible

Web views are accessible by default. The default accessibility trait for a web view is "User Interaction Enabled.”

For general information about making iOS views accessible, see “Making Views Accessible” (page 21).

Internationalizing Web Views

Web views have no special properties related to internationalization.

For more information, see *Internationalization Programming Topics*.

Debugging Web Views

When debugging issues with web views, watch for these common pitfalls:

- **Placing a web view inside of a scroll view.** Web views handle their own scrolling. You should not embed web view objects in scroll views. If you do so, unexpected behavior can result because touch events for the two objects can be mixed up and wrongly handled.
- **Not testing web content size.** Web content comes in a variety of sizes, and it may be difficult to view content that is too large or too small for a device screen. Enable the `scalesPageToFit` property to allow users to zoom in or out if you anticipate that this might be the case for your app.
- **Not having a valid Internet connection.** Since web views rely entirely on the Internet, a working connection is essential to loading web view content. Slow or disabled connections may make it appear as if your web view is not functioning properly when the actual problem is with the connection.

Elements Similar to a Web View

The following element provides similar functionality to a web view:

Scroll View. Use a scroll view for scrollable content. For more information, see "Scroll Views" (page 69).

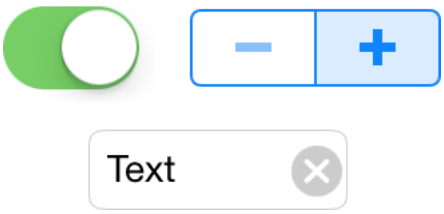
Controls

About Controls

Important: This is a preliminary document for an API or technology in development. Although this document has been reviewed for technical accuracy, it is not final. This Apple confidential information is for use only by registered members of the applicable Apple Developer program. Apple is supplying this confidential information to help you plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future seeds of the API or technology.

A control is a communication tool between a user and an app. The user interacts with a control to convey a particular action or intention to the app. Controls can be used to manipulate content, provide user input, navigate within an app, or execute other pre-defined actions.

Controls are simple, straightforward, and familiar to users because the appear throughout many iOS apps. The `UIControl` class is the base class for all controls on iOS, and defines the functionality that is common to all controls. You should never use it directly; instead, use one of its subclasses. Each subclass of `UIControl` defines appearance, behavior, and intended usage specific to that particular control. By using controls carefully and consistently in your app, you can convey to users what they have the freedom and ability to do within the app.



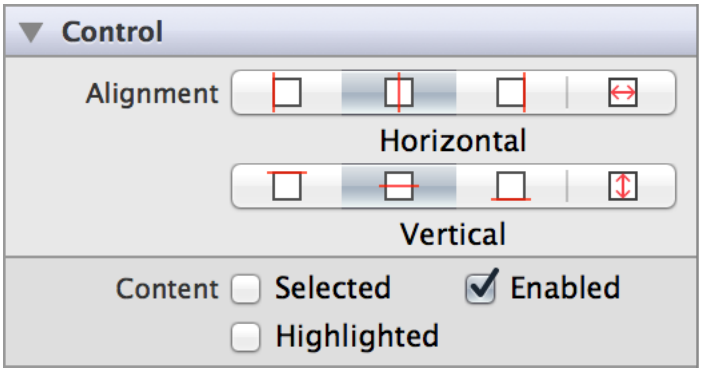
Controls allow the user to:

- Interact with an app.
- Manipulate or edit app content.
- Convey user intent to the app in a straightforward way.

Implementation: Controls are implemented in the `UIControl` class and discussed in *UIControl Class Reference* .

Configuring Controls

Generally, the easiest way to configure controls—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. Each field in the Attributes Inspector that corresponds to a particular API property lists it in parentheses. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



Setting Control Content

Each subclass of `UIControl` has different content or values that you can set. To learn about setting content for a particular control, read its corresponding chapter:

- Buttons
- Date Pickers
- Page Controls
- Segmented Controls
- Text Fields
- Sliders
- Steppers
- Switches

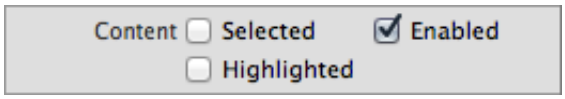
Specifying Control Behavior

Understanding Control States

A **control state** describes the current interactive state of a control: normal, selected, enabled, or highlighted. A control can have more than one state at a time, and you can change a control's state at any point. For a full listing of control states, see `UIControlState`.

When a user interacts with a control, the control's state changes appropriately. You can configure controls to have different appearances for different states to provide users with feedback about which state the control is in. For example, you might configure a button to display one image when it is in its normal state and a different image when it is highlighted.

The fastest way to configure the initial state of a control is by using the Attributes Inspector:



When a control is enabled, a user can interact with it. When a control is disabled, it appears grayed out and does not respond to user interaction. Controls are enabled by default; to disable a control, uncheck the "Enabled" (`enabled`) box in the Attributes Inspector.

A control enters a temporary highlighted state when when a touch enters and exits during tracking and when there is a touch up event. A highlighted state is temporary. You can customize the highlighted appearance of some controls, such as buttons. Controls are not highlighted by default; to set a control's initial state to highlighted, check the "Highlighted" (`highlighted`) box in the Attributes Inspector.

When a user taps on a control, the control enters the selected state. For many controls, this state has no effect on behavior or appearance. However, some subclasses may have different appearance depending on their selected state. For example, `UISegmentedControl` segments have a distinctly different appearance when selected. You can set a control to be selected using the "Selected" (selected) checkbox.

Understanding Control Events

A **control event** represents various physical gestures that users can make on controls, such as lifting a finger from a control, dragging a finger into a control, and touching down within a text field. For a full listing of control events, see `UIControlEvents`.

Understanding the Target-Action Mechanism

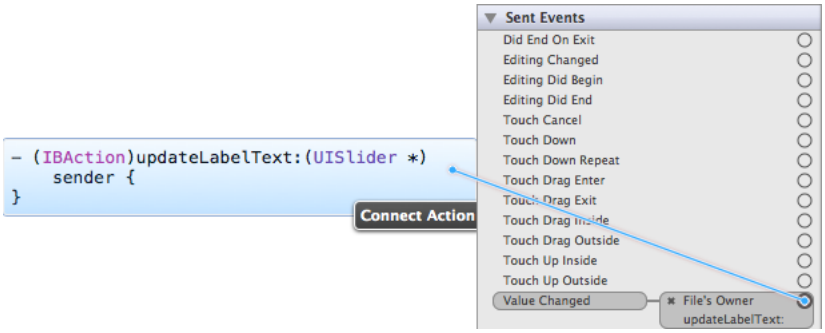
The **target-action mechanism** is a model for configuring a control to send an action message to a target object after a specific control event. For example, when a user interacts with a slider, it generates a `UIControlEventValueChanged` control event. You could use this event to update a label's text to the current value of the slider. In this case, the sender is the slider, the control event is Value Changed, the action is updating the label's text, and the target is the controller file containing the label as an `IBOutlet`.

To create a relationship between the slider, the control event, the target, and the action, you can do one of two things:

1. Call the `addTarget:action:forControlEvents:` method within your target file:

```
[self.mySlider addTarget:self
                  action:@selector(myAction:)
        forControlEvents:UIControlEventValueChanged];
```

2. Use the Connections Inspector in Interface Builder to Control-drag the slider's Value Changed event to the action method in the target file.



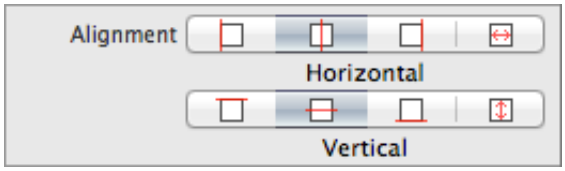
3. Control-click the slider in Interface Builder, and drag its Value Changed event to the target object in your Storyboard. Select the appropriate action from the list of actions available for the target.

For more information, see "Target-Action in UIKit" in *Cocoa Fundamentals Guide*

Customizing Control Appearance

Adjusting Control Content Alignment

Certain controls—such as buttons and text fields—can contain custom images or text. For these controls, you can specify the alignment of that content by using the "Horizontal Alignment" (`contentHorizontalAlignment`) and "Vertical Alignment" (`contentVerticalAlignment`) options in Attributes Inspector. Using the horizontal alignment options, you can specify whether the content appears aligned with the left, center, or right of the control, or whether it fills the width of the control. Using the vertical alignment options, you can specify whether the content appears aligned with the top, center, or bottom of the control, or whether it fills the height of the control. This is a great tool for ensuring your content appears exactly where you want it to within your control (for example, centering text in a text field).



Note: These alignment options apply to the content of a control, not the control itself. For information about aligning controls with respect to other controls or views, see "Using Auto Layout with Controls" (page 120).

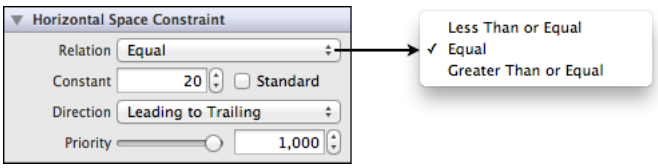
Using Auto Layout with Controls

The auto layout system allows you to define layout constraints for user interface elements, such as views and controls. Constraints represent relationships between user interface elements. You can create auto layout constraints by selecting the appropriate element or group of elements and selecting an option from the menu in the bottom right corner of Xcode's Interface Builder.

Auto layout contains two menus of constraints: pin and align. The Pin menu allows you to specify constraints that define some relationship based on a particular value or range of values. Some apply to the control itself (width) while others define relationships between elements (horizontal spacing). The following tables describes what each group of constraints in the auto layout menu accomplishes:

Constraint Name	Purpose
Width Height	Sets the width or height of a single element.
Horizontal Spacing Vertical Spacing	Sets the horizontal or vertical spacing between exactly two elements.
Leading Space to Superview Trailing Space to Superview Top Space to Superview Bottom Space to Superview	Sets the spacing from one or more elements to the leading, trailing, top, or bottom of their container view. Leading and trailing are the same as left and right in English, but the UI flips when localized in a right-to-left environment.
Widths Equally Heights Equally	Sets the widths or heights of two or more elements equal to each other.
Left Edges Right Edges Top Edges Bottom Edges	Aligns the left, right, top, or bottom edges of two or more elements.
Horizontal Centers Vertical Centers Baselines	Aligns two or more elements according to their horizontal centers, vertical centers, or bottom baselines. Note that baselines are different from bottom edges. These values may not be defined for certain elements.
Horizontal Center in Container Vertical Center in Container	Aligns the horizontal or vertical centers of one or more elements with the horizontal or vertical center of their container view.

The “Constant” value specified for any Pin constraints (besides Widths/Heights Equally) can be part of a “Relation.” That is, you can specify whether you want the value of that constraint to be equal to, less than or equal to, or greater than or equal to the value.



For more information, see *Cocoa Auto Layout Guide*.

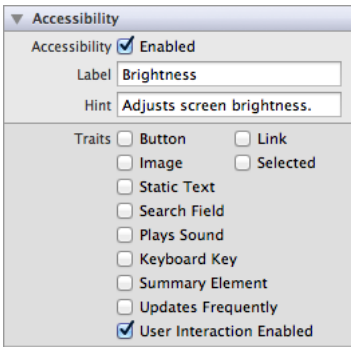
Making Controls Accessible

Controls are accessible by default. To be useful, an accessible user interface element must provide accurate and helpful information about its screen position, name, behavior, value, and type. This is the information VoiceOver speaks to users. Visually impaired users can rely on VoiceOver to help them use their devices.

The iOS SDK contains a programming interface and tools that help you ensure that the user interface elements in your application are both accessible and useful. The UI Accessibility programming interface defines the following attributes:

- **Label.** A short, localized word or phrase that succinctly describes the control or view, but does not identify the element’s type. Examples are “Add” or “Play.”
- **Traits.** A combination of one or more individual traits, each of which describes a single aspect of an element’s state, behavior, or usage. For example, an element that behaves like a keyboard key and that is currently selected can be characterized by the combination of the Keyboard Key and Selected traits.
- **Hint.** A brief, localized phrase that describes the results of an action on an element. Examples are “Adds a title” or “Opens the shopping list.”
- **Frame.** The frame of the element in screen coordinates, which is given by the CGRect structure that specifies an element’s screen location and size.
- **Value.** The current value of an element, when the value is not represented by the label. For example, the label for a slider might be “Speed,” but its current value might be “50%.”

Controls automatically provide value, frame, and default trait information. You can set a label, hint, and adjust the list of traits using the **Identity Inspector** in Interface Builder.



For more information, see *Accessibility Programming Guide for iOS*.

Buttons

Buttons let a user initiate behavior with a tap. You communicate a button's function through a textual label or with an image. Your app changes button appearance based upon user touch interactions, using highlighting, changes in the label or image, color, and state to indicate the button action dynamically.

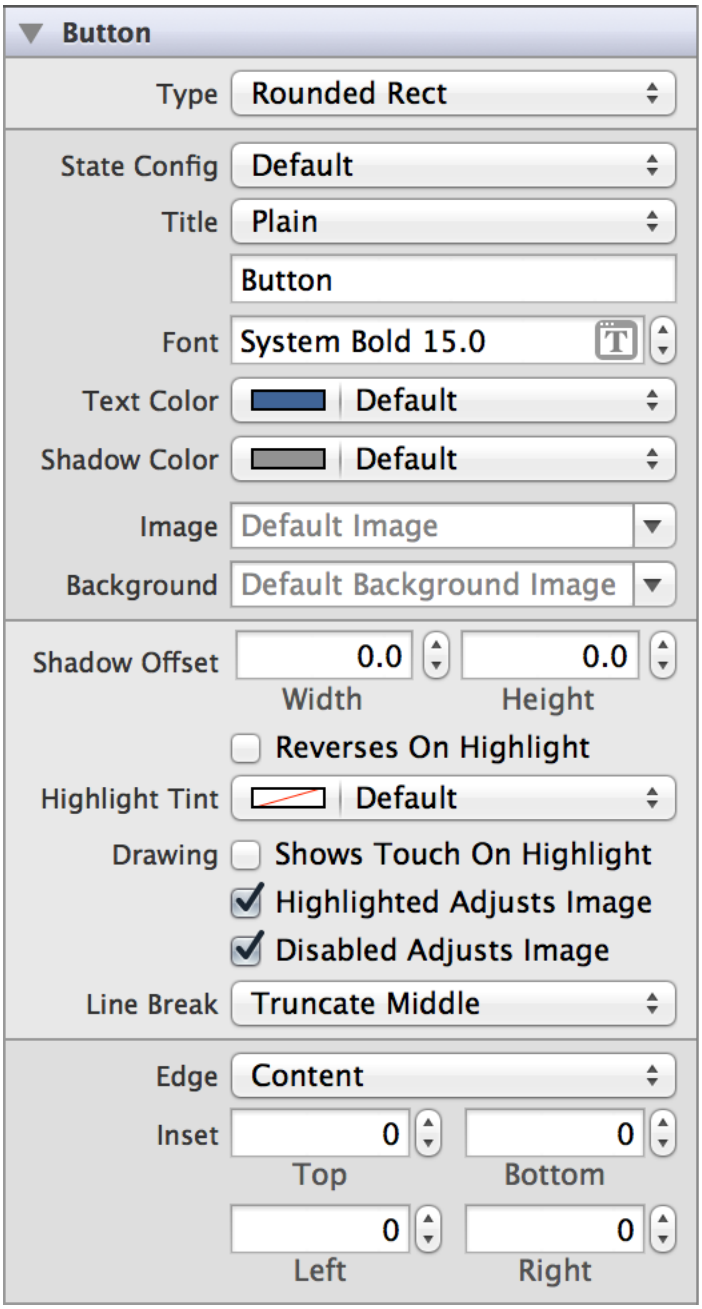
Buttons allow the user to:

- Initiate behavior with a tap
- Initiate an action in the app with a single simple gesture

Implementation: Buttons are implemented in the `UIButton` class and discussed in the *UIButton Class Reference* .

Configuring Buttons

Generally, the easiest way to configure controls—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



Setting Button Content

You can set a button's content using the Type (`buttonType`) field in the Attributes Inspector. In iOS 7, the rounded rect button type has been deprecated in favor of the system button, `UIButtonTypeSystem`. Button objects can be specified as one of five standard types—system, detail disclosure, info light, info dark, and add contact. The detail disclosure, info, and add contact button types are supplied with standard image graphics to indicate their purpose to the user. These images cannot be customized.

There is also a custom type for great versatility in defining a unique interface.

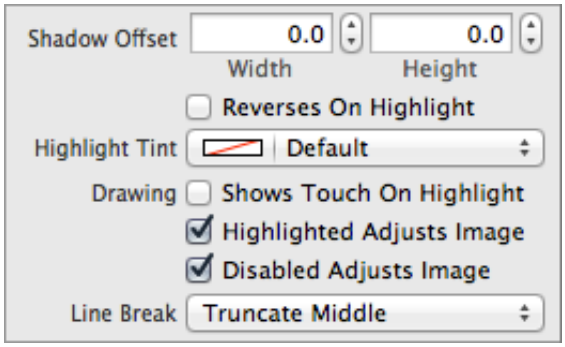
Specifying Button Behavior

Buttons do not need a delegate to function properly; a view controller can define their behavior and functionality without implementing any delegate protocols.

