UIResponder Class Reference

Overview

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

There are two general kinds of events: touch events and motion events. The primary event-handling methods for touches are touchesBegan:withEvent:, touchesMoved:withEvent:, touchesEnded:withEvent:, and touchesCancelled:withEvent:. The parameters of these methods associate touches with their events—especially touches that are new or have changed—and thus allow responder objects to track and handle the touches as the delivered events progress through the phases of a multi-touch sequence. Any time a finger touches the screen, is dragged on the screen, or lifts from the screen, a UIEvent object is generated. The event object contains UITouch objects for all fingers on the screen or just lifted from it.

iPhone OS 3.0 introduced system capabilities for generating motion events, specifically the motion of shaking the device. The event-handling methods for these kinds of events are motionBegan:withEvent:, motionEnded:withEvent:, and motionCancelled:withEvent:. Additionally for iPhone OS 3.0, the canPerformAction:withSender: method allows responders to validate commands in the user interface while the undoManager property returns the nearest NSUndoManager object in the responder chain.

See Event Handling in iPhone Application Programming Guide for further information on event handling.

Tasks

Managing the Responder Chain

– nextResponder
– isFirstResponder
– canBecomeFirstResponder
– becomeFirstResponder
– canResignFirstResponder
– resignFirstResponder

Responding to Touch Events

– touchesBegan:withEvent:
– touchesMoved:withEvent:
– touchesEnded:withEvent:

– touchesCancelled:withEvent:

Responding to Motion Events

– motionBegan:withEvent:
– motionEnded:withEvent:
– motionCancelled:withEvent:

Getting the Undo Manager

undoManager property

Validating Commands

– canPerformAction:withSender:

UIApplication Class Reference

Overview

The UIApplication class provides a centralized point of control and coordination for applications running on iPhone OS.

Every application must have exactly one instance of UIApplication (or a subclass of UIApplication). When an application is launched, the UIApplicationMain function is called; among its other tasks, this function create a singleton UIApplication object. Thereafter you can access this object by invoking the sharedApplication class method.

A major role of a UIApplication object is to handle the initial routing of incoming user events. It also dispatches action messages forwarded to it by control objects (UIControl) to the appropriate target objects. In addition, the UIApplication object maintains a list of all the windows (UIWindow objects) currently open in the application, so through those it can retrieve any of the applicationâ□□s UIView objects. The application object is typically assigned a delegate, an object that the application informs of significant runtime eventsâ□□for example, application launch, low-memory warnings, and application terminationâ□□giving it an opportunity to respond appropriately.

Applications can cooperatively handle a resource such as an email or an image file through the openURL: method. For example, an application opening an email URL with this method may cause the mail client to launch and display the message.

For iPhone OS 3.0, UIApplication has added methods for remote-notification registration, for triggering of the undo-redo UI (applicationSupportsShakeToEdit), and for determining whether any installed application can open a URL (canOpenURL:).

UIApplication defines a delegate that must adopt the UIApplicationDelegate protocol implement one or more of the methods.

The programmatic interfaces of UIApplication and UIApplicationDelegate also allow you to manage behavior that is specific to the device. You can control application response to changes in interface orientation, temporarily suspend incoming user events, and turn proximity sensing (of the userâ□□s face) off and on again.

Subclassing Notes

You might decide to subclass UIApplication to override sendEvent: or sendAction:to:from:forEvent: to implement custom event and action dispatching. However, there is rarely a valid need to extend this class; the application delegate (UIApplicationDelegate is sufficient for most occasions. If you do subclass UIApplication, be very sure of what you are trying to accomplish with the subclass.

Tasks

Getting the Application Instance

+ sharedApplication

Getting Application Windows

keyWindow property
windows property

Controlling and Handling Events

– sendEvent:
– sendAction:to:from:forEvent:
– beginIgnoringInteractionEvents
– endIgnoringInteractionEvents
– isIgnoringInteractionEvents
applicationSupportsShakeToEdit property
proximitySensingEnabled property

Opening a URL Resource

– openURL:
– canOpenURL:

Registering for Remote Notifications

– registerForRemoteNotificationTypes:
– unregisterForRemoteNotifications
– enabledRemoteNotificationTypes

Managing Application Activity

idleTimerDisabled property

Managing Status Bar Orientation

– setStatusBarOrientation:animated:
statusBarOrientation property
statusBarOrientationAnimationDuration property

Controlling Application Appearance

– setStatusBarHidden:animated:
statusBarHidden property
– setStatusBarStyle:animated:
statusBarStyle property
statusBarFrame property
networkActivityIndicatorVisible property
applicationIconBadgeNumber property

Setting and Getting the Delegate

delegate property

UIView Class Reference

Overview

The UIView class is primarily an abstract superclass that provides concrete subclasses with a structure for drawing and handling events. You can also create instances of UIView to contain other views.

UIView objects are arranged within an UIWindow object, in a nested hierarchy of subviews. Parent objects in the view hierarchy are called **superviews**, and children are called **subviews**. A view object claims a rectangular region of its enclosing superview, is responsible for all drawing within that region, and is eligible to receive events occurring in it as well. Sibling views are able to overlap without any issues, allowing complex view placement.

The UIView class provides common methods you use to create all types of views and access their properties. For example, unless a subclass has its own designated initializer, you use the initWithFrame: method to create a view. The frame property specifies the origin and size of a view in superview coordinates. The origin of the coordinate system for all views is in the upper-left corner.

You can also use the center and bounds properties to set the position and size of a view. The center property specifies the view's center point in superview's coordinates. The bounds property specifies the origin in the view's coordinates and its size (the view's content may be larger than the bounds size). The frame property is actually computed based on the center and bounds property values. Therefore, you can set any of these three properties and they affect the values of the others.

It's important to set the autoresizing properties of views so that when they are displayed or the orientation changes, the views are displayed correctly within the superview's bounds. Use the autoresizesSubviews property, especially if you subclass UIView, to specify whether the view should automatically resize its subviews. Use the autoresizingMask property with the constants described in UIViewAutoresizing to specify how a view should automatically resize.

The UIView class provides a number of methods for managing the view hierarchy. Use the superview property to get the parent view and the subviews property to get the child views in the hierarchy. There are also a number of methods, listed in "Managing the View Hierarchy," for adding, inserting, and removing subviews as well as arranging subviews in front of or in back of siblings.

When you subclass UIView to create a custom class that draws itself, implement the drawRect: method to draw the view within the specified region. This method is invoked the first time a view displays or when an event occurs that invalidates a part of the

view's frame requiring it to redraw its content.

Normal geometry changes do not require redrawing the view. Therefore, if you alter the appearance of a view and want to force it to redraw, send setNeedsDisplay or setNeedsDisplayInRect: to the view. You can also set the contentMode to UIViewContentModeRedraw to invoke the drawRect: method when the bounds change; otherwise, the view is scaled and clipped without redrawing the content.

Subclasses can also be containers for other views. In this case, just override the designated initializer, initWithFrame:, to create a view hierarchy. If you want to programmatically force the layout of subviews before drawing, send setNeedsLayout to the view. Then when layoutIfNeeded is invoked, the layoutSubviews method is invoked just before displaying. Subclasses should override layoutSubviews to perform any custom arrangement of subviews.

Some of the property changes to view objects can be animated—for example, setting the frame, bounds, center, and transform properties. If you change these properties in an animation block, the changes from the current state to the new state are animated. Invoke the beginAnimations:context: class method to begin an animation block, set the properties you want animated, and then invoke the commitAnimations class method to end an animation block. The animations are run in a separate thread and begin when the application returns to the run loop. Other animation class methods allow you to control the start time, duration, delay, and curve of the animations within the block.

Use the hitTest:withEvent: and pointInside:withEvent: methods if you are processing events and want to know where they occur. The UIView class inherits other event processing methods from UIResponder. For more information on how views handle events, read UIResponder Class Reference.

Read Window and Views in iPhone Application Programming Guide to learn how to use this class.

Note: Prior to iPhone OS 3.0, UIView instances may have a maximum height and width of 1024 x 1024. In iPhone OS 3.0 and later, views are no longer restricted to this maximum size but are still limited by the amount of memory they consume. Therefore, it is in your best interests to keep view sizes as small as possible. Regardless of which version of iPhone OS is running, you should consider using a CATiledLayer object if you need to create views larger than 1024 x 1024 in size.

Tasks

Creating Instances

– initWithFrame:

Setting and Getting Attributes

userInteractionEnabled property

Modifying the Bounds and Frame Rectangles

frame property
bounds property
center property
transform property

Managing the View Hierarchy

superview property
subviews property
window property
– addSubview:
– bringSubviewToFront:
– sendSubviewToBack:
– removeFromSuperview
– insertSubview:atIndex:
– insertSubview:aboveSubview:
– insertSubview:belowSubview:
– exchangeSubviewAtIndex:withSubviewAtIndex:
– isDescendantOfView:

Converting Coordinates

– convertPoint:toView:
– convertPoint:fromView:
– convertRect:toView:
– convertRect:fromView:

Resizing Subviews

autoresizesSubviews property
autoresizingMask property
– sizeThatFits:
– sizeToFit
contentMode property
contentStretch property

Searching for Views

tag property
– viewWithTag:

Laying out Views

– setNeedsLayout
– layoutIfNeeded
– layoutSubviews

Displaying

clipsToBounds property
backgroundColor property
alpha property
opaque property
clearsContextBeforeDrawing property
– drawRect:

– setNeedsDisplay
– setNeedsDisplayInRect:
+ layerClass
layer property
hidden property

Animating Views

+ beginAnimations:context:
+ commitAnimations
+ setAnimationStartDate:
+ setAnimationsEnabled:
+ setAnimationDelegate:
+ setAnimationWillStartSelector:
+ setAnimationDidStopSelector:
+ setAnimationDuration:
+ setAnimationDelay:
+ setAnimationCurve:
+ setAnimationRepeatCount:
+ setAnimationRepeatAutoreverses:
+ setAnimationBeginsFromCurrentState:
+ setAnimationTransition:forView:cache:
+ areAnimationsEnabled

Handling Events

– hitTest:withEvent:
– pointInside:withEvent:
multipleTouchEnabled property
exclusiveTouch property
– endEditing:

Observing Changes

– didAddSubview:
– didMoveToSuperview
– didMoveToWindow
– willMoveToSuperview:
– willMoveToWindow:
– willRemoveSubview:

UIWindow Class Reference

Overview

The UIWindow class defines objects (known as **windows**) that manage and coordinate the windows an application displays on the screen. The two principal functions of a window are to provide an area for displaying its views and to distribute events to the views. The window is the root view in the view hierarchy. A window belongs to a level; the windows in one level appear above another level. For example, alerts appear above normal windows. Typically, there is only one window in an iPhone OS application.

Read Windows and Views in iPhone Application Programming Guide to learn how to use this class.

Tasks

Configuring Windows

windowLevel property

Making Windows Key

keyWindow property
– makeKeyAndVisible
– becomeKeyWindow
– makeKeyWindow
– resignKeyWindow

Converting Coordinates

– convertPoint:toWindow:
– convertPoint:fromWindow:
– convertRect:toWindow:
– convertRect:fromWindow:

Sending Events

– sendEvent:

UILabel Class Reference

Overview

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

The default content mode of the UILabel class is UIViewContentModeRedraw. This mode causes the view to redraw its contents every time its bounding rectangle changes. You can change this mode by modifying the inherited contentMode property of the class.

New label objects are configured to disregard user events by default. If you want to handle events in a custom subclass of UILabel, you must explicitly change the value of the userInteractionEnabled property to YES after initializing the object.

Tasks

Accessing the Text Attributes

text property
font property
textColor property
textAlignment property
lineBreakMode property
enabled property

Sizing the Label's Text

adjustsFontSizeToFitWidth property
baselineAdjustment property
minimumFontSize property
numberOfLines property

Managing Highlight Values

highlightedTextColor property
highlighted property

Drawing a Shadow

shadowColor property
shadowOffset property

Drawing and Positioning Overrides

– textRectForBounds:limitedToNumberOfLines:

– drawTextInRect:

Setting and Getting Attributes

userInteractionEnabled property

UIPickerView Class Reference

Overview

The UIPickerView class implements objects, called picker views, that use a spinning-wheel or slot-machine metaphor to show one or more sets of values. Users select values by rotating the wheels so that the desired row of values aligns with a selection indicator.

The UIDatePicker class uses a custom subclass of UIPickerView to display dates and times. To see an example, tap the add (“+”) button in the the Alarm pane of the Clock application.

The user interface provided by a picker view consists of components and rows. A component is a wheel, which has a series of items (rows) at indexed locations on the wheel. Each component also has an indexed location (left to right) in a picker view. Each row on a component has content, which is either a string or a view object such as a label (UILabel) or an image (UIImageView).

A UIPickerView object requires the cooperation of a delegate for constructing its components and a data source for providing the numbers of components and rows. The delegate must adopt the UIPickerViewDelegate protocol and implement the required methods to return the drawing rectangle for rows in each component. It also provides the content for each component’s row, either as a string or a view, and it typically responds to new selections or deselections. The data source must adopt the UIPickerViewDataSource protocol and implement the required methods to return the number of components and the number of rows in each component.