A button sends the `UIControlEventTouchUpInside` event when the user taps it. You can respond to this event by performing some corresponding action in your app, such as saving information. You register the target-action methods for a button as shown below.

```
[self.myButton addTarget:self
                  action:@selector(myAction:)
        forControlEvents:UIControlEventValueChanged];
```

Alternatively, you can Control-drag the button's Value Changed event from the Connections Inspector to the action method. For more information, see "Understanding the Target-Action Mechanism" (page 118).



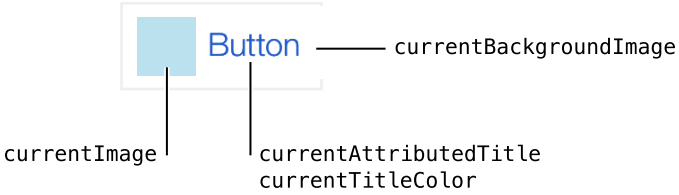
If the Shows Touch On Highlight (`showsTouchWhenHighlighted`) box is enabled, when a user presses on the button, there will be a white glow where the touch event occurred on the button.

If your button has a custom image, the Highlighted Adjusts Image (`adjustsImageWhenHighlighted`) and Disabled Adjusts Image (`adjustsImageWhenDisabled`) options allow you to specify whether highlighted or disabled states affect the appearance of the image. For example, with those options enabled, the image might get darker when the button is highlighted, and dimmer when the button is disabled.

If your button content extends past the bounds of the button, you can specify which part of the content to truncate using the Line Break (`lineBreakMode`) field.

Customizing Button Appearance

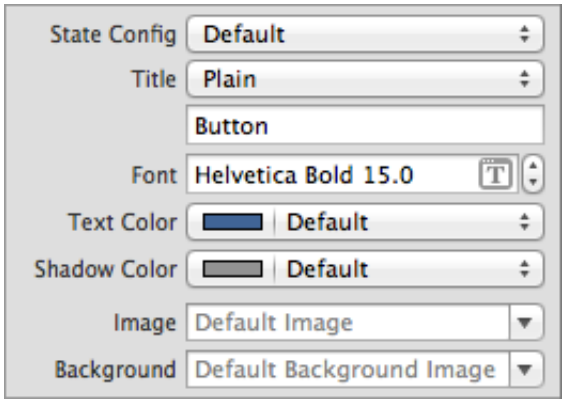
You can customize the appearance of a button by setting the properties depicted below.



To customize the appearance of all buttons in your app, use the appearance proxy (for example, `[UIButton appearance]`). For more information about appearance proxies, see "Using Appearance Proxies" (page 18).

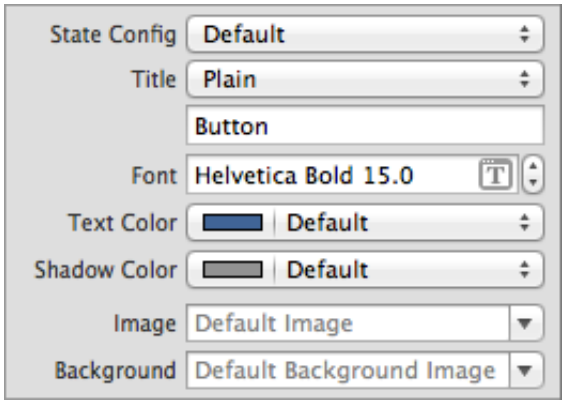
Setting the Button State

A button has four states to configure for appearance—default, highlighted, selected, and disabled. To configure the button's appearance for each state, first select the state from the State Config menu in the Attributes Inspector and then use the other menus and text boxes in the Attributes Inspector's appearance property group.

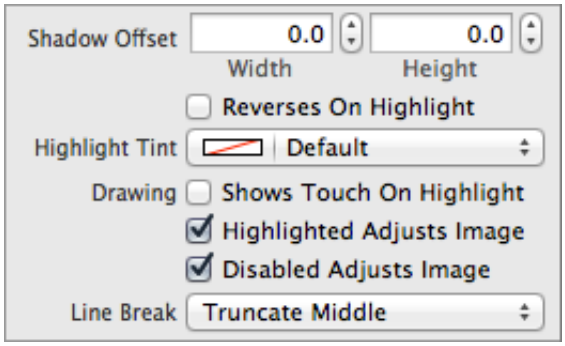


Setting Button Shadow

Shadow offset defines how far the shadow is drawn from the button text. You can customize the offset for both dimensions using the Shadow Offset (`titleShadowOffset`) fields.



You can select the Reverses On Highlight (`reversesTitleShadowWhenHighlighted`) checkbox if you want your shadow offset to automatically flip directions when the button is in the `UIControlStateHighlighted` state.



Note: These shadow properties only have an effect on buttons with plain—not attributed—text.

Adjusting Button Tint

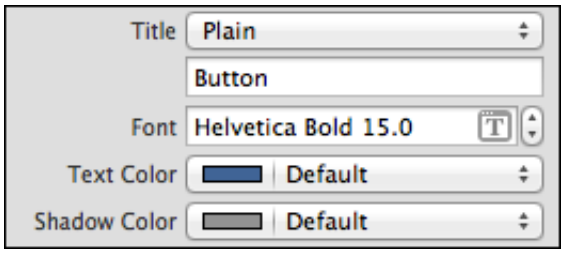
You can specify a custom button tint using the `tintColor` property. This property sets the color of the button image and text.

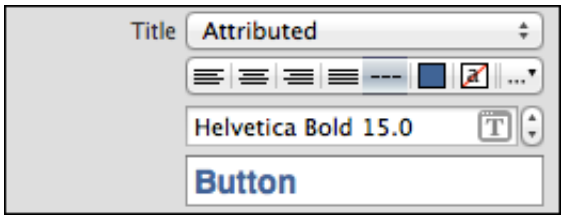
If you do not explicitly set a tint color, the button will inherit its superview's tint color. For more information, see "Adjusting View Tint Color" (page 19).

Formatting Button Title

Button can have one of two types of text: plain or attributed. Plain text supports a single set of formatting attributes—font, size, color, and so on—for the entire string. On the other hand, attributed text supports multiple sets of attributes that apply to individual characters or ranges of characters in the string.

The default title is "Button", intended to be altered to the app need. The title string set in the default state is used in all other states unless you enter a replacement title string for a specific state. Available title customization options differ depending on whether you are using plain or attributed text:



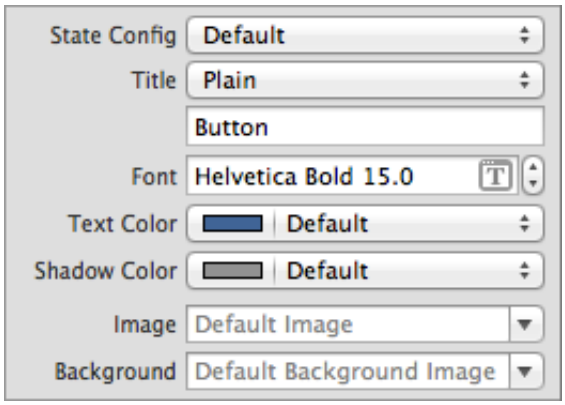


Setting Button Images

Using the Image (`currentImage`) field, you can specify an image to appear within the content of your button. If the button has a title, this image appears to the left of it, and centered otherwise. The image does not stretch or condense, so make sure to select an image that is the proper size to appear in your button. Note that this image will be automatically rendered as a template image within the button, unless you explicitly set its rendering mode to `UIImageRenderingModeAlwaysOriginal`. For more information, see "Using Template Images" (page 19).

The Background (`currentBackgroundImage`) field allows you to specify an image to appear behind button content and fill the entire frame of the button. The image you specify will stretch to fill the button if it is too small. It will be cropped if it is too large.

The images set in the default state are used in all other states unless you enter a replacement image for a specific state.



Adjusting Edge Insets

The part of your button that makes up your image and text is your content. You can offset this content by using edge insets. In the Edge field, select whether you want to offset just your title, just your image, or both together. Depending on your selection, the changes you make to the Inset fields will adjust the `titleEdgeInsets`, `imageEdgeInsets`, and `contentEdgeInsets` properties, respectively.

There should be no reason for you to adjust the edge insets for info, contact, or disclosure buttons. This functionality is intended for custom or rounded rectangle buttons only.

Using Auto Layout with Buttons

You can create Auto Layout constraints between a button and other user interface elements. You can create any type of constraint for a button.

For general information about using Auto Layout with iOS controls, see "Using Auto Layout with Controls" (page 120).

Making Buttons Accessible

Buttons are accessible by default. The default accessibility traits for a button are Button and User Interaction Enabled.

The accessibility label, traits, and hint are spoken back to the user when VoiceOver is enabled on a device. The button's title overwrites its accessibility label; even if you set a custom value for the label, VoiceOver speaks the value of the title. VoiceOver speaks this information when a user taps the button once. For example, when a user taps the Options button in Camera, VoiceOver speaks the following:

```
"Options. Button. Shows additional camera options."
```

For general information about making iOS controls accessible, see "Making Controls Accessible" (page 121).

Internationalizing Buttons

To internationalize a button, you much provide localized strings for its title text.

For more information, see *Internationalization Programming Topics*.

Debugging Buttons

When debugging issues with buttons, watch for these common pitfalls:

- **Setting images that are the wrong dimensions.** A button does not scale or stretch its content images, but it does does not scale or stretch any custom images that you add to it. For example, if you specify an on image that is smaller than the switch, you will see the switch's on tint color in the space that's not covered by the image. On the other hand, if you specify an on image that is too big, it can bleed over into the space intended for the off image. The size of on/off images should be 77 points wide and 27 points tall.
- A common button development problem is when a button is tapped and nothing responds, or the app crashes, when all the code seems well formed and in place. This is commonly caused by an error in linking the button to its action message, often the result of having changed the action message name or the layout of buttons during user interface or code development.

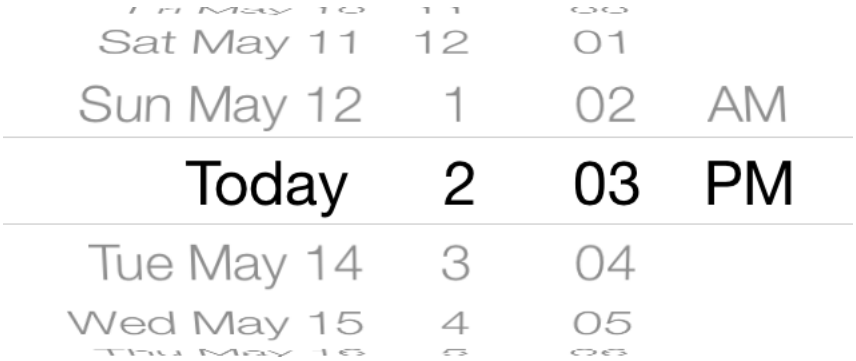
Elements Similar to a Button

The following element provides similar functionality to a button:

Bar Button. An icon used to execute an action from a toolbar or for navigation in a navigation bar. For more information, see "Toolbars" (page 104).

Date Pickers

A date picker is a control used for selecting a specific date, time, or both. It also provides an interface for a countdown timer, although it does not implement the functionality. Date pickers provide a straightforward interface for managing date and time selection, allowing users to specify a particular date quickly and efficiently.



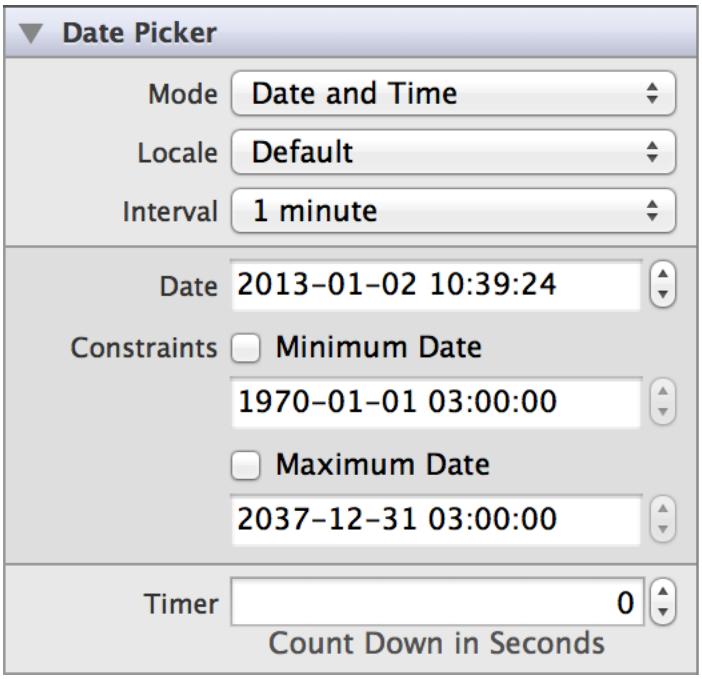
Date pickers allow the user to:

- Specify a particular date and/or time
- Use a countdown timer interface

Implementation: Date pickers are implemented in the `UIDatePicker` class and discussed in the *UIDatePicker Class Reference* .

Configuring Date Pickers

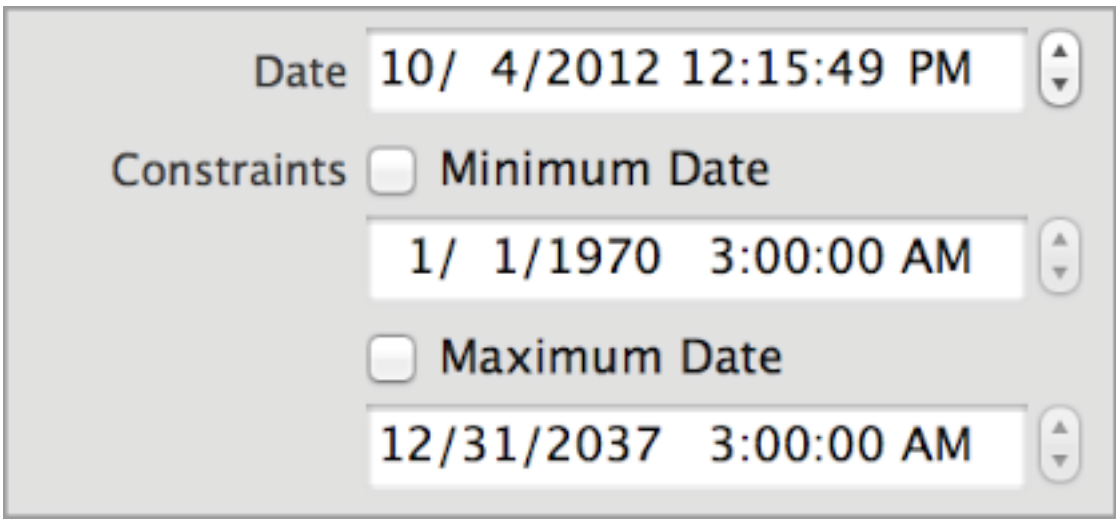
Generally, the easiest way to configure controls—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



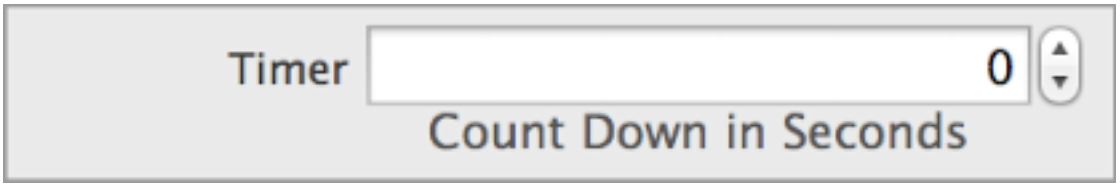
Setting Date Picker Values

A date picker's currently selected time displays at the center of the picker. This value defaults to the time the picker object was created, but you can adjust this value using the Date (`date`) field in Attributes Inspector. Use the Minimum Date (`minimumDate`) and Maximum Date (`maximumDate`) fields to constrain the date picker's

range. For example, if you are asking the user to input a birthday, you might set the maximum date to the current year. Creating a date scope that aligns with your picker's intended functionality simplifies the user's task of finding and setting the correct date.



When a date picker is in the countdown mode, you can use the Timer (`countDownDuration`) field to specify the seconds from which the countdown timer should count down. This value is ignored if the date picker is not in `UIDatePickerModeCountDownTimer` mode. Note that even though the timer shows a countdown in seconds, a user can only specify minute intervals to count down from.



Note: A date picker object presents the countdown timer but does not implement it; the application must set up an `NSTimer` object and update the seconds as they're counted down.

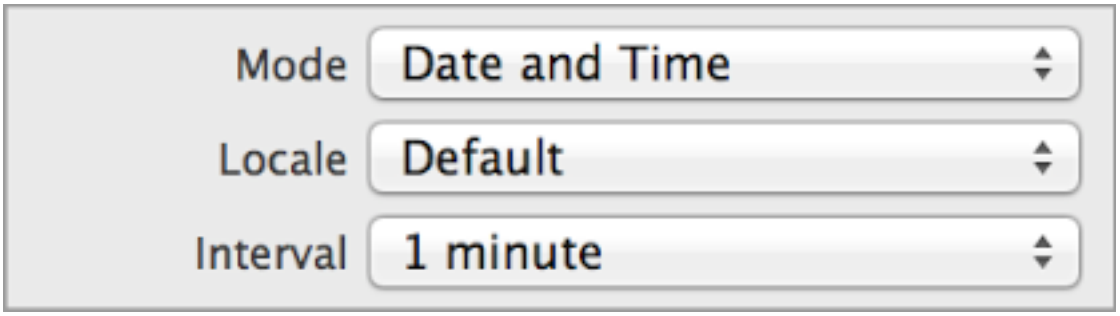
Specifying Date Picker Behavior

Date pickers do not need a delegate to function properly; their parent view controller can define their behavior without implementing any delegate protocols.

A date picker sends the `UIControlEventValueChanged` event when the user finishes rotating one of the wheels to change the date or time. You can respond to this event by performing some corresponding action in your app, such as updating the time for a calendar event. You register the target-action methods for a date picker as shown below.

```
[self.myDatePicker addTarget:self
                      action:@selector(myAction:)
            forControlEvents:UIControlEventValueChanged];
```

Alternatively, you can Control-drag the date picker's Value Changed event from the Connections Inspector to the action method. For more information, see "Understanding the Target-Action Mechanism" (page 118).



The most important setting in determining a date picker's functionality is its mode. A date picker's mode determines what content it displays to the user, as well as how it behaves. There are four mode settings: date and time, date only, time only, or countdown timer. The date and/or time modes allow users to select a specific point in time. The countdown timer allows users to specify a relative time period until an event occurs. You can specify one of these options using the Mode (`datePickerMode`) field in Attributes Inspector.

You can choose a specific locale for your date picker to appear in by adjusting the "Locale" (`locale`) field. For more information, see "Internationalizing Date Pickers" (page 135).

You can also specify the interval at which the date picker displays minutes. A smaller interval gives users more precise control over selecting a date picker time. Choose an interval in the Interval (`minuteInterval`) field.

Customizing Date Picker Appearance

You cannot customize the appearance of date pickers.

Using Auto Layout with Date Pickers

You can create Auto Layout constraints between a date picker and other user interface elements. You can create any type of constraint for a date picker besides a baseline constraint.

Date pickers usually reside at the bottom of the screen in all device orientations. Select "Bottom Space to Superview" and set the relation equal to 0 for the date picker to pin to the bottom of the screen in all device orientations.

For general information about using Auto Layout with iOS controls, see "Using Auto Layout with Controls" (page 120).

Making Date Pickers Accessible

Date pickers are accessible by default. Each rotating wheel in the date picker is its own accessibility element and has the "Adjustable" (`UIAccessibilityTraitAdjustable`) trait.

The accessibility value, traits, and hint for each picker wheel are spoken back to the user when VoiceOver is enabled on a device. VoiceOver speaks this information when a user taps on a picker wheel. For example, when a user taps the hours column on the Add Alarm page (Clock > Alarm > Add), VoiceOver speaks the following:

```
"2 o'clock. Picker item. Adjustable. Swipe up or down with one finger to adjust
the value."
```

For general information about making iOS controls accessible, see "Making Controls Accessible" (page 121).

Internationalizing Date Pickers

Date pickers handle their own internationalization; the only thing you need to do it specify the appropriate locale. You can choose a specific locale for your date picker to appear in by setting the "Locale" (`locale`) field in Attributes Inspector. This changes the language that the date picker is presented in, but also the format of the date and time (for example, certain locales present days before month names, or prefer a 24-hour clock over a 12-hour clock). The width of the date picker automatically accommodates for the length of the localization. To use the system language, leave this property to default.

For more information, see *Internationalization Programming Topics* .

Debugging Date Pickers

When debugging issues with date pickers, watch for these common pitfalls:

- **Specifying conflicting date bounds.** Check the bounds of your `minimumDate` and `maximumDate`. If the maximum date is less than the minimum date, both properties are ignored. The minimum and maximum dates are also ignored in the countdown-timer mode (`UIDatePickerModeCountDownTimer`).
- **Selecting an incorrect interval.** Check that the `minuteInterval` can be evenly divided into 60; otherwise, the default value is used (`1`).

Elements Similar to a Date Picker

The following element provides similar functionality to a date picker:

Picker View. A class like the date picker that can be used for selecting things other than date and time. For more information, see "Picker Views" (page 62).

Page Controls

A page control displays a horizontal series of dots, each of which represents a page or screen in an app. Although a page control doesn't manage the display of content pages, you can write code that lets users navigate between pages by tapping the control. You can see examples of page controls in Weather—when you set more than one location—and in the summary, chart, and news view in Stocks (in portrait orientation). Typically, a page control is used with another view—such as a scroll view—that manages the pages and handles scrolling, panning, and zooming of the content. In this scenario, the scroll view usually uses paging mode to display the content, which is divided into separate views or into separate areas of one view.

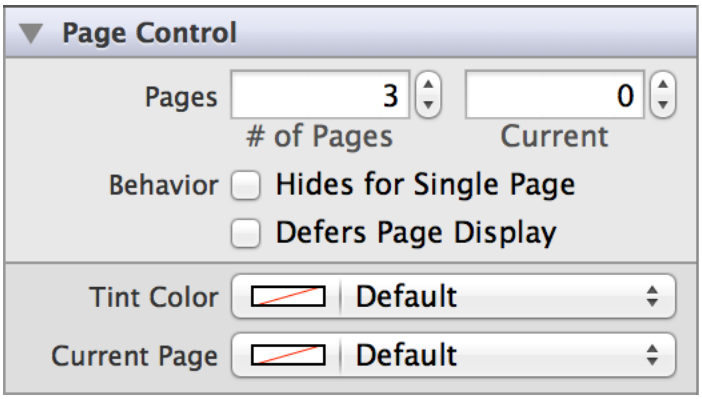
Page controls allow the user to:

- Have a visual indication of which page is currently displayed
- Navigate between pages in an app

Implementation: Page controls are implemented in the `UIPageControl` class and discussed in the *UIPageControl Class Reference*.

Configuring Page Controls

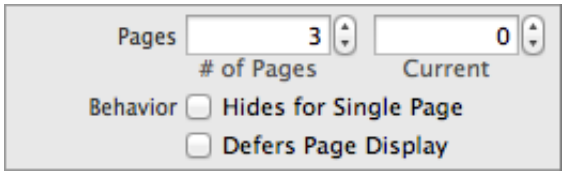
Generally, the easiest way to configure controls—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



Specifying Page Control Values

To specify the number of dots the page control should display, you can use the the # of Pages (`numberOfPages`) field in the Attributes Inspector. Note that when you create a page control in Interface Builder, the default number of dots is 3; when you create a page control programmatically, the default number of dots is 0.

Use the Current (`currentPage`) field to specify the currently selected page. Note that page index begins at 0 instead of 1, so the maximum index of the currently selected page is one less than the total number of pages.



Specifying Page Control Behavior

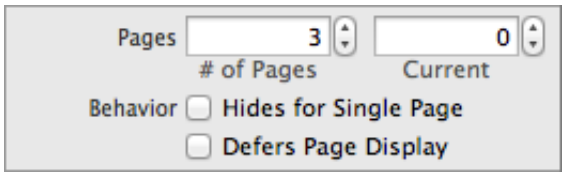
Page controls need a delegate to handle user interaction. A page control doesn't automatically stay synchronized with the currently open page—or let users tap the control to transition between pages—unless you enable these actions in your app. To ensure that a page control's current-page dot corresponds to the page that is currently open in the scroll view, implement the `UIScrollViewDelegate` protocol in your view controller. Then, update the page control in the `scrollViewDidScroll:` delegate method and set the page control's `currentPage` property to the current page.

A page control sends the `UIControlEventValueChanged` event when the user taps it. You can respond to this event by performing some corresponding action in your app, such as transitioning to a different page in the scroll view. You register the target-action methods for a page control as shown below.

```
[self.myPageControl addTarget:self
                       action:@selector(myAction:)
             forControlEvents:UIControlEventValueChanged];
```

Alternatively, you can Control-drag the page control's Value Changed event from the Connections Inspector to the action method. For more information, see "Understanding the Target-Action Mechanism" (page 118).

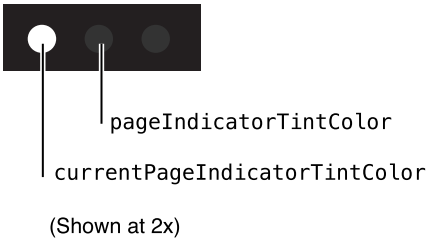
By default, a page control displays one dot when an app contains only one page. To set the page control to display no dots when there is only one page, select the Hides for Single Page (`hidesForSinglePage`) checkbox in the Attributes Inspector.



You can also choose to delay updating the page control's display when the current page changes. If the Defers Page Display (`defersCurrentPageDisplay`) box is enabled, when the user clicks the control to go to a new page, the class defers updating the page control until it calls `updateCurrentPageDisplay`. This behavior is off by default, which means the page indicator updates immediately.

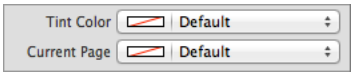
Customizing Page Control Appearance

You can customize the appearance of a page control by setting the properties depicted below.



To customize the appearance of all page controls in your app, use the appearance proxy (for example, `[UIPageControl appearance]`). For more information about appearance proxies, see "Using Appearance Proxies" (page 18).

Adjusting Page Control Tint



The only way to customize the appearance of a page control is by setting custom tints for the dots representing each page. The Current Page (`currentPageIndicatorTintColor`) field affects the color of the dot representing the currently displayed page, and the Tint Color (`pageIndicatorTintColor`) field affects the color of the dots representing every other page. The default color is white for the current page dot, and translucent gray for the other page dots.

If you want your custom colors to be translucent, you must specify a color with an alpha value of less than 1.0. This must be done programmatically, as in the following example:

```
self.myPageControl.currentPageIndicatorTintColor = [UIColor colorWithRed:0.0
green:0.0 blue:1.0 alpha:0.5];

self.myPageControl.pageIndicatorTintColor = [UIColor colorWithRed:1.0 green:0.0
blue:0.0 alpha:0.5];
```

Using Auto Layout with Page Controls

You can create auto layout constraints between a page control and other user interface elements. You can create any type of constraint for a page control besides a baseline constraint.

To keep a page control centered onscreen, you can use auto layout to pin a page control to its superview or align it with other elements. Typically, you leave space for a page control at the bottom of the screen, below the view that displays the pages.

For general information about using auto layout with iOS controls, see "Using Auto Layout with Controls" (page 120).

Making Page Controls Accessible

Page controls are accessible by default. The default accessibility traits for a page control are Updates Frequently and User Interaction Enabled. The Updates Frequently accessibility trait means that the page control doesn't send update notifications when its state changes. This trait tells an assistive app that it should poll for changes in the page control when necessary.

When the user interacts with a page control, VoiceOver speaks "page x of y" where x is the current page and y is the total number of pages.

For general information about making iOS controls accessible, see "Making Controls Accessible" (page 121).

Internationalizing Text Fields

Page controls have no special properties related to internationalization.

For more information, see *Internationalization Programming Topics*.

Debugging Page Controls

When debugging issues with page controls, watch for this common pitfall:

Selecting an out of range current page. If you try to set a page control's current page to be higher than the index of the last page, it will be set equal to the index of the last page. If you try to set a page control's current page to be lower than 0, it will be set to 0.

Elements Similar to a Page Control

The following elements provide similar functionality to a page control:

- **Scroll View.** A class that supports a page-by-page scrolling experience in addition to panning and zooming of content. For more information, see "Scroll Views" (page 69).
- **Page View Controller.** A class that displays multiple content views in a book-like format and enables animated transitions between pages. For more information, see *UIPageViewController Class Reference*.

Segmented Controls

A segmented control is a horizontal control made of multiple segments, each segment functioning as a discrete button.

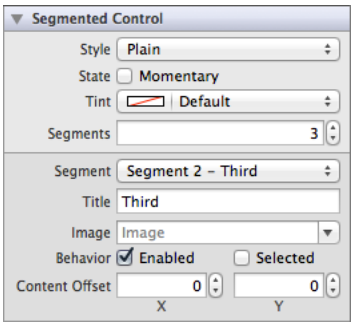
Segmented controls allow the user to:

- Interact with a compact group of a number of controls

Implementation: Segmented controls are implemented in the `UISegmentedControl` class and discussed in the *UISegmentedControl Class Reference*.

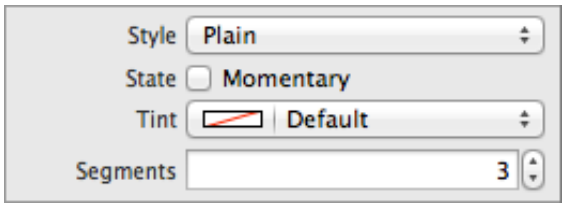
Configuring Segmented Controls

Generally, the easiest way to configure controls—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. As you will read, a few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.

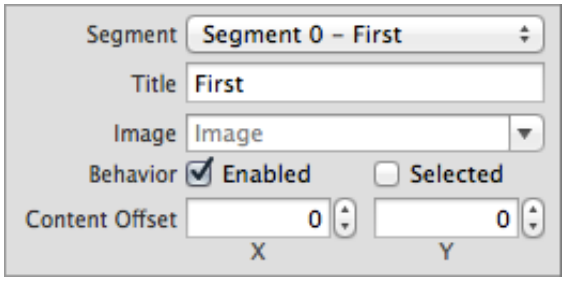


Setting Segmented Control Values

Segmented controls are made up of a number of individual buttons called segments. You can choose the number of segments by setting the Segments (`numberOfSegments`) field. By default, a segmented control is created with two segments.



Content for each segment is set individually. Using the Segment field, you can select a particular segment to modify its content. A segment may either have a text title or an image, but not both. Use the Title (`titleForSegmentAtIndex:`) or Image (`imageForSegmentAtIndex:`) fields to set one of these content properties. As stated in the HI guidelines, avoid creating a segmented control with some segments that contain text and others that images; just choose one or the other.



Specifying Segmented Control Behavior

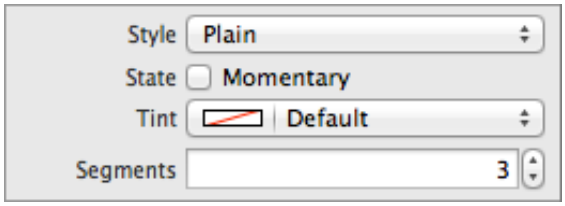
Segmented controls do not need a delegate to function properly; their parent view controller can define their behavior without implementing any delegate protocols.

A segmented control sends the `UIControlEventValueChanged` event when the user presses one of the segments. You can respond to this event by performing some corresponding action in your app, such as switching to a different layout. You register the target-action methods for a segmented control as shown below.

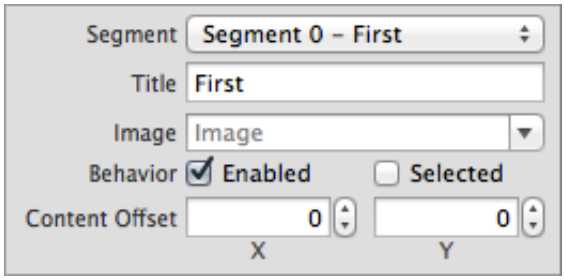
```
[self.mySegmentedControl addTarget:self
                            action:@selector(myAction:)
                  forControlEvents:UIControlEventValueChanged];
```

Alternatively, you can Control-drag the segmented control's Value Changed event from the Connections Inspector to the action method. For more information, see "Understanding the Target-Action Mechanism" (page 118).

If you set a segmented control to have a momentary selection style, its segments do not stay in a selected state when pressed. Instead, they are momentarily highlighted and then restored back to the normal control state. If you would like to enable this behavior, select the Momentary (`momentary`) checkbox in the Attributes Inspector. Note that setting the momentary selection behavior affects every segment in a segmented control; you cannot have a control with some momentary segments and some regular segments.

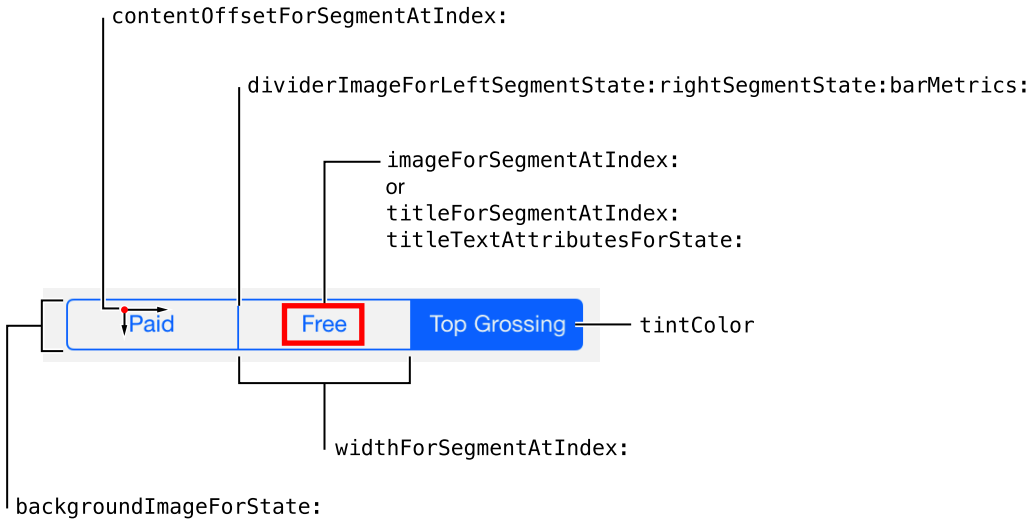


You can specify whether a given segment is enabled or disabled. A user cannot interact with a segment that is disabled. Use the Enabled (`isEnabledForSegmentAtIndex:`) checkbox to specify whether a given segment is enabled for user interaction. Additionally, you can specify whether a particular segment is currently selected using the Selected (`selectedSegmentIndex`) checkbox. Note that only one segment can be selected at a time; if you set the selection for a given segment, the previously selected segment will become unselected.