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.

Tasks

Getting the Dimensions of the View Picker

numberOfComponents property
– numberOfRowsInComponent:
– rowSizeForComponent:

Reloading the View Picker

– reloadAllComponents
– reloadComponent:

Selecting Rows in the View Picker

– selectRow:inComponent:animated:
– selectedRowInComponent:

Returning the View for a Row and Component

– viewForRow:forComponent:

Specifying the Delegate

delegate property

Specifying the Data Source

dataSource property

Managing the Appearance of the Picker View

showsSelectionIndicator property

UIProgressView Class Reference

Overview

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

The UIProgressView class provides properties for managing the style of the progress bar and for getting and setting values that are pinned to the progress of a task.

For an indeterminate progress indicator—or, informally, a "spinner"—use an instance of the UIActivityIndicatorView class.

Tasks

Initializing the UIProgressView Object

– initWithProgressViewStyle:

Managing the Progress Bar

progress property

Configuring the Bar Style

progressViewStyle property

UIActivityIndicatorView Class Reference

Overview

The UIActivityIndicatorView class creates and manages an indicator showing the indeterminate progress of a task. Visually, this indicator is a “gear” that is animated to spin.

You control when the progress indicator animates with the startAnimating and stopAnimating methods. If the hidesWhenStopped property is set to YES, the indicator is automatically hidden when animation stops.

Tasks

Initializing an UIActivityIndicatorView Object

– initWithActivityIndicatorStyle:

Managing the Activity Indicator

– startAnimating
– stopAnimating
– isAnimating
hidesWhenStopped property

Managing the Indicator Style

activityIndicatorViewStyle property

UIImageView Class Reference

Overview

An image view object provides a view-based container for displaying either a single image or for animating a series of images. For animating the images, the UIImageView class provides controls to set the duration and frequency of the animation. You can also start and stop the animation freely.

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.

Subclassing Notes

Special Considerations

The UIImageView class is optimized to draw its images to the display. UIImageView will not call drawRect: a subclass. If your subclass needs custom drawing code, it is recommended you use UIView as the base class.

Tasks

Initializing a UIImageView Object

– initWithImage:
– initWithImage:highlightedImage:

Image Data

image property
highlightedImage property

Animating Images

animationImages property
highlightedAnimationImages property
animationDuration property
animationRepeatCount property
– startAnimating
– stopAnimating
– isAnimating

Setting and Getting Attributes

userInteractionEnabled property
highlighted property

UITabBar Class Reference

Overview

The UITabBar class implements a control for selecting one of two or more buttons, called items. The most common use of a tab bar is to implement a modal interface where tapping an item changes the selection. Use a UIToolbar object if you want to momentarily highlight or not change the appearance of an item when tapped. The UITabBar class provides the ability for the user to customize the tab bar by reordering, removing, and adding items to the bar. You can use a tab bar delegate to augment this behavior.

Use the UITabBarItem class to create items and the setItems:animated: method to add them to a tab bar. All methods with an animated: argument allow you to optionally animate changes to the display. Use the selectedItem property to access the current item.

Important: In iPhone OS 3.0 and later, you should not attempt to use the methods and properties of this class to modify the tab bar when it is associated with a tab bar controller object. Modifying the tab bar in this way results in the throwing of an exception. Instead, any modifications to the tab bar or its items should occur through the tab bar controller interface. You may still directly modify a tab bar object that is not associated with a tab bar controller.

Tasks

Getting and Setting Properties

delegate property

Configuring Items

items property
selectedItem property
– setItems:animated:

Customizing Tab Bars

– beginCustomizingItems:
– endCustomizingAnimated:
– isCustomizing

UIToolbar Class Reference

Overview

An instance of the UIToolbar class is a control for selecting one of many buttons, called toolbar items. A toolbar momentarily highlights or does not change the appearance of an item when tapped. Use the UITabBar class if you need a radio button style control.

Use the UIBarButtonItem class to create items and the setItems:animated: method to add them to a toolbar. All methods with an animated: argument allow you to optionally animate changes to the display.

Toolbar images that represent normal and highlighted states of an item derive from the image you set using the inherited image property from the UIBarItem class. For example, the image is converted to white and then bevelled by adding a shadow for the normal state.

Tasks

Configuring the Toolbar

barStyle property
tintColor property
translucent property

Configuring Toolbar Items

items property
– setItems:animated:

UINavigationBar Class Reference

Overview

The UINavigationBar class implements a control for navigating hierarchical content. It's a bar, typically displayed at the top of the screen, containing buttons for navigating up and down a hierarchy. The primary properties are a left (back) button, a center title, and an optional right button. You can specify custom views for each of these.

You can use a navigation bar as a standalone object or in conjunction with a navigation controller object. To use a navigation bar as a standalone object, you create it and add it to your view hierarchy like you would any other view. Specifically, you can create it in Interface Builder and load it with the rest of your views or you can create it programmatically using the standard alloc and initWithFrame: methods.

You can modify the appearance of the bar using the barStyle, tintColor, and translucent properties. These properties affect the visual appearance of the bar itself but they also affect the way buttons are displayed in the bar. For example, if you set the translucent property to YES, any buttons in the bar are also made partially opaque.

For information about using a navigation bar with a navigation controller object, see "Using With a Navigation Controller."

Adding Content to a Navigation Bar

When you use a navigation bar as a standalone object, you are responsible for providing its contents. Unlike other types of views, you do not add subviews to a navigation bar directly. Instead, you use a navigation item (an instance of the UINavigationItem class) to specify what buttons or custom views you want displayed. A navigation item has properties for specifying views on the left, right, and center of the navigation bar and for specifying a custom prompt string.

A navigation bar manages a stack of UINavigationItem objects. Although the stack is there mostly to support navigation controllers, you can use it as well to implement your own custom navigation interface. The topmost item in the stack represents the navigation item whose contents are currently displayed by the navigation bar. You push new navigation items onto the stack using the pushNavigationItem:animated: method and pop items off the stack using the popNavigationItemAnimated: method. Both of these changes can be animated for the benefit of the user.

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.

If you are using a navigation bar as a standalone object, you should assign a custom delegate object to the delegate property and use that object to intercept messages coming from the navigation bar. Delegate objects must conform to the UINavigationBarDelegate protocol. The delegate notifications let you know when the contents of responsible for deciding when items are pushed or popped from the stack—for example, it should display the previous view when the user clicks the back button.

For more information about creating navigation items, see UINavigationItem Class Reference. For more information about implementing a delegate object, see UINavigationBarDelegate Protocol Reference.

Using With a Navigation Controller

The most common way to use a navigation bar is in conjunction with a UINavigationController object. If you use a navigation controller to manage the navigation between different screens of content, the navigation controller creates the navigation bar automatically and pushes and pops navigation items when appropriate. You do not have to create the navigation bar and you do not have to manage the pushing and popping of navigation items yourself.

When used in conjunction with a navigation controller, there are only a handful of direct customizations you can make to the navigation bar. Specifically, it is alright to modify the barStyle, tintColor, and translucent properties of this class, but you must never directly change UIView-level properties such as the frame, bounds, alpha, or hidden properties directly. In addition, you should let the navigation controller manage the stack of navigation items and not attempt to modify these items yourself.

A navigation controller automatically assigns itself as the delegate of its navigation bar object. Therefore, when using a navigation controller, you must not attempt to assign a custom delegate object to the corresponding navigation bar.

Tasks

Configuring Navigation Bars

barStyle property
tintColor property
translucent property

Assigning the Delegate

delegate property

Pushing and Popping Items

– pushNavigationItem:animated:

– popNavigationItemAnimated:
– setItems:animated:
items property
topItem property
backItem property

UITableViewCell Class Reference

Overview

The UITableViewCell class defines the attributes and behavior of the cells that appear in UITableView objects.

A UITableViewCell object (or table cell) includes properties and methods for managing cell selection, highlighted state, editing state and controls, accessory views, reordering controls, cell background, and content indentation. The class additionally includes properties for setting and managing cell content, specifically text and images.

For iPhone OS 3.0, UITableViewCell includes two major improvements:

- Predefined cell styles that position elements of the cell (labels and images) in certain locations and with certain attributes. See "Cell Styles" for descriptions of the constants that apply to these styles.
- Properties for accessing the content of the cell. These properties include textLabel, detailTextLabel, and imageView. Once you get the associated UILabel and UIImageView objects, you can set their attributes, such as text color, font, image, highlighted image, and so on.

You have two ways of extending the standard UITableViewCell object beyond the given styles. To create cells with multiple, variously formatted and sized strings and images for content, you can get the cell'□□s content view (through its contentView property) and add subviews to it. You can also subclass UITableViewCell to obtain cell characteristics and behavior specific to your application'□□s needs. See "□□A Closer Look at Table-View Cells" in Table View Programming Guide for iPhone OS for details.

Note: Setting the background color of a cell (via the backgroundColor property declared by UIView) that is in a group-style table view has an effect in iPhone OS 3.0 that is different than previous versions of the operating system. It now affects the area inside the rounded rectangle instead of the area outside of it.

Tasks

Initializing a UITableViewCell Object

– initWithStyle:reuseIdentifier:
– initWithFrame:reuseIdentifier: Deprecated in iPhone OS 3.0

Reusing Cells

reuseIdentifier property
– prepareForReuse

Managing Text as Cell Content

textLabel property
detailTextLabel property
text property
font property
textAlignment property
textColor property
selectedTextColor property
lineBreakMode property

Managing Images as Cell Content

imageView property
image property
selectedImage property

Accessing Views of the Cell Object

contentView property
backgroundView property
selectedBackgroundView property

Managing Accessory Views

accessoryType property
accessoryView property
editingAccessoryType property
editingAccessoryView property
hidesAccessoryWhenEditing property

Managing Cell Selection and Highlighting

selected property
selectionStyle property
– setSelected:animated:
highlighted property
– setHighlighted:animated:

Editing the Cell

editing property
– setEditing:animated:
editingStyle property
showingDeleteConfirmation property
showsReorderControl property

Adjusting to State Transitions

– willTransitionToState:
– didTransitionToState:

Managing Content Indentation

indentationLevel property

indentationWidth property
shouldIndentWhileEditing property

Managing Targets and Actions

These properties are deprecated as of iPhone OS 3.0. Instead, use the tableView:commitEditingStyle:forRowAtIndexPath: method of the UITableViewDataSource protocol or the tableView:accessoryButtonTappedForRowWithIndexPath: method of the UITableViewDelegate protocol.

target property
editAction property
accessoryAction property

UIActionSheet Class Reference

Overview

Use the UIActionSheet class to implement an action sheet that displays a message and presents buttons that let the user decide how to proceed. An action sheet is similar in function but differs in appearance from an alert view.

Use the properties and methods in this class to set the message, set the style, set the delegate, configure the buttons, and display the action sheet. You must set a delegate if you add custom buttons. The delegate should conform to the UIActionSheetDelegate protocol. When you display an action sheet, you can optionally animate it from the bottom bar or an arbitrary view. How the action sheet is animated depends on the bar style or the action sheet style you set.

Tasks

Creating Action Sheets

– initWithTitle:delegate:cancelButtonTitle:destructiveButtonTitle:otherButtonTitles:

Setting Properties

delegate property
title property
visible property
actionSheetStyle property

Configuring Buttons

– addButtonWithTitle:
numberOfButtons property
– buttonTitleAtIndex:
cancelButtonIndex property
destructiveButtonIndex property
firstOtherButtonIndex property

Displaying

– showFromTabBar:
– showFromToolbar:
– showInView:

Dismissing

– dismissWithClickedButtonIndex:animated:

UIAlertView Class Reference

Overview

Use the UIAlertView class to display an alert message to the user. An alert view functions similar to but differs in appearance from an action sheet (an instance of UIActionSheet).

Use the properties and methods defined in this class to set the title, message, and delegate of an alert view and configure the buttons. You must set a delegate if you add custom buttons. The delegate should conform to the UIAlertViewDelegate protocol. Use the show method to display an alert view once it is configured.

Tasks

Creating Alert Views

– initWithTitle:message:delegate:cancelButtonTitle:otherButtonTitles:

Setting Properties

delegate property
title property
message property
visible property

Configuring Buttons

– addButtonWithTitle:
numberOfButtons property
– buttonTitleAtIndex:
cancelButtonIndex property
firstOtherButtonIndex property

Displaying

– show

Dismissing

– dismissWithClickedButtonIndex:animated:

UIScrollView Class Reference

Overview

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

UIScrollView is the superclass of several UIKit classes including UITableView and UITextView.

The central notion of a UIScrollView object (or, simply, a scroll view) is that it is a view whose origin is adjustable over the content view. It clips the content to its frame, which generally (but not necessarily) coincides with that of the application's main window. A scroll view tracks the movements of fingers and adjusts the origin accordingly. The view that is showing its content "through" the scroll view draws that portion of itself based on the new origin, which is pinned to an offset in the content view. The scroll view itself does no drawing except for displaying vertical and horizontal scroll indicators. The scroll view must know the size of the content view so it knows when to stop scrolling; by default, it "bounces" back when scrolling exceeds the bounds of the content.