Customizing Segmented Control Appearance

You can customize the appearance of a segmented control by setting the properties depicted below.

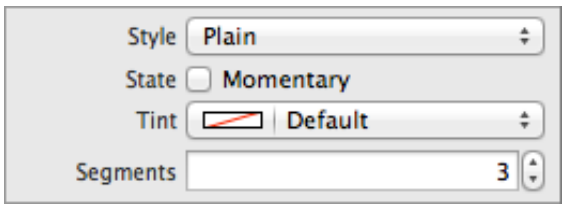


To customize the appearance of all segmented controls in your app, use the appearance proxy (for example, `[UISegmentedControl appearance]`). For more information about appearance proxies, see "Using Appearance Proxies" (page 18).

Adjusting Segmented Control Tint

You can specify a custom segmented control tint using the Tint (`tintColor`) field. This property sets the color of the segment images, text, dividers, borders, and selected segment. A translucent version of this color is also used to tint a segment when it is pressed and transitioning to being selected, as shown on the first segment below.

If you do not explicitly set a tint color, the segmented control will inherit its superview's tint color. For more information, see "Adjusting View Tint Color" (page 19).

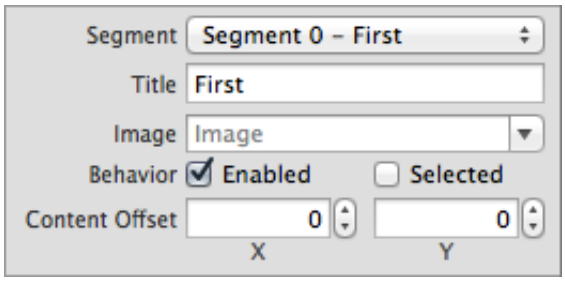


Specifying Segmented Control Style

You cannot customize the segmented control's style on iOS 7. Segmented controls only have one style, and the Style (`segmentedControlStyle`) field has been deprecated.

Specifying Segment Content Offset

If you want the content of a particular segment to be offset from its default position, you can alter it using the Content Offset fields in Attributes Inspector.



Setting Segmented Control Images

If you need to customize the appearance of your segmented control beyond standard tinting, you might consider doing so using custom images. Since segmented controls have different metrics for portrait and landscape device orientations, remember to specify an appropriate image for each set of metrics.

You can set a background image for each control state of your segmented control using the `backgroundImageForState:barMetrics:` method. You should also specify divider images for each combination of left and right segment states to give selected or highlighted segments a different look than segments in a normal state, as shown here:

```
[mySegmentedControl setDividerImage:image1 forLeftSegmentState:UIControlStateNormal
                  rightSegmentState:UIControlStateNormal barMetrics:barMetrics];

[mySegmentedControl setDividerImage:image2 forLeftSegmentState:UIControlStateSelected
                  rightSegmentState:UIControlStateNormal barMetrics:barMetrics];

[mySegmentedControl setDividerImage:image3 forLeftSegmentState:UIControlStateNormal
                rightSegmentState:UIControlStateSelected barMetrics:barMetrics];
```

Formatting Segment Titles

The `titleTextAttributesForState:` property specifies the attributes for displaying the segment's title text. You can specify the font, text color, text shadow color, and text shadow offset for the title in the text attributes dictionary, using the text attribute keys described in *NSString UIKit Additions Reference*.

Setting Segment Glyphs

You can use an image instead of title text for your segments. Note that a segment image will be automatically rendered as a template image within a segmented control, unless you explicitly set its rendering mode to `UIImageRenderingModeAlwaysOriginal`. For more information, see "Using Template Images" (page 19).

Using Auto Layout with Segmented Controls

You can create auto layout constraints between a segmented control and other user interface elements. You can create any type of constraint for a segmented control.

For general information about using auto layout with iOS controls, see "Using Auto Layout with Controls" (page 120).

Making Segmented Controls Accessible

The following listing demonstrates how you can set the accessibility label of programmatically-generated segments.

```
NSString *title = @"∫";
title.accessibilityLabel = @"Integral";
[segmentedControl insertedSegmentedWithTitle:title];


UIImage *image = [UIImage imageNamed:@"GearImage.png"];
image.accessibilityLabel = @"Settings";
[segmentedControl insertedSegmentWithImage:image];
```

For general information about making iOS controls accessible, see "Making Controls Accessible" (page 121).

Internationalizing Segmented Controls

To internationalize a segmented control, you must provide localized strings for all segment titles. Remember to test all localizations, as segment size may change unexpectedly when using localized strings.

For more information, see *Internationalization Programming Topics*.

Debugging Segmented Controls

When debugging issues with segmented controls, watch for these common pitfalls:

- **Specifying conflicting appearance settings.** When customizing segment content with text or images, use one or the other, but not both. A segment cannot have both text and an image as its content. Whichever content property was set last will override the other one.
- **Not setting custom images for every control state.** If you use custom background and divider images for your segmented control, remember to set an image for every possible `UIControlState` combination. Any control state that does not have a corresponding custom image assigned to it will display the standard image instead. If you set one custom image, make sure to set them all.

Elements Similar to a Segmented Control

The following elements provide similar functionality to a segmented control:

- **Tab Bar.** A class used for navigating between views in an app. You should use a tab bar instead of a segmented control when you want to let the user move back and forth between distinct pages in your app. For more information, see "Tab Bars" (page 83).
- **Toolbar.** A class that allows users to perform certain actions in the current context. For more information, see "Toolbars" (page 104).

Sliders

Sliders enable users to interactively modify some adjustable value in an app, such as speaker volume or screen brightness. For example, in GarageBand, sliders allow users to mix different settings for various effects. Users control a slider by moving its current value indicator along a continuous range of values between a specified minimum and maximum.

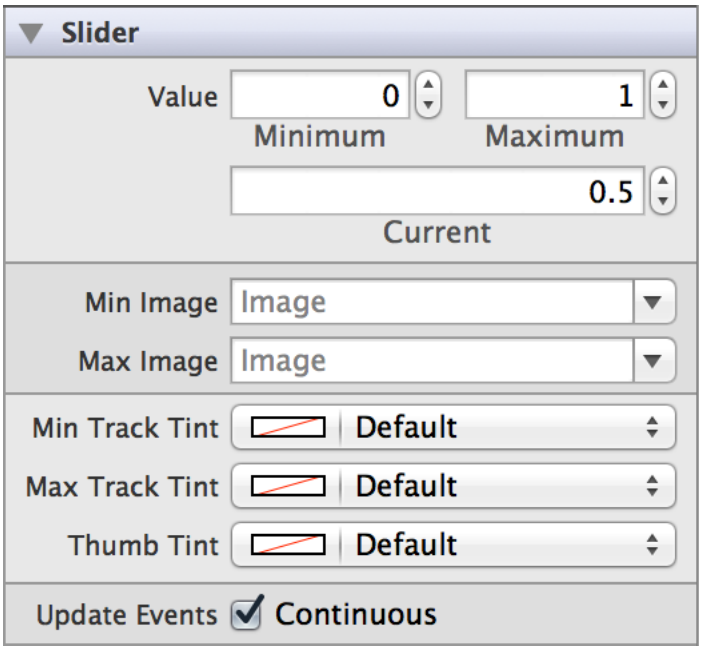
Sliders allow the user to:

- Make smooth and continuous adjustments to a value
- Have relative control over a value within a range
- Set a value with a single simple gesture

Implementation: Sliders are implemented in the `UISlider` class and discussed in the *UISlider Class Reference* .

Configuring Sliders

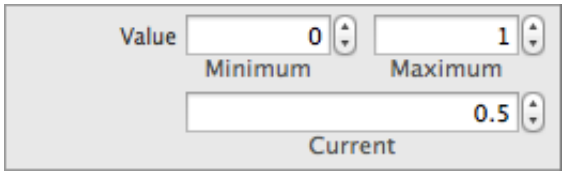
Generally, the easiest way to configure controls—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



You can customize the appearance of all sliders in your app using the appearance proxy (for example, `[UISlider appearance]`), or just of a single control. For more information about appearance proxies, see "Using Appearance Proxies" (page 18).

Setting Slider Values

You can configure a minimum, maximum, and current value for your slider. By default, a slider's minimum is set to 0, its maximum is set to 1, and its current value is set to 0.5. You can change these values by adjusting the Minimum (`minimumValue`), Maximum (`maximumValue`), and Current (`value`) fields in the Attributes Inspector.



Specifying Slider Behavior

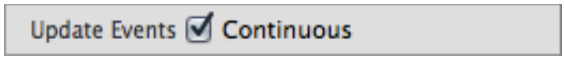
Sliders do not need a delegate to function properly; their parent view controller can define their behavior without implementing any delegate protocols.

A slider sends the `UIControlEventValueChanged` event when the user interacts it. You can respond to this event by performing some corresponding action in your app, such as adjusting music volume. You register the target-action methods for a slider as shown below.

```
[self.mySlider addTarget:self
                  action:@selector(myAction:)
        forControlEvents:UIControlEventValueChanged];
```

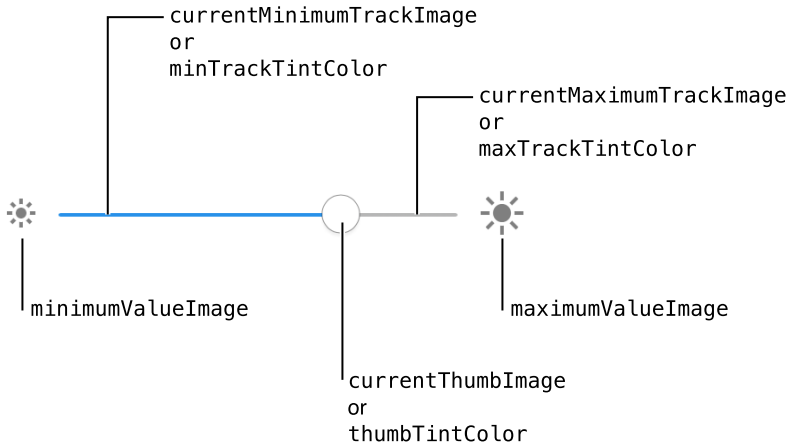
Alternatively, you can Control-drag the slider's Value Changed event from the Connections Inspector to the action method. For more information, see "Understanding the Target-Action Mechanism" (page 118).

You can specify when a slider's `UIControlEventValueChanged` events are sent by toggling the Continuous (`continuous`) checkbox in the Attributes Inspector. In continuous delivery, the slider sends multiple value changed events as the user moves the thumb. In noncontinuous delivery, the slider sends one value changed event when the user releases the thumb. Continuous control event delivery is enabled by default.



Customizing Slider Appearance

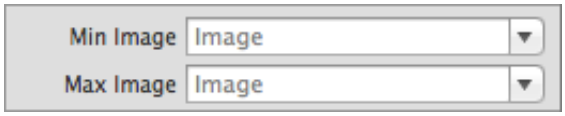
You can customize the appearance of a slider by setting the properties depicted below.



To customize the appearance of all sliders in your app, use the appearance proxy (for example, `[UISlider appearance]`). For more information about appearance proxies, see "Using Appearance Proxies" (page 18).

Setting Minimum and Maximum Value Images

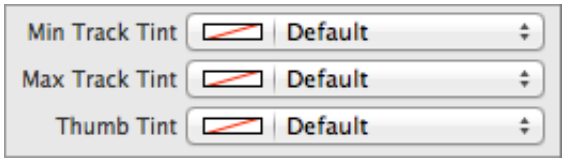
The most common way to customize the slider's appearance is to provide custom minimum and maximum value images. You can set a slider's minimum and maximum value images in Interface Builder by selecting custom images in the Min Image (`minimumValueImage`) and Max Image (`maximumValueImage`) fields in the Attributes Inspector.



These images sit at either end of the slider control and indicate which value that end of the slider represents. For example, a slider used to control volume might display a speaker with no sound waves emanating from it for the minimum value and display a speaker with many sound waves emanating from it for the maximum value, as illustrated here.

Adjusting Slider Tint

You can also set custom tints for each part of a slider. The minimum track image contains a blue highlight by default, while the maximum track and thumb images contain a white highlight. You can assign different tints for all of the standard parts provided by the slider. This can be done in the Attributes Inspector by setting the Min Track Tint (`minimumTrackTintColor`), Max Track Tint (`maximumTrackTintColor`), and Thumb Tint (`thumbTintColor`) fields.



Note that you can only adjust the tint of the default track and thumb images, not custom images. Setting the tint of a part of the slider that has custom images associated with it will remove those images.

Setting Track and Thumb Images Programmatically

Slider controls draw the track using two customizable images. The region between the thumb and the end of the track associated with the slider's minimum value is drawn using the minimum track image. The region between the thumb and the end of the track associated with the slider's maximum value is drawn using the maximum track image. Different track images are used to provide context as to which end contains the minimum

value. You can customize their appearance by assigning different pairs of track images to each `UIControlState` of the slider. Assigning different images to each state lets you customize the appearance of the slider when it is enabled, disabled, highlighted, and so on.

Use stretchable images for the slider's track to create custom track images of any arbitrary width. You can create a stretchable image by using the `UIImage` method `resizableImageWithCapInsets:`. Use this method to add cap insets to an image or to change the existing cap insets of an image. During scaling or resizing of the image, areas covered by a cap are not scaled or resized. Instead, the pixel area not covered by the cap in each direction is tiled, left-to-right and top-to-bottom, to resize the image. For best performance, use a single-pixel image.

You can also customize the appearance of the thumb. Like the track images, you can assign different thumb images to each control state of the slider. To use a custom thumb image, add the image you wish to use to your Xcode project, and use it to create a `UIImage`. Use this `UIImage` to programmatically set the thumb image for your slider, as shown here. The same steps apply for the track images.

```
UIImage *thumbImage = [UIImage imageNamed:@"custom_thumb.png"];
[self.mySlider setThumbImage:thumbImage forState:UIControlStateNormal];
```

Track and thumb images can only be customized programmatically, as there is no process for accomplishing this task in the Attributes Inspector.

Using Auto Layout with Sliders

You can create Auto Layout constraints between a slider and other user interface elements. You can create any type of constraint for a slider besides a baseline constraint.

To keep a slider centered and adjust its width according to device orientation or screen size, you can use Auto Layout to pin it to its superview. Using the Auto Layout "Pin" menu, create "Leading Space to Superview" and "Trailing Space to Superview" constraints and set their values equal to each other. Doing this ensures that the endpoints of your slider are a specified distance from the edges of its superview. With these constraints, the slider stays centered and its width adjusts automatically for different device orientations and screen sizes.

For general information about using Auto Layout with iOS controls, see "Using Auto Layout with Controls" (page 120).

Making Sliders Accessible

Sliders are accessible by default. The default accessibility traits for a slider are User Interaction Enabled and Adjustable.

When enabled on a device, VoiceOver speaks the accessibility label, value, traits, and hint are spoken back to the user. VoiceOver speaks this information when a user swipes up and down (not left and right) over the slider. For example, using the Ringer and Alerts volume slider (Settings > Sounds > Ringer and Alerts), VoiceOver speaks the following:

```
"Sound volume: 13 percent. Adjustable. Swipe up or down with one finger to adjust
 the value."
```

For general information about making iOS controls accessible, see "Making Controls Accessible" (page 121).

Internationalizing Sliders

Sliders have no special properties related to internationalization. However, if you use a slider with a label, make sure you provide localized strings for the label.

For more information, see *Internationalization Programming Topics* .

Debugging Sliders

When debugging issues with sliders, watch for these common pitfalls:

- **Specifying conflicting appearance settings.** When customizing slider appearance with images or tint, use one option or the other, but not both. Conflicting settings for track and thumb appearance will be resolved in favor of the most recently set value. For example, setting a new minimum track image for any state clears any custom tint color you may have provided for minimum track images. Similarly, setting the thumb tint color removes any custom thumb images associated with the slider.
- **Selecting an out of range value.** If you try to programmatically set a slider's current value to be below the minimum or above the maximum, it is set to the minimum or maximum instead. However, if you try to do this using Interface Builder, the behavior is much different. For example, let's say your slider has a minimum value of 0, a maximum value of 5, and the current value is set to 1. If you change the current value to 999, the maximum value for the slider automatically changes to 999 because that value is higher than the maximum value, so Interface Builder adjusts accordingly.