The object that manages the drawing of content displayed in a scroll view should tile the content's subviews so that no view exceeds the size of the screen. As users scroll in the scroll view, this object should add and remove subviews as necessary.

Because a scroll view has no scroll bars, it must know whether a touch signals an intent to scroll versus an intent to track a subview in the content. To make this determination, it temporarily intercepts a touch-down event by starting a timer and, before the timer fires, seeing if the touching finger makes any movement. If the time fires without a significant change in position, the scroll view sends tracking events to the touched subview of the content view. If the user then drags their finger far enough before the timer elapses, the scroll view cancels any tracking in the subview and performs the scrolling itself. Subclasses can override the touchesShouldBegin:withEvent:inContentView:, pagingEnabled, and touchesShouldCancelInContentView: methods (which are called by the scroll view) to affect how the scroll view handles scrolling gestures.

A scroll view also handles zooming and panning of content. As the user makes a pinch-in or pinch-out gesture, the scroll view adjusts the offset and the scale of the content. When the gesture ends, the object managing the content view should should update subviews of the content as necessary. (Note that the gesture can end and a finger could still be down.) While the gesture is in progress, the scroll view does not send any tracking calls to the subview.

The UIScrollView class can have a delegate that must adopt the UIScrollViewDelegate protocol. For zooming and panning to work, the delegate must implement both viewForZoomingInScrollView: and scrollViewDidEndZooming:withView:atScale:; in addition, the maximum (maximumZoomScale) and minimum (minimumZoomScale) zoom scale must be different.

Tasks

Managing the Display of Content

– setContentOffset:animated:
contentOffset property
contentSize property
contentInset property

Managing Scrolling

scrollEnabled property
directionalLockEnabled property
scrollsToTop property
– scrollRectToVisible:animated:
pagingEnabled property
bounces property
alwaysBounceVertical property
alwaysBounceHorizontal property
– touchesShouldBegin:withEvent:inContentView:
– touchesShouldCancelInContentView:
canCancelContentTouches property
delaysContentTouches property
decelerationRate property
dragging property
tracking property
decelerating property

Managing the Scroll Indicator

indicatorStyle property
scrollIndicatorInsets property
showsHorizontalScrollIndicator property
showsVerticalScrollIndicator property
– flashScrollIndicators

Zooming and Panning

– zoomToRect:animated:
zoomScale property
– setZoomScale:animated:
maximumZoomScale property
minimumZoomScale property
zoomBouncing property
zooming property
bouncesZoom property

Managing the Delegate

delegate property

UITableView Class Reference

Overview

An instance of UITableView (or simply, a table view) is a means for displaying and editing hierarchical lists of information.

A table view in the UIKit framework is limited to a single column because it is designed for a device with a small screen. UITableView is a subclass of UIScrollView, which allows users to scroll through the table, although UITableView allows vertical scrolling only. The cells comprising the individual items of the table are UITableViewCell objects; UITableView uses these objects to draw the visible rows of the table. Cells have content—□□titles and images—and can have, near the right edge, accessory views. Standard accessory views are disclosure indicators or detail disclosure buttons; the former leads to the next level in a data hierarchy and the latter leads to a detailed view of a selected item. Accessory views can also be framework controls, such as switches and sliders, or can be custom views. Table views can enter an editing mode where users can insert, delete, and reorder rows of the table.

A table view is made up of zero or more sections, each with its own rows. Sections are identified by their index number within the table view, and rows are identified by their index number within a section. Any section can optionally be preceded by a section header, and optionally be followed by a section footer.

Table views can have one of two styles, UITableViewStylePlain and UITableViewStyleGrouped. When you create a UITableView instance you must specify a table style, and this style cannot be changed. In the plain style, section headers and footers float above the content if the part of a complete section is visible. A table view can have an index that appears as a bar on the right hand side of the table (for example, "a" through "z"). You can touch a particular label to jump to the target section. The grouped style of table view provides a default background color and a default background view for all cells. The background view provides a visual grouping for all cells in a particular section. For example, one group could be a person's name and title, another group for phone numbers that the person uses, and another group for email accounts and so on. See the Settings application for examples of grouped tables. Table views in the grouped style cannot have an index.

Many methods of UITableView take NSIndexPath objects as parameters and return values. UITableView declares a category on NSIndexPath that enables you to get the represented row index (row property) and section index (section property), and to construct an index path from a given row index and section index (indexPathForRow:inSection: method). Especially in table views with multiple sections, you must evaluate the section index before identifying a row by its index number.

A UITableView object must have an object that acts as a data source and an object that acts as a delegate; typically these objects are either the application delegate or, more frequently, a custom UITableViewController object. The data source must adopt the UITableViewDataSource protocol and the delegate must adopt the UITableViewDelegate protocol. The data source provides information that UITableView needs to construct tables and manages the data model when rows of a table are inserted, deleted, or reordered. The delegate provides the cells used by tables and performs other tasks, such as managing accessory views and selections.

When sent a setEditing:animated: message (with a first parameter of YES), the table view enters into editing mode where it shows the editing or reordering controls of each visible row, depending on the editingStyle of each associated UITableViewCell. Clicking on the insertion or deletion control causes the data source to receive a tableView:commitEditingStyle:forRowAtIndexPath: message. You commit a deletion or insertion by calling deleteRowsAtIndexPaths:withRowAnimation: or insertRowsAtIndexPaths:withRowAnimation:, as appropriate. Also in editing mode, if a table-view cell has its showsReorderControl property set to YES, the data source receives a tableView:moveRowAtIndexPath:toIndexPath: message. The data source can selectively remove the reordering control for cells by implementing tableView:canMoveRowAtIndexPath:.

UITableView caches table-view cells only for visible rows, but caches row, header, and footer heights for the entire table. You can create custom UITableViewCell objects with content or behavioral characteristics that are different than the default cells; â□□A Closer Look at Table-View Cells" in Table View Programming Guide for iPhone OS explains how.

Tasks

Initializing a UITableView Object

– initWithFrame:style:

Configuring a Table View

– dequeueReusableCellWithIdentifier:
style property
– numberOfRowsInSection:
– numberOfSections
rowHeight property
separatorStyle property
separatorColor property
tableHeaderView property
tableFooterView property
sectionHeaderHeight property
sectionFooterHeight property
sectionIndexMinimumDisplayRowCount property

Accessing Cells and Sections

– cellForRowAtIndexPath:
– indexPathForCell:
– indexPathForRowAtPoint:
– indexPathsForRowsInRect:
– visibleCells
– indexPathsForVisibleRows

Scrolling the Table View

– scrollToRowAtIndexPath:atScrollPosition:animated:
– scrollToNearestSelectedRowAtScrollPosition:animated:

Managing Selections

– indexPathForSelectedRow
– selectRowAtIndexPath:animated:scrollPosition:
– deselectRowAtIndexPath:animated:
allowsSelection property
allowsSelectionDuringEditing property

Inserting and Deleting Cells

– beginUpdates
– endUpdates
– insertRowsAtIndexPaths:withRowAnimation:
– deleteRowsAtIndexPaths:withRowAnimation:
– insertSections:withRowAnimation:
– deleteSections:withRowAnimation:

Managing the Editing of Table Cells

editing property
– setEditing:animated:

Reloading the Table View

– reloadData
– reloadRowsAtIndexPaths:withRowAnimation:
– reloadSections:withRowAnimation:
– reloadSectionIndexTitles

Accessing Drawing Areas of the Table View

– rectForSection:
– rectForRowAtIndexPath:
– rectForFooterInSection:
– rectForHeaderInSection:

Managing the Delegate and the Data Source

dataSource property
delegate property

UITextView Class Reference

Overview

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

This class does not support multiple styles for text. The font, color, and text alignment attributes you specify always apply to the entire contents of the text view. To display more complex styling in your application, you need to use a UIWebView object and render your content using HTML.

Managing the Keyboard

When the user taps in an editable text view, that text view becomes the first responder and automatically asks the system to display the associated keyboard. Because the appearance of the keyboard has the potential to obscure portions of your user interface, it is up to you to make sure that does not happen by repositioning any views that might be obscured. Some system views, like table views, help you by scrolling the first responder into view automatically. If the first responder is at the bottom of the scrolling region, however, you may still need to resize or reposition the scroll view itself to ensure the first responder is visible.

It is your application's responsibility to dismiss the keyboard at the time of your choosing. You might dismiss the keyboard in response to a specific user action, such as the user tapping a particular button in your user interface. To dismiss the keyboard, send the resignFirstResponder message to the text view that is currently the first responder. Doing so causes the text view object to end the current editing session (with the delegate object's consent) and hide the keyboard.

The appearance of the keyboard itself can be customized using the properties provided by the UITextInputTraits protocol. Text view objects implement this protocol and support the properties it defines. You can use these properties to specify the type of keyboard (ASCII, Numbers, URL, Email, and others) to display. You can also configure the basic text entry behavior of the keyboard, such as whether it supports automatic capitalization and correction of the text.

Keyboard Notifications

When the system shows or hides the keyboard, it posts several keyboard notifications. These notifications contain information about the keyboard, including its size, which you can use for calculations that involve repositioning or resizing views. Registering for

these notifications is the only way to get some types of information about the keyboard. The system delivers the following notifications for keyboard-related events:

- UIKeyboardWillShowNotification
- UIKeyboardDidShowNotification
- UIKeyboardWillHideNotification
- UIKeyboardDidHideNotification

For more information about these notifications, see their descriptions in UIWindow Class Reference.

Tasks

Configuring the Text Attributes

text property
font property
textColor property
editable property
dataDetectorTypes property
textAlignment property
– hasText

Working with the Selection

selectedRange property
– scrollRangeToVisible:

Accessing the Delegate

delegate property

UISearchBar Class Reference

Overview

The UISearchBar class implements a text field control for text-based searches. The control provides a text field for entering text, a search button, a bookmark button, and a cancel button. The UISearchBar object does not actually perform any searches. You use a delegate, an object conforming to the UISearchBarDelegate protocol, to implement the actions when text is entered and buttons are clicked.

Tasks

Text Content

placeholder property
prompt property
text property

Display Attributes

barStyle property
tintColor property
translucent property

Text Input Properties

autocapitalizationType property
autocorrectionType property
keyboardType property

Button Configuration

showsBookmarkButton property
showsCancelButton property
– setShowsCancelButton:animated:

Scope Buttons

scopeButtonTitles property
selectedScopeButtonIndex property
showsScopeBar property

Delegate

delegate property

UIWebView Class Reference

Overview

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

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.

If you allow the user to move back and forward through the webpage history, then you can use the goBack and goForward methods as actions for buttons. Use the canGoBack and canGoForward properties to disable the buttons when the user can't move in a direction.

By default, a web view automatically converts telephone numbers that appear in web content to Phone links. When a Phone link is tapped, the Phone application launches and dials the number. Set the detectsPhoneNumbers property to NO to turn off this default behavior.

You can also use the scalesPageToFit property to programmatically set the scale of web content the first time it is displayed in a web view. Thereafter, the user can change the scale using gestures.

Set the delegate property to an object conforming to the UIWebViewDelegate protocol if you want to track the loading of web content.

Read Safari Web Content Guide for how to create web content that is compatible with and optimized for displaying in Safari on iPhone and your web views.

Subclassing Notes

The UIWebView class should not be subclassed.

Tasks

Setting the Delegate

delegate property

Loading Content

– loadData:MIMEType:textEncodingName:baseURL:
– loadHTMLString:baseURL:
– loadRequest:
request property
loading property
– stopLoading
– reload

Moving Back and Forward

canGoBack property
canGoForward property
– goBack
– goForward

Setting Web Content Properties

detectsPhoneNumbers property
scalesPageToFit property

Running JavaScript

– stringByEvaluatingJavaScriptFromString:

New Methods

dataDetectorTypes property

UIControl Class Reference

Overview

UIControl is the base class for controls: objects such as buttons and sliders that are used to convey user intent to the application. You cannot use UIControl directly to instantiate controls. It instead defines the common interface and behavioral structure for all subclasses of it.

The main role of UIControl is to define an interface and base implementation for preparing action messages and initially dispatching them to their targets when specified events occur. (See "The Target-Action Mechanism" for an overview.) It also includes methods for getting and setting control state (for example, for determining whether a control is enabled or highlighted) and it defines methods for tracking touches within a control (the latter group of methods are for overriding by subclasses).

The Target-Action Mechanism

The design of the target-action mechanism in the UIKit framework is based on the Multi-Touch event model. In iPhone OS the user's fingers (or touches) convey intent (instead of mouse clicks and drags), and there can be multiple touches at any moment on a control going in different directions.

Note: For more information on the Multi-Touch event model, see Event Handling in iPhone Application Programming Guide.