- **Not setting custom images for every control state.** If you use custom track and thumb images for your slider, remember to set an image for every possible `UIControlState`. Any control state that does not have a corresponding custom image assigned to it will display the standard image instead. If you set one custom image, make sure to set them all.

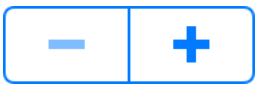
Elements Similar to a Slider

The following elements provide similar functionality to a slider:

- **Switch.** A control that represents an on/off toggle button. You should use a switch instead of a slider when you want to give users a choice between two opposing, discrete options instead of a range of values. For more information, see "Switches" (page 161).
- **Stepper.** A control that uses a set of two buttons for incrementing or decrementing a value. You should use a stepper instead of a slider when you want to give users very precise control over the value of an element. For more information, see "Steppers" (page 156).

Steppers

A stepper lets the user adjust a value by increasing and decreasing it in small steps. Steppers are used in situations where a user needs to adjust a value by a small amount.



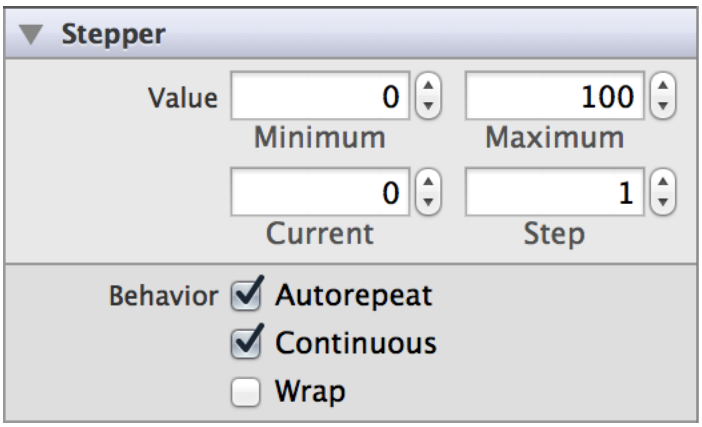
Steppers allow the user to:

- Make discrete and incremental adjustments to a value
- Have precise control over a value within a range

Implementation: Steppers are implemented in the `UIStepper` class. For API reference, see *UIStepper Class Reference*.

Configuring Steppers

Generally, the easiest way to configure controls—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.

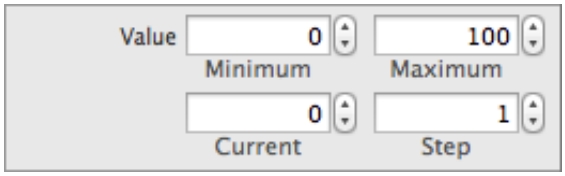


You can customize the appearance of all steppers in your app using the appearance proxy (for example, `[UIStepper appearance]`), or just of a single control. For more information about appearance proxies, see "Using Appearance Proxies" (page 18).

Setting Stepper Values

Configure a minimum, maximum, and current value for the stepper by setting these properties in Interface Builder. By default, a stepper's minimum is set to 0, its maximum is set to 100, and its current value is set to 0. You can change these values by adjusting the Minimum (`minimumValue`), Maximum (`maximumValue`), and Current (`value`) fields.

Steppers also allow you to specify step size, the amount by which the current value changes when the increase or decrease buttons are pressed. The default step size is 1. The corresponding Attributes Inspector field is called Step (`stepValue`).



Specifying Stepper Behavior

Steppers do not need a delegate to function properly; their parent view controller can define their behavior without implementing any delegate protocols.

A stepper sends the `UIControlEventValueChanged` event when the user interacts it. You can respond to this event by performing some corresponding action in your app, such as adjusting music volume. You register the target-action methods for a page control as shown below.

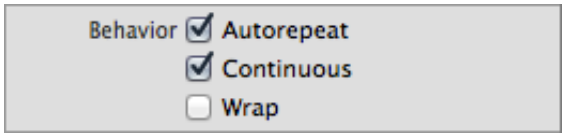
```
[self.myStepper addTarget:self
                   action:@selector(myAction:)
         forControlEvents:UIControlEventValueChanged];
```

Alternatively, you can Control-drag the stepper's Value Changed event from the Connections Inspector to the action method. For more information, see "Understanding the Target-Action Mechanism" (page 118).

You can specify when a stepper's `UIControlEventValueChanged` events are sent by toggling the "Continuous" (`continuous`) checkbox in the Attributes Inspector. In continuous delivery, the stepper sends multiple Value Changed events while the user keeps pressing on the stepper. In noncontinuous delivery, the stepper sends one Value Changed event when the user releases the stepper. Continuous control event delivery is enabled by default.

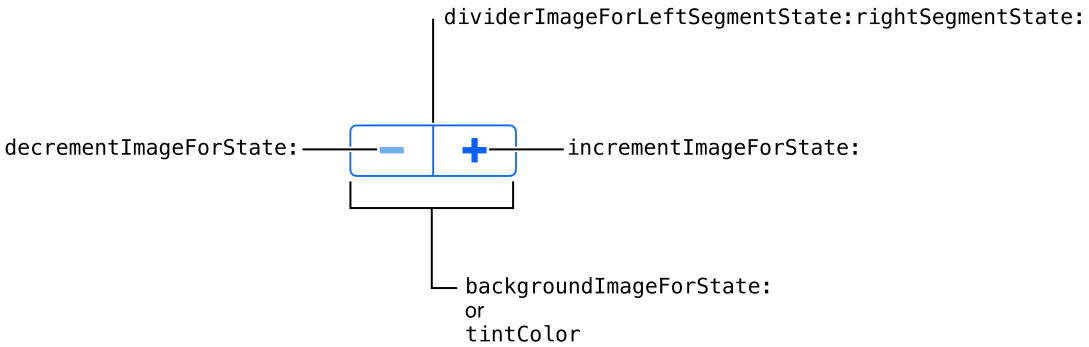
A stepper defaults to autorepeat, which means that pressing and holding one of its buttons increments or decrements the stepper's value repeatedly. The rate of change depends on how long the user continues pressing the control. The user can hold the stepper to quickly approach a desired value, and then increment or decrement to the desired value. Uncheck the "Autorepeat" (`autorepeat`) box if you want the stepper to be incremented or decremented one step at a time.

You can set a stepper to wrap around to the minimum value when you try to increment it past its maximum—and vice versa. This functionality is disabled by default; to enable it, check the "Wrap" (`wraps`) box.



Customizing Stepper Appearance

You can customize the appearance of a stepper by setting the properties depicted below.



To customize the appearance of all steppers in your app, use the appearance proxy (for example, `[UIStepper appearance]`). For more information about appearance proxies, see "Using Appearance Proxies" (page 18).

Adjusting Stepper Tint Programmatically

You can specify a custom stepper tint by setting the `tintColor` property programmatically. This property sets the color of the glyphs, divider, and border of the stepper. A translucent version of this color is also used to tint a stepper button when it is pressed down, as shown on the increment button below.

If you do not explicitly set a tint color, the stepper will inherit its superview's tint color. For more information, see "Adjusting View Tint Color" (page 19).

Setting Stepper Glyphs

The increment and decrement images are the glyphs that sit on top of each stepper button. They appear on top of the background image. Use the `setDecrementImage:forState:` and `setIncrementImage:forState:` methods to specify custom increment and decrement images for each control state. Note that an increment or decrement image will be automatically rendered as a template image within a toolbar, unless you explicitly set its rendering mode to `UIImageRenderingModeAlwaysOriginal`. For more information, see "Using Template Images" (page 19).

Setting Stepper Images

You can set custom images for each `UIControlState` of a stepper. For more information about control states, see "Understanding Control States" (page 117).

A stepper can have a background image that covers the entirety of the control except for the divider, filling the entire frame of the stepper. Use the `backgroundImageForState:` method to set a background image for each control state of the stepper.

To strengthen the visual effect of a stepper button being pressed, you can set custom divider images for different combinations of button states: increment button pressed, decrement button pressed, and neither pressed. Note that it is impossible to press both buttons at the same time. Use the `setDividerImage:forLeftSegmentState:rightSegmentState:` method to specify custom divider images. Don't forget to set an image for every state.

Using Auto Layout with Steppers

You can create Auto Layout constraints between a stepper and other user interface elements. You can create any type of constraint for a stepper besides a baseline constraint.

For general information about using Auto Layout with iOS controls, see "Using Auto Layout with Controls" (page 120).

Making Steppers Accessible

Steppers are accessible by default. The default accessibility traits for a stepper are User Interaction Enabled and Adjustable.

For general information about making iOS controls accessible, see "Making Controls Accessible" (page 121).

Internationalizing Steppers

Steppers have no special properties related to internationalization. However, if you use a stepper with a label, make sure you provide localized strings for the label.

For more information, see *Internationalization Programming Topics*.

Elements Similar to a Stepper

The following elements provide similar functionality to a stepper:

- **Slider.** Use sliders to adjust a value continuously, rather than in discrete steps. Sliders are more appropriate than steppers for setting a value that has a large range. For more information, see "Sliders" (page 149).
- **Picker View.** Use pickers to let the user select one of a list of options, rather than stepping through the range of a value. A picker is more appropriate when selecting from a fixed set of options—for example, choosing a month. For more information, see "Picker Views" (page 62).

Switches

A switch lets the user turn an option on and off. You see switches used throughout the Settings app to let a user quickly toggle a specific setting.

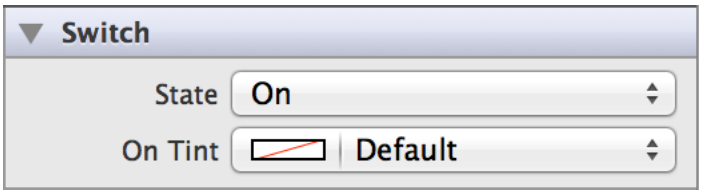
Switches allow the user to:

- Choose between two mutually exclusive options
- Quickly toggle an option on and off

Implementation: Sliders are implemented in the `UISwitch` class and discussed in the UISwitch.

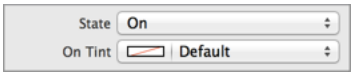
Configuring Switches

Generally, the easiest way to configure controls—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



Setting Switch Value

Specify a switch's state to indicate whether the switch is initially on or off. The default value is on. Use the State (on) field in the Attributes Inspector to accomplish this task.



Specifying Switch Behavior Programmatically

Switches do not need a delegate to function properly; their parent view controller can define their behavior without implementing any delegate protocols.

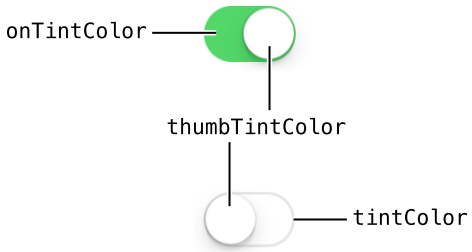
A switch sends the `UIControlEventValueChanged` event when the user toggles it. You can respond to this event by performing some corresponding action in your app, such as turning a setting on or off. You register the target-action methods for a switch as shown below.

```
[mySwitch addTarget:self
             action:@selector(myAction:)
   forControlEvents:UIControlEventValueChanged];
```

Alternatively, you can Control-drag the switch's Value Changed event from the Connections Inspector to the action method. For more information, see "Understanding the Target-Action Mechanism" (page 118).

Customizing Switch Appearance

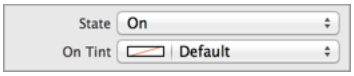
You can customize the appearance of a switch by setting the properties depicted below.



To customize the appearance of all switches in your app, use the appearance proxy (for example, `[UISwitch appearance]`). For more information about appearance proxies, see "Using Appearance Proxies" (page 18).

Adjusting Switch Tint

A switch's on tint can be configured in the On Tint (`onTintColor`) field in Attributes Inspector. This is the color you see when a switch is in the on position. The default on tint is green.



Thumb tint and off tint can only be configured programmatically. By default, the thumb tint is white and can be set using the `thumbTintColor` property . You can also set a custom off tint using the `tintColor` property. The off tint is light gray by default, but will inherit its superview's tint color if a custom one is set. For more information, see "Adjusting View Tint Color" (page 19).

```
self.mySwitch.thumbTintColor = [UIColor blueColor];
self.mySwitch.tintColor = [UIColor redColor];
```

Using Auto Layout with Switches

Switches need a label to tell the user what they are for. To label a switch, drag a label out of the elements library. Make a bottom alignment constraint between their baselines, and a horizontal space constraint between them of standard size.

For general information about using Auto Layout with iOS controls, see "Using Auto Layout with Controls" (page 120).

Making Switches Accessible

Switches are accessible by default. A switch's default accessibility traits are Button and User Interaction Enabled.

Switches are typically used in a table cell. When a table cell with a switch is tapped, VoiceOver speaks the cell name, state of the switch, and hint are spoken back to the user. For example, when a user taps the Invert Colors switch (Settings > General > Accessibility), VoiceOver speaks the following:

```
Invert Colors. Off. Double tap to toggle setting.
```

For general information about making iOS controls accessible, see "Making Controls Accessible" (page 121).

Internationalizing Switches

Switches have no special properties related to internationalization. However, if you use a switch with a label, make sure you provide localized strings for the label.

For more information, see *Internationalization Programming Topics*.

Debugging Switches

When debugging issues with sliders, watch for these common pitfalls:

- **Setting on/off images that are the wrong dimensions.** A switch does not scale or stretch any custom images that you add to it. For example, if you specify an on image that is smaller than the switch, you will see the switch's on tint color in the space that's not covered by the image. On the other hand, if you specify an on image that is too big, it can bleed over into the space intended for the off image. The size of on/off images should be 77 points wide and 27 points tall.
- **Specifying conflicting appearance settings.** When customizing switch appearance with images or tint, you can use one option or the other, but not both. Custom images appear on top of the tint layer. While you may think you are adjusting the tint of the image itself, you're simply setting the tint for a layer that is not visible under the image.

Elements Similar to a Switch

The following element provides similar functionality to a switch:

Slider. A control that allows users to make adjustments to a value within a range of value. You should use a slider instead of a switch when you want to let users select from a range of values instead of giving them a choice between two opposing, discrete options. For more information, see "Sliders" (page 149).

Text Fields

Text fields allows the user to input a single line of text into an app. You typically use text fields to gather small amounts of text from the user and perform some immediate action, such as a search operation, based on that text.

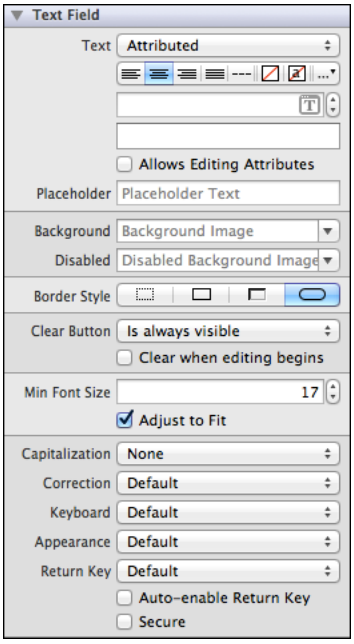
Text fields allow the user to:

- Enter text as input to an app

Implementation: Text fields are implemented in the `UITextField` class and discussed in the *UITextField Class Reference* .

Configuring Text Fields

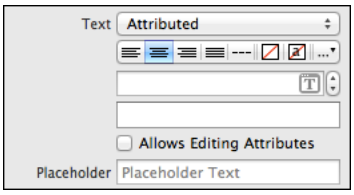
Generally, the easiest way to configure controls—namely, their content, behavior, and appearance—is by using the **Attributes Inspector** in Interface Builder. As an alternative to using the Attributes Inspector, you can set some configurations programmatically. A few configurations cannot be made through the Attributes Inspector, so you must make them programmatically.



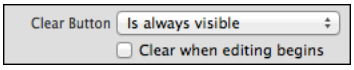
Setting Text Field Content

Set the content of the text field using the Text (`text`) field. You can select whether you want plain or attributed text. The placeholder appears in place whenever a text field has no characters (before a user begins typing, or if the user deletes everything in the text field).

Both placeholder and text can be attributed strings. For information about using attributed text, see "Specifying Text Appearance" (page 168).



A user can use the Clear button to delete all text in the text field, and display the placeholder string if one is set. You can specify when the Clear button is displayed to the user using the Clear Button (`clearButtonMode`) field. Additionally, you can indicate whether the text field should automatically clear itself when the user begins editing it by checking the Clear When Editing Begins (`clearsOnBeginEditing`) box.



> **Note:** The Clear button only appears when there is text shown in the text field, not the placeholder. Even if you select the "Is always visible" option, it will not appear when only placeholder text appears.

Specifying Text Field Behavior

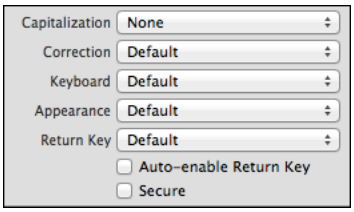
Text fields need a delegate to handle any custom behaviors, such as displaying additional overlay views when a user begins editing it. By assigning the parent view controller as the text field's delegate and implementing any or all of the `UITextFieldDelegate` methods, you can implement such custom behaviors.