The UIControl.h header file declares a large number of control events as constants for a bit mask described in "Control Events". Some control events specify the behavior of touches in and around the control—various permutations of actions such a finger touching down in a control, dragging into and away from a control, and lifting up from a control. Other control events specify editing phases initiated when a finger touches down in a text field. And yet another control event, UIControlEventValueChanged, is for controls such as sliders, where a value continuously changes based on the manipulation of the control. For any particular action message, you can specify one or more control events as the trigger for sending that message. For example, you could request a certain action message be sent to a certain target when a finger touches down in a control or is dragged into it (UIControlEventTouchDown | UIControlEventTouchDragEnter).

You prepare a control for sending an action message by calling addTarget:action:forControlEvents: for each target-action pair you want to specify. This method builds an internal dispatch table associating each target-action pair with a

control event. When a user touches the control in a way that corresponds to one or more specified events, UIControl sends itself sendActionsForControlEvents:. This results in UIControl sending the action to UIApplication in a sendAction:to:from:forEvent: message. UIApplication is the centralized dispatch point for action messages; if a nil target is specified for an action message, the application sends the action to the first responder where it travels up the responder chain until it finds an object willing to handle the action message—that is, object that implements a method corresponding to the action selector. (Event Handling gives an overview of the first responder and the responder chain.)

UIKit allows three different forms of action selector:

```
- (void)action
- (void)action:(id)sender
- (void)action:(id)sender forEvent:(UIEvent *)event
```

The sendAction:to:fromSender:forEvent: method of UIApplication pushes two parameters when calling the target. These last two parameters are optional for the application because it's up to the caller (usually a UIControl object) to remove any parameters it added.

Subclassing Notes

You may want to extend a UIControl subclass for two basic reasons:

- To observe or modify the dispatch of action messages to targets for particular eventsTo do this, override sendAction:to:forEvent:, evaluate the passed-in selector, target object, or "Note" bit mask and proceed as required.
- To provide custom tracking behavior (for example, to change the highlight appearance)To do this, override one or all of the following methods: beginTrackingWithTouch:withEvent:, continueTrackingWithTouch:withEvent:, endTrackingWithTouch:withEvent:.

Tasks

Preparing and Sending Action Messages

– sendAction:to:forEvent:
– sendActionsForControlEvents:
– addTarget:action:forControlEvents:
– removeTarget:action:forControlEvents:
– actionsForTarget:forControlEvent:
– allTargets
– allControlEvents

Setting and Getting Control Attributes

state property
enabled property
selected property
highlighted property

contentVerticalAlignment property
contentHorizontalAlignment property

Tracking Touches and Redrawing Controls

– beginTrackingWithTouch:withEvent:
– continueTrackingWithTouch:withEvent:
– endTrackingWithTouch:withEvent:
– cancelTrackingWithEvent:
tracking property
touchInside property

UIButton Class Reference

Overview

An instance of the UIButton class implements a button on the touch screen. A button intercepts touch events and sends an action message to a target object when tapped. Methods for setting the target and action are inherited from UIControl. This class provides methods for setting the title, image, and other appearance properties of a button. By using these accessors, you can specify a different appearance for each button state.

Tasks

Creating Buttons

+ buttonWithType:

Configuring Button Title

buttonType property
font property
lineBreakMode property
titleShadowOffset property
titleLabel property
reversesTitleShadowWhenHighlighted property
– setTitle:forState:
– setTitleColor:forState:
– setTitleShadowColor:forState:
– titleColorForState:
– titleForState:
– titleShadowColorForState:

Configuring Button Images

adjustsImageWhenHighlighted property
adjustsImageWhenDisabled property
showsTouchWhenHighlighted property
– backgroundImageForState:
– imageForState:
– setBackgroundImage:forState:
– setImage:forState:

Configuring Edge Insets

contentEdgeInsets property
titleEdgeInsets property
imageEdgeInsets property

Getting the Current State

currentTitle property
currentTitleColor property
currentTitleShadowColor property
currentImage property
currentBackgroundImage property
imageView property

Getting Dimensions

– backgroundRectForBounds:
– contentRectForBounds:
– titleRectForContentRect:
– imageRectForContentRect:

UIDatePicker Class Reference

Overview

The UIDatePicker class implements an object that uses multiple rotating wheels to allow users to select dates and times. iPhone examples of a date picker are the Timer and Alarm (Set Alarm) panes of the Clock application. You may also use a date picker as a countdown timer.

When properly configured, a UIDatePicker object sends an action message when a user finishes rotating one of the wheels to change the date or time; the associated control event is UIControlEventValueChanged. A UIDatePicker 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.

UIDatePicker does not inherit from UIPickerView, but it manages a custom picker-view object as a subview.

Tasks

Managing the Date and Calendar

calendar property
date property
locale property
– setDate:animated:
timeZone property

Configuring the Date Picker Mode

datePickerMode property

Configuring Temporal Attributes

maximumDate property
minimumDate property
minuteInterval property
countDownDuration property

UIPageControl Class Reference

Overview

You use the UIPageControl class to create and manage page controls. A page control is a succession of dots centered in the control. Each dot corresponds to a page in the application's document (or other data-model entity), with the white dot indicating the currently viewed page.

For an example of a page control, see the Weather application (with a number of locations configured) or Safari (with a number of tab views set).

When a user taps a page control to move to the next or previous page, the control sends the UIControlEventValueChanged event for handling by the delegate. The delegate can then evaluate the currentPage property to determine the page to display. The page control advances only one page in either direction.

Note: Because of physical factors—namely the size of the device screen and the size and layout of the page indicators—there is a limit of about 20 page indicators on the screen before they are clipped.

Tasks

Managing the Page Navigation

currentPage property
numberOfPages property
hidesForSinglePage property

Updating the Page Display

defersCurrentPageDisplay property
– updateCurrentPageDisplay

Resizing the Control

– sizeForNumberOfPages:

UISegmentedControl Class Reference

Overview

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

A segmented control can display a title (an NSString object) or an image (UIImage object). The UISegmentedControl object automatically resizes segments to fit proportionally within their superview unless they have a specific width set. When you add and remove segments, you can request that the action be animated with sliding and fading effects.

You register the target-action methods for a segmented control using the UIControlEventValueChanged constant as shown below.

```
[segmentedControl addTarget:self

                     action:@selector(action:)

           forControlEvents:UIControlEventValueChanged];
```

How you configure a segmented control can affect its display behavior:

- If you set a segmented control to have a momentary style, a segment doesn't show itself as selected (blue background) when the user touches it. The disclosure button is always momentary and doesn't affect the actual selection.
- Prior to iPhone OS 3.0, if a segmented control has only two segments, then it behaves like a switch—tapping the currently-selected segment causes the other segment to be selected. (On iPhone OS 3.0 and later, tapping the currently-selected segment does not cause the other segment to be selected.)