A text field sends the `UIControlEventEditingDidBegin`, `UIControlEventEditingChanged`, `UIControlEventEditingDidEnd`, and `UIControlEventEditingDidEndOnExit` events when the user edits it. You can respond to these events by performing some corresponding action in your app, such as updating information as the user types it. You register the target-action methods for a text field as shown below.

```
[self.myTextField addTarget:self
                     action:@selector(myAction:)
           forControlEvents:UIControlEventEditingDidEnd];
```

Alternatively, you can Control-drag the text field's Value Changed event from the Connections Inspector to the action method. For more information, see "Understanding the Target-Action Mechanism" (page 118).

A user types content into a text field using a keyboard, which has a number of customization options:



- **Keyboard layout.** The Keyboard field allows you to select from a number of different keyboard layouts. Match the keyboard layout to the purpose of the text field. If the user will be entering a web address, select the URL keyboard. The default keyboard layout is an alphanumeric keyboard in the device's default language. For a list of possible keyboard types, see `UIKeyboardType`. You cannot customize the appearance of the keyboard on iOS 7.
- **Return key.** The return key, which appears in the bottom right of the keyboard, allows the user to notify the system when they are finished editing the text field. You can select one of several standard return key types by using the Return Key field. The return key is disabled by default, and only becomes enabled when a user types something into the text field. If you want your user to be able to press the return key any time the keyboard is open, even if the input is empty or incomplete, you can enable the Auto-enable Return Key option. Different return keys are intended to provide the user with an understanding of what action hitting the key will trigger. Note that simply selecting a different return key appearance does not provide you with the functionality intended by that key; you must implement custom return key behavior yourself using the `textFieldShouldReturn:` method in your text field's delegate.
- **Capitalization scheme.** The Capitalization field specifies how text should be capitalized in the text field: no capitalization, every word, every sentence, or every character. Although no capitalization is selected by default, you should select the capitalization scheme that reflects the intended use of your text field. For example, if you ask for a user's full name, you can configure the keyboard to capitalize every word so the user does not have to do it manually.
- **Auto-correction.** The Correction field simply disables or enables auto-correct in the text field.
- **Secure content**. The Secure option is off by default. Enabling it causes the text field to obscure text once it is typed, allowing the user to safely enter secure content—such as a password—into the field.

You can use the text field delegate methods to handle custom keyboard dismissal.

Customizing Text Field Appearance

You can customize the appearance of a text field by setting the properties depicted below.

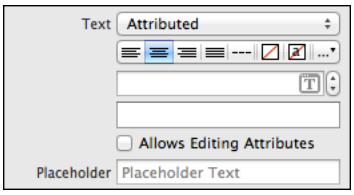
To customize the appearance of all text fields in your app, use the appearance proxy (for example, `[UITextField appearance]`), or just of a single control. For more information about appearance proxies, see "Using Appearance Proxies" (page 18).

Selecting Border Style

You can select one of the following border styles for your text field by selecting it next to the Border Style (`borderStyle`) field:

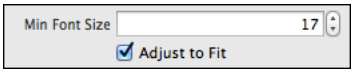
Specifying Text Appearance

Text views can have one of two types of text: plain or attributed. Plain text supports a single set of formatting attributes—font, size, color, and so on—for the entire string. On the other hand, attributed text supports multiple sets of attributes that apply to individual characters or ranges of characters in the string.



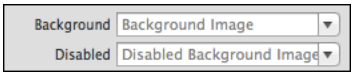
Specifying Font Size

By default, the Adjusts to Fit (`adjustsFontSizeToFitWidth`) box is selected in the Attributes Inspector. When this option is enabled, the font size of your text label will automatically scale to fit inside the text field. If you anticipate your text label to change—such as if the string is localized—you should keep this selected. Setting a minimum font size ensures that your text will never appear smaller than intended. Use the Min Font Size (`minimumFontSize`) field if you want to change the value from its default.



Using Images in Text Fields

A text field can have a background image that sits under the content of the text field. Use the Background (`background`) field to set a background image for the normal state and the Disabled (`disabledBackground`) field to set a background image for when the control is disabled.



Using Auto Layout with Text Fields

You can create Auto Layout constraints between a text field and other user interface elements. You can create any type of constraint for a text field.

You will generally need to specify what a text field is intended for. You can use a label to do this. Place the label to the left of the text field and give the label and text field a "Horizontal Spacing" constraint.

For general information about using Auto Layout with iOS controls, see "Using Auto Layout with Controls" (page 120).

Making Text Fields Accessible

Text fields are accessible by default. The default accessibility traits for a text field are User Interaction Enabled and Adjustable.

For general information about making iOS controls accessible, see "Making Controls Accessible" (page 121).

Internationalizing Text Fields

The default language of the device affects the keyboard that pops up with the text field (including the return key). You don't need to do anything to enable this functionality; it is enabled by default. However, your text field should be able to handle input that comes from any language.

For more information, see *Internationalization Programming Topics*.

Elements Similar to a Text Field

The following elements provide similar functionality to a text field:

- **Text View.** A text view accepts and displays multiple lines of text. Text views support scrolling and text editing. You typically use a text view to display a large amount of text, such as the body of an email message. For more information, see "Text Views" (page 98).
- **Label.** A label displays static text. Labels are often used in conjunction with controls to describe their intended purpose, such as explaining which value a button or slider affects. For more information, see "Labels" (page 49).

Attributes Inspector Reference

Important: This is a preliminary document for an API or technology in development. Although this document has been reviewed for technical accuracy, it is not final. This Apple confidential information is for use only by registered members of the applicable Apple Developer program. Apple is supplying this confidential information to help you plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future seeds of the API or technology.

Activity Indicator View

Important: This is a preliminary document for an API or technology in development. Although this document has been reviewed for technical accuracy, it is not final. This Apple confidential information is for use only by registered members of the applicable Apple Developer program. Apple is supplying this confidential information to help you plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future seeds of the API or technology.

Activity Indicator View Attributes Inspector Reference

Appearance and Behavior

Style

The basic appearance of the activity indicator.

Selection	Method	Argument
Large White	`activityIndicatorViewStyle`	`UIActivityIndicator-ViewStyleWhiteLarge`
White	`activityIndicatorViewStyle`	`UIActivityIndicatorViewStyleWhite`
Gray	`activityIndicatorViewStyle`	`UIActivityIndicatorViewStyleGray`

Color

The color of the activity indicator.

Access: `color`

Behavior

Animating

Whether the activity indicator is moving.

Access: `isAnimating`

Selection	Method
Unselected	`stopAnimating`
Selected	`startAnimating`

Hides When Stopped

Whether the activity indicator is not shown when it is not animating.

Selection	Method	Argument
Unselected	`hidesWhenStopped`	`NO`
Selected	`hidesWhenStopped`	`YES`

Bar Button Item

Bar Button Item Attributes Inspector Reference

Appearance

Style

The bar button item style.

Selection	Method	Argument
Plain	`style`	`UIBarButtonItemStylePlain`
Bordered	`style`	`UIBarButtonItemStyleBordered`
Done	`style`	`UIBarButtonItemStyleDone`

Identifier

The button type of the bar button item.

To specify a custom button, use `initWithTitle:image:tag:` or `image`.

Selection	Method	Argument
Custom		
Flexible Space	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystem-ItemFlexibleSpace`
Fixed Space	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystem-ItemFixedSpace`
Add	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystemItemAdd`
Edit	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystemItemEdit`

Selection	Method	Argument
Done	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystemItemDone`
Cancel	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystemItemCancel`
Save	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystemItemSave`
Undo	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystemItemUndo`
Redo	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystemItemRedo`
Compose	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystem– ItemCompose`
Reply	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystemItemReply`
Action	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystemItemAction`
Organize	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystem– ItemOrganize`
Trash	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystemItemTrash`
Bookmarks	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystem– ItemBookmarks`
Search	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystemItemSearch`
Refresh	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystem– ItemFlexibleSpace`
Stop	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystemItemStop`
Camera	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystemItemCamera`

Selection	Method	Argument
Play	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystemItemPlay`
Pause	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystemItemPause`
Rewind	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystemItemRewind`
Fast Forward	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystem-ItemFastForward`
Page Curl	`initWithBarButtonSystemItem: target:action:`	`UIBarButtonSystem-ItemPageCurl`

Tint

The bar button item color.

Access: `tintColor`

Bar Item

Bar Item Attributes Inspector Reference

Appearance, Behavior, and Tagging

Title

The bar item title.

Access: `title`.

Image

The bar item image.

Access: `image`.

Tag

The bar item tag.

Access: `tag`.

Enabled

Whether the bar item is enabled.

Selection	Method	Argument
Unselected	`enabled`	`NO`
Selected	`enabled`	`YES`

Button

Button Attributes Inspector Reference

Type

Type

The button type, which determines its functionality.

Selection	Method	Argument
Custom	`buttonType`	`UIButtonTypeCustom`
Rounded Rect	`buttonType`	`UIButtonTypeRoundedRect`
Detail Disclosure	`buttonType`	`UIButtonTypeDetailDisclosure`
Info Light	`buttonType`	`UIButtonTypeInfoLight`
Info Dark	`buttonType`	`UIButtonTypeInfoDark`
Add Contact	`buttonType`	`UIButtonTypeContactAdd`

Appearance

This group specifies the appearance of the button in each of its possible states.

State Config

The button state to configure.

Choose a state, and then configure the remaining properties in this group. The settings are applied to those properties when the button goes to that state.

Selection	Method	Argument
Default	`state`	`UIControlStateNormal`

Selection	Method	Argument
Highlighted	`state`	`UIControlStateHighlighted`
Selected	`state`	`UIControlStateSelected`
Disabled	`state`	`UIControlStateDisabled`

Title

Title Type

The type of text to use for the button title when the button is in the state identified by State Config.

Selection
Plain
Attributed

Title Type: Plain

These are the properties you can configure for plain text titles in the state identified by State Config.

Text

The plain text for the button title in the state identified by State Config.

Access: `titleForState:`, `setTitle:forState:`.

Font

The font for the button plain text title in the state identified by State Config.

Use `<button>.titleLabel.font` to access the value of this property.

Text Color

The color for the button plain text title in the state identified by State Config.

Access: `titleColorForState:`, `setTitleColor:forState:`.

Shadow Color

The color for the button plain text title in the state identified by State Config.

Access: `titleShadowColorForState:`, `setTitleShadowColor:forState:`.

Title Type: Attributed

These are the properties you can configure for attributed text titles in the state identified by State Config.

Attributed Title Layout

The alignment and other layout characteristics for the button attributed text title in the state identified by State Config.

You can set these layout characteristics: alignment (left, center, right, justified, and natural), text color, background color, text direction, line breaking, line height and spacing, paragraph spacing, indentation, hyphenation, truncation, and header level).

Selection
Left
Center
Right
Justified
Natural
Text Color
Background Color
More

Font

The font for the button attributed text title in the state identified by State Config.

Access: `button.titleLabel.font`.

Attributed Text

The attributed text for the button title in the state identified by State Config.

Access: `attributedTitleForState:`.

Image

The image for the button when it is in the state identified by State Config.

Access: `imageForState:`, `setImage:forState:`.

Background

The background image for the button when it is in the state identified by State Config.

Access: `backgroundImageForState:`, `setBackgroundImage:forState:`.

Behavior

Shadow Offset

Offset Size

The width and height of the shadow of the button title.

Access: `titleShadowOffset`.

Reverses On Highlight

Whether the shadow of the button title changes when the button state changes to or from highlighted.

Selection	Method	Argument
Unselected	`reversesTitleShadowWhenHighlighted`	`NO`
Selected	`reversesTitleShadowWhenHighlighted`	`YES`

Highlight Tint

The color for the button tint.

`tintColor`

Drawing

Shows Touch On Highlight

Whether the button glows when it is tapped.

Selection	Method	Argument
Unselected	`showsTouchWhenHighlighted`	`NO`

Selection	Method	Argument
Selected	showsTouchWhenHighlighted	YES

Highlighted Adjusts Image

Whether the button image changes when the button state changes to or from highlighted.

Selection	Method	Argument
Unselected	adjustsImageWhenHighlighted	NO
Selected	adjustsImageWhenHighlighted	YES

Disabled Adjusts Image

Whether the button image changes when the button state changes to or from disabled.

Selection	Method	Argument
Unselected	adjustsImageWhenDisabled	NO
Selected	adjustsImageWhenDisabled	YES

Line Break

The line break mode for the button title.

Selection	Method	Argument
Clip	lineBreakMode	UILineBreakModeClip
Character Wrap	lineBreakMode	UILineBreakModeCharacterWrap
Word Wrap	lineBreakMode	UILineBreakModeWordWrap
Truncate Head	lineBreakMode	UILineBreakModeHeadTruncation
Truncate Middle	lineBreakMode	UILineBreakModeMiddleTruncation
Truncate Tail	lineBreakMode	UILineBreakModeTailTruncation

Edge Insets

Edge insets resize and reposition the effective drawing rectangle for the button's entire content, its title, and its image.

Edge

The button edge to configure.

Selection
Content
Title
Image

Inset

The inset or outset margins of the rectangle identified by the Edge property.

You can specify a value for each the edges (top, left, bottom, right). A positive value shrinks (or insets) the corresponding edge, moving it closer to the center of the button. A negative value expands (or outsets) the corresponding edge.

Access: `contentEdgeInsets`, `titleEdgeInsets`, `imageEdgeInsets`.

Collection Reusable View

Collection Reusable View Attributes Inspector Reference

Cell Reuse

Identifier

The collection reusable view reuse identifier.

Access: `reuseIdentifier`.

Collection View

Collection View Attributes Inspector Reference

Layout, Scrolling, Header, and Footer

Layout

The layout to use to lay out the collection view content.

Access: `collectionViewLayout`

Selection
Flow
Custom

Layout: Flow

Scroll Direction

The direction along which the collection view scrolls.

Access: `collection_view.collectionViewLayout.scrollDirection`

Selection	Method	Argument
Vertical	`scrollDirection`	`UICollectionViewScrollDirectionVertical`
Horizontal	`scrollDirection`	`UICollectionViewScrollDirectionHorizontal`

Accessories

Section Header

Whether the collection view has a header.

Access: `collection_view.collectionViewLayout.headerReferenceSize` `headerReferenceSize`.

Selection	Code
Unselected	`collection_view.collectionViewLayout.headerReferenceSize = 0.0`
Selected	`collection_view.collectionViewLayout.headerReferenceSize = // Floating-point value greater than 0.0`

Section Footer

Whether the collection view has a footer.

Access: `collection_view.collectionViewLayout.footerReferenceSize` `footerReferenceSize`.

Selection	Code
Unselected	`collection_view.collectionViewLayout.footerReferenceSize = 0.0`
Selected	`collection_view.collectionViewLayout.footerReferenceSize = // Floating-point value greater than 0.0`

Layout: Custom

Class

The class of the layout to use to lay out the collection view content.

Collection View Cell

Collection View Cell Attributes Inspector Reference

Cell Reuse

Identifier

The collection view cell reuse identifier.

Access: `reuseIdentifier`.

Control

Control Attributes Inspector Reference

Layout

Alignment

Horizontal

The horizontal alignment of the content.

Selection	Method	Argument
Left	`contentHorizontalAlignment`	`UIControlContentHorizontalAlignment-Left`
Center	`contentHorizontalAlignment`	`UIControlContentHorizontalAlignment-Center`
Right	`contentHorizontalAlignment`	`UIControlContentHorizontalAlignment-Right`
Fill	`contentHorizontalAlignment`	`UIControlContentHorizontalAlignment-Fill`

Vertical

The vertical alignment of the content.

Selection	Method	Argument
Top	`contentVerticalAlignment`	`UIControlContentVerticalAlignmentTop`
Center	`contentVerticalAlignment`	`UIControlContentVerticalAlignmentCenter`
Bottom	`contentVerticalAlignment`	`UIControlContentVerticalAlignmentBottom`
Fill	`contentVerticalAlignment`	`UIControlContentVerticalAlignmentFill`

Behavior

Content

Selected

Whether the control is selected.

Selection	Method	Argument
Unselected	`selected`	`NO`
Selected	`selected`	`YES`

Enabled

Whether the user can interact with the control.

Selection	Method	Argument
Unselected	`enabled`	`NO`
Selected	`enabled`	`YES`

Highlighted

Whether the control is highlighted.

Choice	Method	Argument
Unselected	`highlighted`	`NO`
Selected	`highlighted`	`YES`

Date Picker

Date Picker Attributes Inspector Reference

Functionality

Mode

The functionality of the date picker.

The mode indicates whether the date picker is a date or time picker, or a count down timer.

Selection	Method	Argument
Time	`datePickerMode`	`UIDatePickerModeTime`
Date	`datePickerMode`	`UIDatePickerModeDate`
Date and Time	`datePickerMode`	`UIDatePickerModeDateAndTime`
Count Down Timer	`datePickerMode`	`UIDatePickerModeCountDownTimer`

Locale

The locale for the date picker.

Access: `locale`.

Interval

The interval at which the date picker displays minutes.

Access: `minuteInterval`.

Selection
1 minute
2 minutes

Selection
3 minutes
4 minutes
5 minutes
6 minutes
10 minutes
12 minutes
15 minutes
20 minutes
30 minutes

Date

These properties are unused when the date picker mode is count down timer.

Date

The date the date picker displays.

Access: `date`.

Constraints

Minimum Date

The earliest date the date picker displays.

Access: `minimumDate`.

Maximum Date

The latest date the date picker displays.

Access: `maximumDate`.

Count Down Timer

Count Down in Seconds

The number of seconds from which the date picker counts down.

Access: `countDownDuration`.

Image View

Image View Attributes Inspector Reference

Images

The properties in this group are not used when you specify an image sequence using `animationImages`.

Image

The image displayed in the image view when its state is not highlighted.

Access: `image`.

Highlighted

The image displayed in the image view when its state is highlighted.

Access: `highlightedImage`.

Behavior

State

Highlighted

Whether the image view is highlighted.

Toggle this property after setting Image and Highlighted to preview the images for each state.

Selection	Method	Argument
Unselected	`highlighted`	`NO`
Selected	`highlighted`	`YES`

Label

Label Attributes Inspector Reference

Text and Behavior

Text

Text Type

The type of the label text.

Selection
Plain
Attributed

Text Type: Plain

This group configures the label plain text.

Text

The label plain text.

Access: `text`.

Color

The color of the label plain text.

Access: `textColor`.

Font

The font of the label plain text.

Access: `font`.

Alignment

The alignment of the label plain text.

Selection	Method	Argument
Left	`textAlignment`	`NSTextAlignmentLeft`
Center	`textAlignment`	`NSTextAlignmentCenter`
Right	`textAlignment`	`NSTextAlignmentRight`

Text Type: Attributed

This group configures the text selected in Attributed Text.

Attributed Text Layout

The alignment and other layout characteristics of the selected attributed text.

To set alignment and other layout characteristics of part of the attributed text in code, you need to create an attributed string with the desired characteristics and assign it to `attributedText`.

Selection
Left
Center
Right
Justified
Natural
Text Color
Background Color
More

Font

The font of the selected attributed text.

Access: `font`.

Attributed Text

The label attributed text.

Access: `attributedText`.

Lines

The maximum number of lines for the label text.

Set to 0 for an unlimited number of lines.

Access: `numberOfLines`.

Behavior

Enabled

Whether the label is enabled.

Selection	Method	Argument
Unselected	`enabled`	`NO`
Selected	`enabled`	`YES`

Highlighted

Whether the label is highlighted.

Toggle this property after setting the text highlight color to preview the highlighted label.

Selection	Method	Argument
Unselected	`highlighted`	`NO`
Selected	`highlighted`	`YES`

Text Baseline and Line Breaks

Baseline

How to adjust the text baseline so that the text fits in the label.

Selection	Method	Argument
Align Baselines	baselineAdjustment	UIBaselineAdjustmentAlignBaselines
Align Centers	baselineAdjustment	UIBaselineAdjustmentAlignCenters
None	baselineAdjustment	UIBaselineAdjustmentNone

Line Breaks

The line break mode of the label text.

Selection	Method	Argument
Clip	lineBreakMode	UILineBreakModeClip
Character Wrap	lineBreakMode	UILineBreakModeCharacterWrap
Word Wrap	lineBreakMode	UILineBreakModeWordWrap
Truncate Head	lineBreakMode	UILineBreakModeHeadTruncation
Truncate Middle	lineBreakMode	UILineBreakModeMiddleTruncation
Truncate Tail	lineBreakMode	UILineBreakModeTailTruncation

Text Sizing

Autoshrink

How to shrink the label text so that it fits in the label.

Selection	Method	Argument	Method
Fixed Font Size	adjustsFontSize-ToFitWidth	NO	
Minimum Font Scale	adjustsFontSize-ToFitWidth	YES	minimumScaleFactor
Minimum Font Size	adjustsFontSize-ToFitWidth	YES	minimumFontSize

Tighten Letter Spacing

Whether to reduce the label text letter spacing so that it fits in the label.

Selection	Method	Argument
Unselected	`adjustsLetterSpacingToFitWidth`	`NO`
Selected	`adjustsLetterSpacingToFitWidth`	`YES`

Text Highlight and Shadow

Highlighted

The color for the label text when the label is highlighted.

This property applies only to plain text labels.

Access: `highlightedTextColor`.

Shadow

The color of the label text shadow.

Access: `shadowColor`.

Shadow Offset

The offset of the label text shadow.

Access: `shadowOffset`.

Navigation Bar

Navigation Bar Attributes Inspector Reference

Appearance

Style

The basic appearance of the navigation bar.

Selection	Method	Argument
Default	`barStyle`	`UIBarStyleDefault`
Black Opaque	`barStyle`	`UIBarStyleBlack`
Black Translucent	`barStyle`	`UIBarStyleBlackTranslucent`

Tint

The color of the navigation bar.

Access: `tintColor`.

Navigation Item

Navigation Item Attributes Inspector Reference

PropertyGroup

Title

The navigation item title. Access: `title`.

Prompt

The navigation item prompt. Access: `prompt`.

Back Button

The bar button item the navigation item displays when it needs a back button. Access: `backBarButtonItem`.

Page Control

Page Control Attributes Inspector Reference

Behavior and Pages

Number of Pages

The number of pages the page control shows.

Access: `numberOfPages`

Current

The index of the page control current page.

Access: `currentPage`

Behavior

Hides for Single Page

Whether the page control is hidden when the number of pages is 1.

Choice	Method	Argument
Unselected	`hidesForSinglePage`	`NO`
Selected	`hidesForSinglePage`	`YES`

Defers Page Display

Whether the page control defers updating its display when the user selects a page.

When display-update is deferred, the page control does not update its display when the user selects a page. To update the display, call `updateCurrentPageDisplay`.

Choice	Method	Argument
Unselected	`defersCurrentPageDisplay`	`NO`
Selected	`defersCurrentPageDisplay`	`YES`

Appearance

Tint Color

The color of the page control dots that correspond to non-open pages.

Access: `pageIndicatorTintColor`

Current Page

The color of the page control dot that corresponds to the currently open page.

Access: `currentPageIndicatorTintColor`

Picker View

Picker View Attributes Inspector Reference

Behavior

Shows Selection Indicator

Whether the picker view shows the bar that identifies the selected row in each component.

Selection	Method	Argument
Unselected	`showsSelectionIndicator`	`NO`
Selected	`showsSelectionIndicator`	`YES`

Progress View

Progress View Attributes Inspector Reference

Appearance and Progress

Style

The basic appearance of the progress view.

Selection	Method	Argument
Default	`progressViewStyle`	`UIProgressViewStyleDefault`
Bar	`progressViewStyle`	`UIProgressViewStyleBar`

Progress

The progress view progress.

Access: `progress`.

Progress Tint

The color of the progress view completed-progress bar.

Access: `progressTintColor`.

Track Tint

The color of the progress view track.

Access: `trackTintColor`.

Scroll View

Scroll View Attributes Inspector Reference

Appearance

Style

The appearance of the scroll view scroll indicators.

Selection	Method	Argument
Default	`indicatorStyle`	`UIScrollViewIndicatorStyleDefault`
Black	`indicatorStyle`	`UIScrollViewIndicatorStyleBlack`
White	`indicatorStyle`	`UIScrollViewIndicatorStyleWhite`

Scroll Indicators and Scrolling

Scrollers

Shows Horizontal Scrollers

Whether the scroll view horizontal scroll indicator is visible.

Selection	Method	Argument
Unselected	`showsHorizontalScrollIndicator`	`NO`
Selected	`showsHorizontalScrollIndicator`	`YES`

Shows Vertical Scrollers

Whether the scroll view vertical scroll indicator is visible.

Selection	Method	Argument
Unselected	`showsVerticalScrollIndicator`	`NO`
Selected	`showsVerticalScrollIndicator`	`YES`

Scrolling Enabled

Whether the user can scroll the scroll view contents.

When scrolling is enabled the user can scroll the scroll view contents.

Selection	Method	Argument
Unselected	`scrollEnabled`	`NO`
Selected	`scrollEnabled`	`YES`

Paging Enabled

Whether the scroll view stops at regular intervals as the user scrolls.

When paging is enabled the scroll view stops on multiples of the scroll view bounds as the user scrolls.

Selection	Method	Argument
Unselected	`pagingEnabled`	`NO`
Selected	`pagingEnabled`	`YES`

Direction Lock Enabled

Whether the user can scroll in only one cardinal direction at a time.

When direction lock is enabled the user can scroll in only one cardinal direction at a time, vertically or horizontally, except when the user scrolls diagonally.

Selection	Method	Argument
Unselected	`directionalLockEnabled`	`NO`
Selected	`directionalLockEnabled`	`YES`

Scroll Bounce

Bounce

Bounces

Whether the scroll view bounces when the user scrolls past a content edge.

Selection	Method	Argument
Unselected	`bounces`	`NO`
Selected	`bounces`	`YES`

Bounce Horizontally

Whether the scroll view bounces when the user scrolls past a horizontal content edge.

When horizontal bouncing is allowed, the scroll view lets the user scroll horizontally even when the scroll view content does not fill the scroll view bounds horizontally.

Selection	Method	Argument
Unselected	`alwaysBounceHorizontal`	`NO`
Selected	`alwaysBounceHorizontal`	`YES`

Bounce Vertically

Whether the scroll view bounces when the user scrolls past a vertical content edge.

When vertical bouncing is allowed, the scroll view lets the user scroll vertically even when the scroll view content does not fill the scroll view bounds vertically.

Selection	Method	Argument
Unselected	`alwaysBounceVertical`	`NO`
Selected	`alwaysBounceVertical`	`YES`

Zooming

Zoom

Min

The minimum zoom scale factor the user can apply to the scroll view content.

Access: `minimumZoomScale`.

Max

The maximum zoom scale factor the user can apply to the scroll view content.

Access: `maximumZoomScale`.

Events and Zoom Bounce

Touch

Bounces Zoom

Whether the scroll view bounces when the user zooms past the scroll view minimum or maximum zoom scale factors.

Selection	Method	Argument
Unselected	`bouncesZoom`	`NO`
Selected	`bouncesZoom`	`YES`

Delays Content Touches

Whether the scroll view delays the handling of every touch event to determine whether the event is a scrolling event.

Selection	Method	Argument
Unselected	`delaysContentTouches`	`NO`
Selected	`delaysContentTouches`	`YES`

Cancellable Content Touches

Whether the scroll view can convert a touch event into a scroll event to reflect user intent.

When content touches are cancellable, the view can cancel the handling of a touch event and initiate a scroll event when the user drags.

Selection	Method	Argument
Unselected	`canCancelContentTouches`	`NO`
Selected	`canCancelContentTouches`	`YES`

Search Bar

Search Bar Attributes Inspector Reference

Search Term and Captions

Text

The initial search string.

Access: `text`.

Placeholder

The text that appears when there is no search string.

Access: `placeholder`.

Prompt

The text that appears above the text field.

Access: `prompt`.

Appearance

Style

The basic appearance of the search bar.

Selection	Method	Argument
Default	`barStyle`	`UIBarStyleDefault`
Black Opaque	`barStyle`	`UIBarStyleBlack`
Black Translucent	`barStyle`	`UIBarStyleBlackTranslucent`

Tint

The color of the search bar.

Access: `tintColor`.

Capabilities

Shows Search Results Button

Whether the search bar displays the Search Results button.

Selection	Method	Argument
Unselected	`showsSearchResultsButton`	`NO`
Selected	`showsSearchResultsButton`	`YES`

Shows **Bookmarks Button**

Whether the search bar displays the Bookmarks button.

Selection	Method	Argument
Unselected	`showsBookmarkButton`	`NO`
Selected	`showsBookmarkButton`	`YES`

Shows Cancel Button

Whether the search bar displays the Cancel button.

Selection	Method	Argument
Unselected	`showsCancelButton`	`NO`
Selected	`showsCancelButton`	`YES`

Shows Scope Bar

Whether the search bar displays the scope bar.

Selection	Method	Argument
Unselected	`showsScopeBar`	`NO`
Selected	`showsScopeBar`	`YES`

Scope Titles

Scope Title List

The titles of the scope buttons in the scope bar.

Access: `scopeButtonTitles`.

Text Input

Capitalize

Whether and when the keyboard activates the Shift key while the user types text.

Selection	Method	Argument
None	`autocapitalizationType`	`UITextAutocapitalizationTypeNone`
Words	`autocapitalizationType`	`UITextAutocapitalizationTypeWords`
Sentences	`autocapitalizationType`	`UITextAutocapitalizationTypeSentences`
All Characters	`autocapitalizationType`	`UITextAutocapitalization-TypeAllCharacters`

Correction

Whether to auto-correct text the user types.

Selection	Method	Argument
Default	`autocorrectionType`	`UITextAutocorrectionTypeDefault`
No	`autocorrectionType`	`UITextAutocorrectionTypeNo`
Yes	`autocorrectionType`	`UITextAutocorrectionTypeYes`

Keyboard

The type of keyboard that appears on text input.

Selection	Method	Argument
Default	`keyboardType`	`UIKeyboardTypeDefault`
ASCII Capable	`keyboardType`	`UIKeyboardTypeASCIICapable`
Numbers and Punctuation	`keyboardType`	`UIKeyboardTypeNumbersAndPunctuation`
URL	`keyboardType`	`UIKeyboardTypeURL`
Number Pad	`keyboardType`	`UIKeyboardTypeNumberPad`
Phone Pad	`keyboardType`	`UIKeyboardTypePhonePad`
E-mail Address	`keyboardType`	`UIKeyboardTypeEmailAddress`

Segmented Control

Segmented Control Attributes Inspector Reference

Appearance and Behavior

This group specifies the appearance and behavior for the segmented control.

Style

The basic appearance of the segmented control.

Selection	Method	Argument
Plain	`segmentedControlStyle`	`UISegmentedControlStylePlain`
Bordered	`segmentedControlStyle`	`UISegmentedControlStyleBordered`
Bar	`segmentedControlStyle`	`UISegmentedControlStyleBar`

State

Momentary

Whether the segmented control segments are only momentarily selected.

A segmented control selection behavior can be *normal* or *momentary*.

- **Normal selection.** When the user taps a segment, the segment becomes selected (the segmented control `selectedSegmentIndex` is set to that segment's index).
- **Momentary selection.** When the user taps a segment, the segment does not become selected, but the segment appears selected while the user holds it.

Selection	Method	Argument
Unselected	`momentary`	`NO`
Selected	`momentary`	`YES`

Tint

The color of the segmented control.

`tintColor`

Segments

The number of segments in the segmented control.

Access: `numberOfSegments`.

Segment Appearance and Behavior

This group specifies the appearance and behavior for each of the segmented control segments.

Segment

The segment to configure.

Choose a segment, and then configure the properties in this group.

Title

The title of the segment identified by Segment.

Access: `setTitle:forSegmentAtIndex:`.

Image

The image of the segment identified by Segment.

Access: `setImage:forSegmentAtIndex:`.

Behavior

Enabled

Whether segment identified by Segment is enabled.

Selection	Method	Argument
Unselected	`setEnabled: forSegmentAtIndex:`	`NO`
Selected	`setEnabled: forSegmentAtIndex:`	`YES`

Selected

Whether segment identified by Segment is selected.

Use `selectedSegmentIndex` to select a segment.

Selection
Unselected
Selected

Content Offset

The offset from the segment origin at which to draw the content of the segment identified by Segment.

Access: `setContentOffset:forSegmentAtIndex:`.

Slider

Slider Attributes Inspector Reference

Value

Minimum

The bottom of the range of values the slider can have.

Access: `minimumValue`.

Maximum

The top of the range of values the slider can have.

Access: `maximumValue`.

Current

The numeric value of the slider.

Access: `value`.

Images

Min Image

The image displayed on the minimum side of the slider.

Access: `minimumValueImage`.

Max Image

The image displayed on the maximum side of the slider.

Access: `maximumValueImage`.

Appearance

Min Track Tint

The color of the track on the minimum side of the slider.

Access: `minimumTrackTintColor`.

Max Track Tint

The color of the track on the maximum side of the slider.

Access: `maximumTrackTintColor`.

Thumb Tint

The color of the slider thumb.

Access" `thumbTintColor`.

Update Events

Continuous

Whether the slider continuously sends update events as the user moves the thumb.

You can receive updated values on the slider by responding to the value changed event `UIControlEventValueChanged`. In continuous updates, the slider sends multiple value changed events as the user moves the thumb. In noncontinuous updates, the slider sends one value changed event when the user releases the thumb.

Selection	Method	Argument
Unselected	`continuous`	`NO`
Selected	`continuous`	`YES`

Stepper

Stepper Attributes Inspector Reference

Value

Minimum

The minimum value of the stepper.

Access: `minimumValue`.

Maximum

The maximum value of the stepper.

Access: `maximumValue`.

Current

The value of the stepper.

Access: `value`.

Step

The amount by which the stepper increments or decrements its value.

Access: `stepValue`.

Behavior

Autorepeat

Whether the stepper keeps incrementing or decrementing its value while either of its buttons is held down.

Selection	Method	Argument
Unselected	`autorepeat`	`NO`

Selection	Method	Argument
Selected	`autorepeat`	`YES`

Continuous

Whether the stepper continuously sends update events as either of its buttons is held down.

In continuous updates the stepper sends update events as the user holds the button. In noncontinuous updates the stepper sends one update event when the user releases the button.

Selection	Method	Argument
Unselected	`continuous`	`NO`
Selected	`continuous`	`YES`

Wraps

Whether the stepper wraps around the minimum and maximum values.

Selection	Method	Argument
Unselected	`wraps`	`NO`
Selected	`wraps`	`YES`

Switch

Switch Attributes Inspector Reference

Appearance and State

State

Whether the switch is on or off.

Selection	Method	Argument
On	`on`	`YES`
Off	`on`	`NO`

On Tint

The color of the switch when it is on.

Access: `onTintColor`.

Tab Bar

Tab Bar Attributes Inspector Reference

Appearance

Tint

The color of the tab bar.

Access: `tintColor`.

Image Tint

The color for the image of tab bar items when selected at runtime.

Access: `selectedImageTintColor`.

Tab Bar Item

Tab Bar Item Attributes Inspector Reference

Appearance

Badge

The tab bar item badge.

Access: `badgeValue`.

Identifier

The tab bar item icon.

Selection	Method	Argument
Custom		
More	`initWithTabBarSystemItem: tag:`	`UITabBarSystemItemMore`
Favorites	`initWithTabBarSystemItem: tag:`	`UITabBarSystemItemFavorites`
Featured	`initWithTabBarSystemItem: tag:`	`UITabBarSystemItemFeatured`
Top Rated	`initWithTabBarSystemItem: tag:`	`UITabBarSystemItemTopRated`
Recents	`initWithTabBarSystemItem: tag:`	`UITabBarSystemItemRecents`
Contacts	`initWithTabBarSystemItem: tag:`	`UITabBarSystemItemContacts`
History	`initWithTabBarSystemItem: tag:`	`UITabBarSystemItemHistory`
Bookmarks	`initWithTabBarSystemItem: tag:`	`UITabBarSystemItemBookmarks`
Search	`initWithTabBarSystemItem: tag:`	`UITabBarSystemItemSearch`
Downloads	`initWithTabBarSystemItem: tag:`	`UITabBarSystemItemDownloads`
Most Recent	`initWithTabBarSystemItem: tag:`	`UITabBarSystemItemMostRecent`
Most Viewed	`initWithTabBarSystemItem: tag:`	`UITabBarSystemItemMostViewed`

Table View

Table View Attributes Inspector Reference

Appearance

Style

The basic appearance of the table view.

Selection	Method	Argument
Plain	`style`	`UITableViewStylePlain`
Grouped	`style`	`UITableViewStyleGrouped`

Separator

Separator Style

The style for table view cells used as separators.

Selection	Method	Argument
None	`separatorStyle`	`UITableViewCellSeparatorStyleNone`
Single Line	`separatorStyle`	`UITableViewCellSeparatorStyleSingleLine`
Single Line Etched	`separatorStyle`	`UITableViewCellSeparatorStyleSingle-LineEtched`

Separator Color

The color for table view cells used as separators.

Access: `separatorColor`

Behavior

Selection

The type of selection the table view allows.

Access: `allowsSelection` `allowsMultipleSelection`

Selection	Code
No Selection	`table_view.allowsSelection = NO`
Single Selection	`table_view.allowsSelection = YES`
Multiple Selection	`table_view.allowsMultipleSelection = YES`

Editing

The type of selection the table view allows in editing mode.

Access: `allowsSelectionDuringEditing` `allowsMultipleSelectionDuringEditing`

Selection	Code
No Selection During Editing	`table_view.allowsSelectionDuringEditing = NO`
Single Selection During Editing	`table_view.allowsSelectionDuringEditing = YES`
Multiple Selection During Editing	`table_view.allowsMultipleSelectionDuringEditing = YES`

Show Selection on Touch

This item appears to be nonoperational.

Table Index

Index Row Limit

This item appears to be nonoperational.

Table View Cell

Table View Cell Attributes Inspector Reference

Style

The Image property appears in this group only when Style is Basic, Right Detail, or Subtitle.

Style

The appearance and layout of the table view cell labels.

For the Custom style, your custom subclass of `UITableViewCell` determines the appearance of the table view cell.

Access: `reuseIdentifier`

Selection	Method	Argument
Custom		
Basic	`initWithStyle: reuseIdentifier:`	`UITableViewCellStyleDefault`
Right Detail	`initWithStyle: reuseIdentifier:`	`UITableViewCellStyleValue1`
Left Detail	`initWithStyle: reuseIdentifier:`	`UITableViewCellStyleValue2`
Subtitle	`initWithStyle: reuseIdentifier:`	`UITableViewCellStyleSubtitle`

Image

The image for the table view cell content.

Access: `image`

Cell Reuse

Identifier

Identifies the table view cell for cell reuse.

Access: `initWithStyle:reuseIdentifier:reuseIdentifier`

Appearance

Selection

Whether and how the table view cell indicates that it is selected.

Selection	Method	Argument
None	`selectionStyle`	`UITableViewCellSelectionStyleNone`
Blue	`selectionStyle`	`UITableViewCellSelectionStyleBlue`
Gray	`selectionStyle`	`UITableViewCellSelectionStyleGray`

Accessory

The accessory view type for the table view cell in the Normal state.

The three states of a table view cell are: Normal, Editing, and Delete Confirmation (`Cell State Mask Constants`).

Selection	Method	Argument
None	`accessoryType`	`UITableViewCellAccessoryNone`
Disclosure Indicator	`accessoryType`	`UITableViewCellAccessoryDisclosureIndicator`
Detail Disclosure	`accessoryType`	`UITableViewCellAccessoryDetail-DisclosureButton`
Checkmark	`accessoryType`	`UITableViewCellAccessoryCheckmark`

Editing Acc.

The accessory view type of the table view cell in the Editing state.

The three states of a table view cell are: Normal, Editing, and Delete Confirmation (`Cell State Mask Constants`).

Selection	Method	Argument
None	`editingAccessoryType`	`UITableViewCellAccessoryNone`
Disclosure Indicator	`editingAccessoryType`	`UITableViewCellAccessoryDisclosure-Indicator`
Detail Disclosure	`editingAccessoryType`	`UITableViewCellAccessoryDetail-DisclosureButton`
Checkmark	`editingAccessoryType`	`UITableViewCellAccessoryCheckmark`

Indentation and Behavior

Indentation

Level

The table view cell indentation level.

Access: `indentationLevel`

Width

The width in points of each level of indentation.

Access: `indentationWidth`

Indent While Editing

Whether to indent the table view cell background when the table view cell is in the Editing state.

Selection	Method	Argument
Unselected	`shouldIndentWhileEditing`	`NO`
Selected	`shouldIndentWhileEditing`	`YES`

Shows Re-order Controls

Whether the table view cell shows the reordering control.

Selection	Method	Argument
Unselected	`showsReorderControl`	`NO`

Selection	Method	Argument
Selected	`showsReorderControl`	`YES`

Text View

Text View Attributes Inspector Reference

Text

Text

Text Type

The type of the text view text.

Selection
Plain
Attributed

Text Type: Plain

Use this group to configure the text view plain text.

Text

The text view plain text.

Access: `text`

Color

The color of the text view plain text.

Access: `textColor`

Font

The font of the text view plain text.

Access: `font`

Alignment

The alignment of the text view plain text.

Selection	Method	Argument
Left	`textAlignment`	`NSTextAlignmentLeft`
Center	`textAlignment`	`NSTextAlignmentCenter`
Right	`textAlignment`	`NSTextAlignmentCenter`

Text Type: Attributed

Use this group set the characteristics of the selected text in Attributed Text.

To set alignment and other layout characteristics of part of the attributed text in code, you need to create an attributed string with the desired characteristics and assign it to `attributedText`.

Attributed Text Layout

The alignment and other layout characteristics of the selected attributed text.

Selection
Left
Center
Right
Justified
Natural
Text Color
Background Color
More

Font

The font of the selected attributed text.

Attributed Text

The the text view attributed text.

Access: `attributedText`

Allows Editing Attributes

Whether the user can change characteristics of the attributed text.

Selection	Method	Argument
Unselected	`allowsEditingTextAttributes`	`NO`
Selected	`allowsEditingTextAttributes`	`YES`

Behavior

Editable

Whether the user can edit text view text.

Selection	Method	Argument
Unselected	`editable`	`NO`
Seleted	`editable`	`YES`

Data Detection

Detection

Links

Whether the text view detects hyperlinks in the text.

Access: `dataDetectorTypes`

Selection	Code
Unselected	`text_view.dataDetectorTypes &= ~UIDataDetectorTypeLink`
Selected	`text_view.dataDetectorTypes |= UIDataDetectorTypeLink`

Addresses

Whether the text view detects addresses in the text.

Access: `dataDetectorTypes`

Selection	Code
Unselected	`text_view.dataDetectorTypes &= ~UIDataDetectorTypeAddress`
Selected	`text_view.dataDetectorTypes |= UIDataDetectorTypeAddresses`

Phone Numbers

Whether the view detects phone numbers in the text.

Access: `dataDetectorTypes`

Selection	Code
Unselected	`text_view.dataDetectorTypes &= ~UIDataDetectorTypePhoneNumber`
Selected	`text_view.dataDetectorTypes |= UIDataDetectorTypePhoneNumber`

Events

Whether the text view detects calendar events in the text.

Access: `dataDetectorTypes`

Selection	Code
Unselected	`text_view.dataDetectorTypes &= ~UIDataDetectorTypeCalendarEvent`
Selected	`text_view.dataDetectorTypes |= UIDataDetectorTypeCalendarEvent`

Text Input and Keyboard

Capitalization

Whether and when the keyboard activates the Shift key while the user types text.

Selection	Method	Argument
None	`autocapitalizationType`	`UITextAutocapitalizationTypeNone`

Selection	Method	Argument
Words	autocapitalizationType	UITextAutocapitalizationTypeWords
Sentences	autocapitalizationType	UITextAutocapitalizationTypeSentences
All Characters	autocapitalizationType	UITextAutocapitalization-TypeAllCharacters

Correction

Whether to auto-correct text the text user types.

Selection	Method	Argument
Default	autocorrectionType	UITextAutocorrectionTypeDefault
No	autocorrectionType	UITextAutocorrectionTypeNo
Yes	autocorrectionType	UITextAutocorrectionTypeYes

Keyboard

The keyboard that appears on text input.

Selection	Method	Argument
Default	keyboardType	UIKeyboardTypeDefault
ASCII Capable	keyboardType	UIKeyboardTypeASCIICapable
Numbers and Punctuation	keyboardType	UIKeyboardTypeNumbersAnd-Punctuation
URL	autocorrectionType	UIKeyboardTypeURL
Number Pad	keyboardType	UIKeyboardTypeNumberPad
Phone Pad	keyboardType	UIKeyboardTypePhonePad
Name Phone Pad	keyboardType	UIKeyboardTypeNamePhonePad
E-mail Address	keyboardType	UIKeyboardTypeEmailAddress

Appearance

The keyboard to use for text input.

Selection	Method	Argument
Default	keyboardAppearance	UIKeyboardAppearanceDefault
Alert	keyboardAppearance	UIKeyboardAppearanceAlert

Return Key

The type of the keyboard Return key.

The type of the Return key specifies the Return key title.

Selection	Method	Argument
Default	returnKeyType	UIReturnKeyDefault
Go	returnKeyType	UIReturnKeyGo
Google	returnKeyType	UIReturnKeyGoogle
Join	returnKeyType	UIReturnKeyJoin
Next	returnKeyType	UIReturnKeyNext
Route	returnKeyType	UIReturnKeyRoute
Search	returnKeyType	UIReturnKeySearch
Send	returnKeyType	UIReturnKeySend
Yahoo	returnKeyType	UIReturnKeyYahoo
Done	returnKeyType	UIReturnKeyDone
Emergency Call	returnKeyType	UIReturnKeyEmergencyCall

Auto-enable Return Key

Whether the user can press the Return key after entering text.

Selection	Method	Argument
Unselected	enablesReturnKeyAutomatically	NO
Selected	enablesReturnKeyAutomatically	YES

Secure

Whether the text view hides entered text.

Selection	Method	Argument
Unselected	`secureTextEntry`	`NO`
Selected	`secureTextEntry`	`YES`

Toolbar

Toolbar Attributes Inspector Reference

Appearance

Style

The basic appearance of the toolbar.

Selection	Method	Argument
Default	`barStyle`	`UIBarStyleDefault`
Black Opaque	`barStyle`	`UIBarStyleBlack`
Black Translucent	`barStyle`	`UIBarStyleBlackTranslucent`

Tint

The color of the toolbar.

Access: `tintColor`.

View

View Attributes Inspector Reference

Layout and Tagging

Mode

How the view presents its content when the view size changes.

Selection	Method	Argument
Scale to Fill	`contentMode`	`UIViewContentModeScaleToFill`
Aspect Fit	`contentMode`	`UIViewContentModeScaleAspectFit`
Aspect Fill	`contentMode`	`UIViewContentModeScaleAspectFill`
Redraw	`contentMode`	`UIViewContentModeRedraw`
Center	`contentMode`	`UIViewContentModeCenter`
Top	`contentMode`	`UIViewContentModeTop`
Bottom	`contentMode`	`UIViewContentModeBottom`
Left	`contentMode`	`UIViewContentModeLeft`
Right	`contentMode`	`UIViewContentModeRight`
Top Left	`contentMode`	`UIViewContentModeTopLeft`
Top Right	`contentMode`	`UIViewContentModeTopRight`
Bottom Left	`contentMode`	`UIViewContentModeBottomLeft`
Bottom Right	`contentMode`	`UIViewContentModeBottomRight`

Tag

The view tag.

Access: `tag`.

Events

Interaction

User Interaction Enabled

Whether the view processes touch and keyboard events.

Selection	Method	Argument
Unselected	`userInteractionEnabled`	`NO`
Selected	`userInteractionEnabled`	`YES`

Multiple Touch

Whether the view processes multiple touch events.

Selection	Method	Argument
Unselected	`multipleTouchEnabled`	`NO`
Selected	`multipleTouchEnabled`	`YES`

Appearance

Alpha

The view transparency.

The range of values is from 0.0 and 1.0:

- 0.0 makes the view completely transparent.
- 1.0 makes the view completely opaque.

Access: `alpha`.

Background

The view background color.

Access: `backgroundColor`.

Drawing and Sizing

Drawing

Opaque

Whether the drawing system treats the view as opaque.

Selection	Method	Argument
Unselected	`opaque`	`NO`
Selected	`opaque`	`YES`

Hidden

Whether the view is hidden.

Selection	Method	Argument
Unselected	`hidden`	`NO`
Selected	`hidden`	`YES`

Clears Graphics Context

Whether the view clears its bounds before drawing.

Selection	Method	Argument
Unselected	`clearsContextBeforeDrawing`	`NO`
Selected	`clearsContextBeforeDrawing`	`YES`

Clip Subviews

Whether subviews are clipped to the view bounds.

If the view alpha value (`alpha`) is 1.0, drawing performance can be improved by indicating that the view is opaque.

Selection	Method	Argument
Unselected	`clipsToBounds`	`NO`
Selected	`clipsToBounds`	`YES`

Autoresize Subviews

Whether the view resizes subviews when the view size changes.

If the view alpha value (`alpha`) is 1.0, drawing performance can be improved by indicating that the view is opaque.

Selection	Method	Argument
Unselected	`autoresizesSubviews`	`NO`
Selected	`autoresizesSubviews`	`YES`

Sizing

Stretching

The rectangle that identifies stretchable area of the view.

The values for the rectangle specifiers (X, Y, Width, and Height) are in the range 0.0 to 1.0. For example, to make only half of the view stretchable, specify the rectangle (0.0, 0.0, 0.5, 1.0).

Access: `contentStretch`.

Web View

Web View Attributes Inspector

Data Detection and Sizing

Scaling

Scales Page to Fit

Whether the view resizes the webpage to fit the view bounds.

Selection	Method	Argument
Unselected	`autoresizesSubviews`	`NO`
Selected	`autoresizesSubviews`	`YES`

Detection

Links

Whether the view detects hyperlinks in the webpage.

Access: `dataDetectorTypes`.

Selection	Code
Unselected	`web_view.dataDetectorTypes &amp;= ~UIDataDetectorTypeLink`
Selected	`web_view.dataDetectorTypes |= UIDataDetectorTypeLink`

Addresses

Whether the view detects addresses in the webpage.

Access: `dataDetectorTypes`.

Selection	Code
Unselected	`web_view.dataDetectorTypes &= ~UIDataDetectorTypeAddress`
Selected	`web_view.dataDetectorTypes |= UIDataDetectorTypeAddresses`

Phone Numbers

Whether the view detects phone numbers in the webpage.

Selection	Code
Unselected	`web_view.dataDetectorTypes &= ~UIDataDetectorTypePhoneNumber`
Selected	`web_view.dataDetectorTypes |= UIDataDetectorTypePhoneNumber`

Events

Whether the view detects calendar events in the webpage.

Selection	Code
Unselected	`web_view.dataDetectorTypes &= ~UIDataDetectorTypeCalendarEvent`
Selected	`web_view.dataDetectorTypes |= UIDataDetectorTypeCalendarEvent`

Document Revision History

This table describes the changes to *UIKit User Interface Catalog* .

Date	Notes
2013-06-10	New document that describes the views and controls in UIKit, drawing from the human interface guidelines for more depth.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010