Tasks

Initializing a Segmented Control

– initWithItems:

Managing Segment Content

– setImage:forSegmentAtIndex:
– imageForSegmentAtIndex:

– setTitle:forSegmentAtIndex:
– titleForSegmentAtIndex:

Managing Segments

– insertSegmentWithImage:atIndex:animated:
– insertSegmentWithTitle:atIndex:animated:
numberOfSegments property
– removeAllSegments
– removeSegmentAtIndex:animated:
selectedSegmentIndex property

Managing Segment Behavior and Appearance

momentary property
segmentedControlStyle property
tintColor property
– setEnabled:forSegmentAtIndex:
– isEnabledForSegmentAtIndex:
– setContentOffset:forSegmentAtIndex:
– contentOffsetForSegmentAtIndex:
– setWidth:forSegmentAtIndex:
– widthForSegmentAtIndex:

UITextField Class Reference

Overview

A UITextField object is a control that displays editable text and sends an action message to a target object when the user presses the return button. You typically use this class to gather small amounts of text from the user and perform some immediate action, such as a search operation, based on that text.

In addition to its basic text-editing behavior, the UITextField class supports the use of overlay views to display additional information (and provide additional command targets) inside the text field boundaries. You can use custom overlay views to display features such as a bookmarks button or search icon. The UITextField class also provides a built-in button for clearing the current text.

A text field object supports the use of a delegate object to handle editing-related notifications. You can use this delegate to customize the editing behavior of the control and provide guidance for when certain actions should occur. For more information on the methods supported by the delegate, see the UITextFieldDelegate protocol.

Managing the Keyboard

When the user taps in a text field, that text field becomes the first responder and automatically asks the system to display the associated keyboard. Because the appearance of the keyboard has the potential to obscure portions of your user interface, it is up to you to make sure that does not happen by repositioning any views that might be obscured. Some system views, like table views, help you by scrolling the first responder into view automatically. If the first responder is at the bottom of the scrolling region, however, you may still need to resize or reposition the scroll view itself to ensure the first responder is visible.

It is your application's responsibility to dismiss the keyboard at the time of your choosing. You might dismiss the keyboard in response to a specific user action, such as the user tapping a particular button in your user interface. You might also configure your text field delegate to dismiss the keyboard when the user presses the "return" key on the keyboard itself. To dismiss the keyboard, send the resignFirstResponder message to the text field that is currently the first responder. Doing so causes the text field object to end the current editing session (with the delegate object's consent) and hide the keyboard.

The appearance of the keyboard itself can be customized using the properties provided by the UITextInputTraits protocol. Text field objects implement this protocol and support the properties it defines. You can use these properties to specify the type of keyboard (ASCII, Numbers, URL, Email, and others) to display. You can also configure the basic

text entry behavior of the keyboard, such as whether it supports automatic capitalization and correction of the text.

Keyboard Notifications

When the system shows or hides the keyboard, it posts several keyboard notifications. These notifications contain information about the keyboard, including its size, which you can use for calculations that involve moving views. Registering for these notifications is the only way to get some types of information about the keyboard. The system delivers the following notifications for keyboard-related events:

- UIKeyboardWillShowNotification
- UIKeyboardDidShowNotification
- UIKeyboardWillHideNotification
- UIKeyboardDidHideNotification

For more information about these notifications, see their descriptions in UIWindow Class Reference. For information about how to show and hide the keyboard, see Text and Web.

Tasks

Accessing the Text Attributes

text property
placeholder property
font property
textColor property
textAlignment property

Sizing the Text Field’s Text

adjustsFontSizeToFitWidth property
minimumFontSize property

Managing the Editing Behavior

editing property
clearsOnBeginEditing property

Setting the View’s Background Appearance

borderStyle property
background property
disabledBackground property

Managing Overlay Views

clearButtonMode property
leftView property
leftViewMode property
rightView property
rightViewMode property

Accessing the Delegate

delegate property

Drawing and Positioning Overrides

– textRectForBounds:
– drawTextInRect:
– placeholderRectForBounds:
– drawPlaceholderInRect:
– borderRectForBounds:
– editingRectForBounds:
– clearButtonRectForBounds:
– leftViewRectForBounds:
– rightViewRectForBounds:

UISlider Class Reference

Overview

A UISlider object is a visual control used to select a single value from a continuous range of values. Sliders are always displayed as horizontal bars. An indicator, or **thumb**, notes the current value of the slider and can be moved by the user to change the setting.

Customizing the Slider's Appearance

The most common way to customize the slider's appearance is to provide custom minimum and maximum value images. 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 small speaker with no sound waves emanating from it for the minimum value and display a large speaker with many sound waves emanating from it for the maximum value.

The bar on which the thumb rides is referred to as the slider's **track**. Slider controls draw the track using two distinct images, which are customizable. 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 in order to provide context as to which end contains the minimum value. For example, the minimum track image typically contains a blue highlight while the maximum track image contains a white highlight. You can assign different pairs of track images to each of control states of the slder. Assigning different images to each state lets you customize the appearance of the slider when it is enabled, disabled, highlighted, and so on.

In addition to customizing the track images, you can also customize the appearance of the thumb itself. Like the track images, you can assign different thumb images to each control state of the slider.

Note: The slider control provides a set of default images for both the track and thumb. If you do not specify any custom images, those images are used automatically.

Tasks

Accessing the Slider's Value

value property

– setValue:animated:

Accessing the Slider's Value Limits

minimumValue property
maximumValue property

Modifying the Slider's Behavior

continuous property

Changing the Slider's Appearance

minimumValueImage property
maximumValueImage property
currentMinimumTrackImage property
– minimumTrackImageForState:
– setMinimumTrackImage:forState:
currentMaximumTrackImage property
– maximumTrackImageForState:
– setMaximumTrackImage:forState:
currentThumbImage property
– thumbImageForState:
– setThumbImage:forState:

Overrides for Subclasses

– maximumValueImageRectForBounds:
– minimumValueImageRectForBounds:
– trackRectForBounds:
– thumbRectForBounds:trackRect:value:

UISwitch Class Reference

Overview

You use the UISwitch class to create and manage the On/Off buttons you see, for example, in the preferences (Settings) for such services as Airplane Mode. These objects are known as switches.

The UISwitch class declares a property and a method to control its on/off state. As with UISlider, when the user manipulates the switch control (“flips” it) a UIControlEventValueChanged event is generated, which results in the control (if properly configured) sending an action message.

The UISwitch class is not customizable.

Tasks

Initializing the Switch Object

– initWithFrame:

Setting the Off/On State

on property
– setOn:animated: