View Controller Catalog for iOS

Developer

Contents

Figures, Tables, and Listings

About View Controllers

View controllers present and manage a hierarchy of views. The UIKit framework includes classes for view controllers you can use to set up many of the common user interaction idioms in iOS. You use these view controllers with any custom view controllers you may need to build your app's user interface. This document describes how to use the view controllers that are provided by the UIKit framework.

At a Glance

Familiarize yourself with the view controllers that you can use in your app. Although you can build an app entirely from custom view controllers, using the provided view controllers reduces the amount of code you have to write and helps you maintain a consistant user experience.

Note This document assumes that you are developing for iOS 5 or later, and that you are are using storyboards and ARC.

Navigation Controllers Manage Stacks of Other View Controllers

A navigation controller is an instance of the `UINavigationController` class that you use as-is in your app. Apps that contain structured content can use navigation controllers to navigate between levels of content. The navigation controller itself manages the display of one or more custom view controllers, each of which manages the data at a specific level in your data hierarchy. The navigation controller also provides controls for determining the current location in this data hierarchy and for navigating back up the hierarchy.

Relevant chapters "Navigation Controllers" (page 11)

Tab Bar Controllers Manage Independent Sets of View Controllers

A tab bar controller is an instance of the `UITabBarControllerDelegate` class that you use as-is in your app. Apps use tab bar controllers to manage multiple distinct interfaces, each of which consists of any number of custom views and view controllers. The tab bar controller also manages interactions with a tab bar view, which the user taps to change the currently selected interface. For example, the iPod app on iPhone and iPod touch uses a tab bar interface where each tab represents a different way of viewing the user's music and media.

Relevant chapters "Tab Bar Controllers" (page 33)

Page View Controllers Manage Paged Display of View Controllers

A page view controller is an instance of the `UIPageViewController` class that you use as-is in your app. Apps can use a page view controller to present a paged view of content. The page view controller itself manages the display of one or more content view controllers, each of which provides a single page of content. The page view controller also provides gesture recognizers that allow the user to navigate through its content.

Relevant chapters "Page View Controllers" (page 46)

Split View Controllers Manage Two Panes of Information

A split view controller is an instance of the `UISplitViewController` class that you use as-is in your app. Apps can use a split view controller to manage two panes of information, where both portions of the interface are themselves managed by view controllers. This is interface is similar to a navigation controller, but it takes advantage of the larger screen size of iPad to present more content at a time.

Relevant chapters "Split View Controllers" (page 52)

Popovers Present Content in a Floating View

The `UIPopoverController` class works with your app's view controllers to present content in a floating view. This interface takes advantage of the larger screen size of iPad to temporarily present a smaller portion of content in the context of a larger body of content.

Relevant chapters "Popovers" (page 55)

View Controllers Are Combined to Build More Sophisticated Interfaces

In all but the simplest apps, it is common for two or more view controllers to work together. Navigation, tab bar, and split view controllers always work in conjunction with other view controllers, and even your custom view controllers may occasionally need to present other view controllers. However, some combinations of view controllers work better than others. Combining view controllers in ways that make sense is important to creating a straightforward, easily navigable user interface.

Relevant chapters "Combined View Controller Interfaces" (page 61)

Prerequisites

View Controller Programming Guide for iOS describes the design patterns and general usage of view controllers on iOS.

iOS App Programming Guide introduces the development process and describes the core architecture.

Developing for the App Store describes the steps for developing apps and submitting them to the App Store.

Tools Workflow Guide for iOS describes how to build, run, debug, and tune your apps on both the simulator and devices.

See Also

For guidance about how to design iOS apps, see *iOS Human Interface Guidelines*.

For information about the view controller classes discussed in this document, see *UIKit Framework Reference*.

Navigation Controllers

A navigation controller manages a stack of view controllers to provide a drill-down interface for hierarchical content. The view hierarchy of a navigation controller is self contained. It is composed of views that the navigation controller manages directly and views that are managed by content view controllers you provide. Each content view controller manages a distinct view hierarchy, and the navigation controller coordinates the navigation between these view hierarchies.

Although a navigation interface consists mostly of your custom content, there are still places where your code must interact directly with the navigation controller object. In addition to telling the navigation controller when to display a new view, you are responsible for configuring the navigation bar—the view at the top of the screen that provides context about the user's place in the navigation hierarchy. You can also provide items for a toolbar that is managed by the navigation controller.

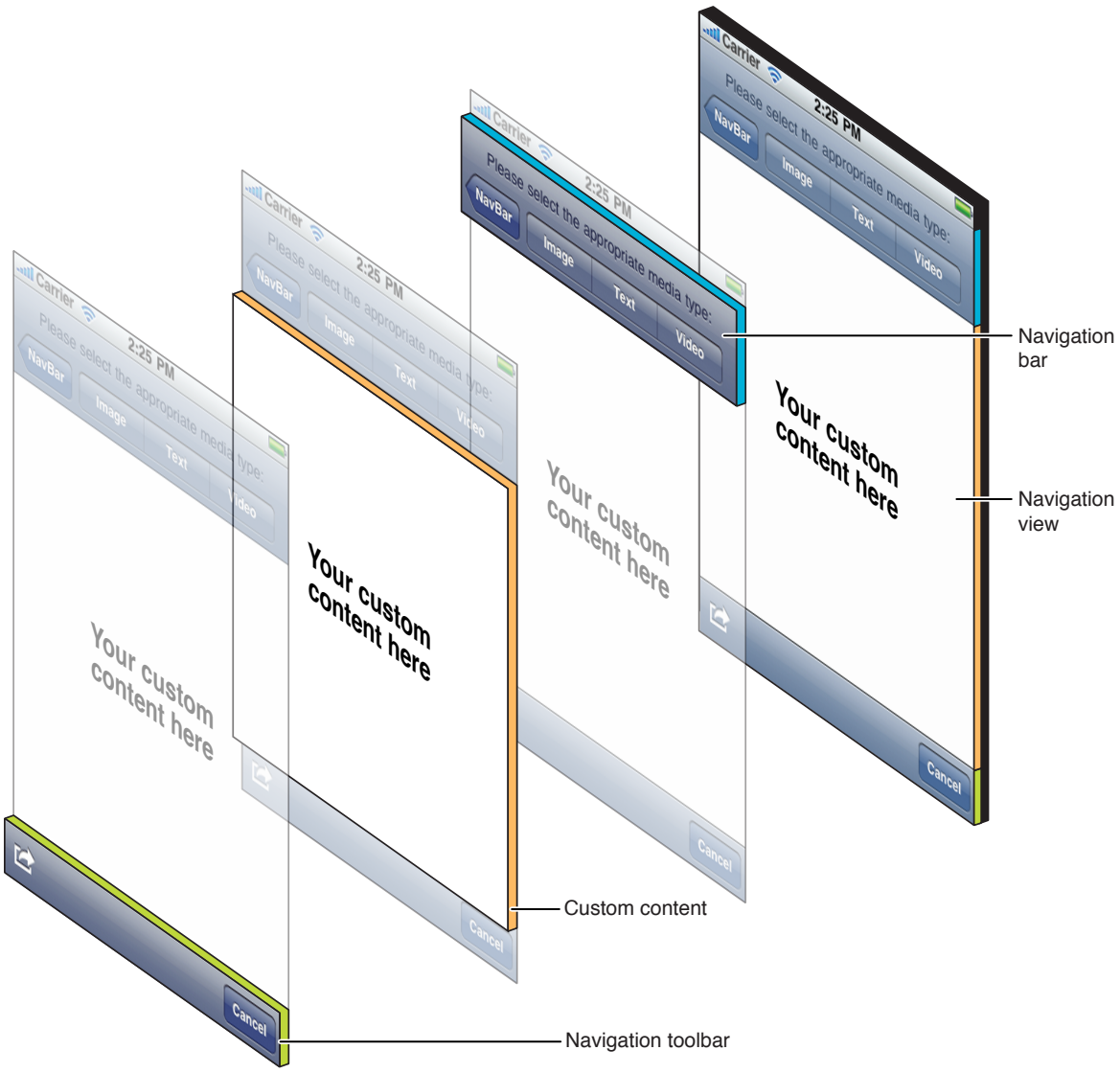
This chapter describes how you configure and use navigation controllers in your app. For information about ways in which you can combine navigation controllers with other types of view controller objects, see "Combined View Controller Interfaces" (page 61).

Anatomy of a Navigation Interface

A navigation controller's primary job is to manage the presentation of your content view controllers, and it is also responsible for presenting some custom views of its own. Specifically, it presents a navigation bar, which contains a back button and some buttons you can customize. A navigation controller can also optionally present a navigation toolbar view and populate it with custom buttons.

Figure 1-1 shows a navigation interface. The navigation view in this figure is the view stored in the navigation controller's `view` property. All of the other views in the interface are part of an opaque view hierarchy managed by the navigation controller.

Figure 1-1 The views of a navigation interface

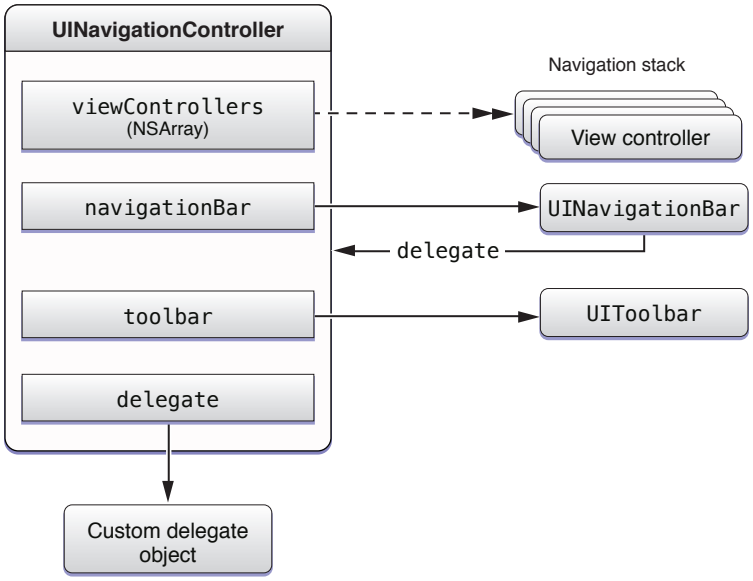


Although the navigation bar and toolbar are customizable views, you must never modify the views in the navigation hierarchy directly. The only way to customize these views is through methods of the `UINavigationController` and `UIViewController` classes. For information on how to customize the contents of the navigation bar, see "Customizing the Navigation Bar Appearance" (page 22). For information about how to display and configure custom toolbar items in a navigation interface, see "Displaying a Navigation Toolbar" (page 29).

The Objects of a Navigation Interface

A navigation controller uses several objects to implement the navigation interface. You are responsible for providing some of these objects and the rest are created by the navigation controller itself. Specifically, you are responsible for providing the view controllers with the content you want to present. If you want to respond to notifications from the navigation controller, you can also provide a delegate object. The navigation controller creates the views—such as the navigation bar and toolbar—that are used for the navigation interface, and it is responsible for managing those views. Figure 1-2 shows the relationship between the navigation controller and these key objects.

Figure 1-2 Objects managed by the navigation controller

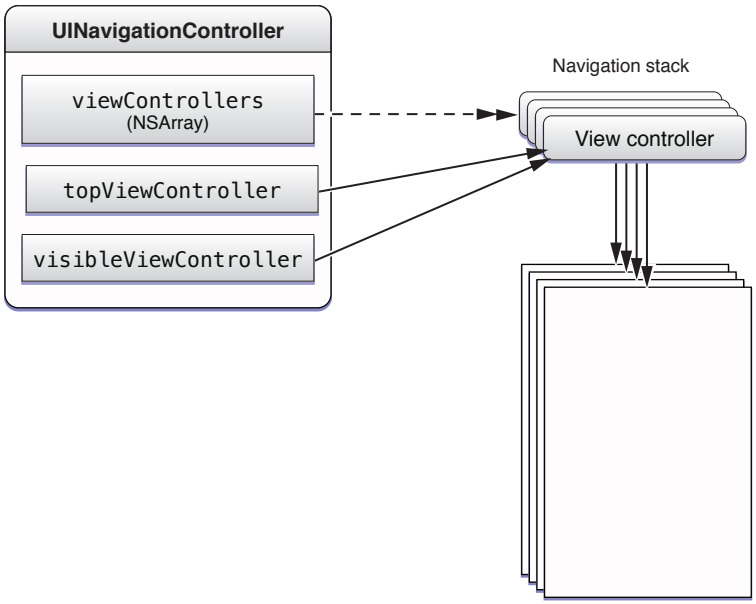


You can modify some aspects of the appearance of the navigation bar and toolbar objects associated with a navigation controller but, otherwise, you should not modify them. The navigation controller alone is responsible for configuring and displaying them. In addition, a navigation controller object automatically assigns itself as the delegate of its `UINavigationBar` object and prevents other objects from changing that relationship.

You can modify the delegate and the other view controllers on the navigation stack. The **navigation stack** is a last-in, first-out collection of custom view controller objects that is managed by the navigation controller. The first item added to the stack becomes the **root view controller** and is never popped off the stack. Additional items can be added to the stack using the methods of the `UINavigationController` class.

Figure 1-3 shows the relevant relationships between the navigation controller and the objects on the navigation stack. (Note that the top view controller and the visible view controller are are not necessarily the same. For example, if you present a view controller modally, the value of the `visibleViewController` property changes to reflect the modal view controller that was presented, but the `topViewController` property does not change.)

Figure 1-3 The navigation stack



Your main interaction with the navigation controller is to respond to user actions by pushing new content view controllers onto the stack or popping content view controllers off of the stack. Each view controller you push on the navigation stack is responsible for presenting some portion of your app's data. Typically, when the user selects an item in the currently visible view, you set up a view controller that has the details for that item and push it onto the navigation stack. For example, when the user selects a photo album, the Photos app pushes a view controller that displays the photos in that album.

The design pattern is that each content view controller in the stack configures and pushes the content view controller that is on top of it in the stack. You should avoid making a view controller depend on being pushed onto the stack by an instance of a specific class. Instead, to pass data back down the stack while popping view controllers, set the view controller that is lower in the stack as a delegate of the one that is above it in the stack.

In most cases, you do not have to pop view controllers off the stack programmatically. Instead, the navigation controller provides a back button on the navigation bar that pops the topmost view controller automatically when the user taps it.

For more information about how to customize the navigation bar, see "Customizing the Navigation Bar Appearance" (page 22). For information about pushing view controllers onto the navigation stack (and removing them later), see "Modifying the Navigation Stack" (page 19). For information on how to customize the contents of the toolbar, see "Displaying a Navigation Toolbar" (page 29).

Creating a Navigation Interface

Before creating a navigation interface, you need to decide how you intend to use a navigation interface. Because it imposes an overarching organization on your data, you should only use it in these specific ways:

- Install it directly as a window's root view controller.
- Install it as the view controller of a tab in a tab bar interface.
- Install it as one of the two root view controllers in a split view interface. (iPad only)
- Present it modally from another view controller.
- Display it from a popover. (iPad only)

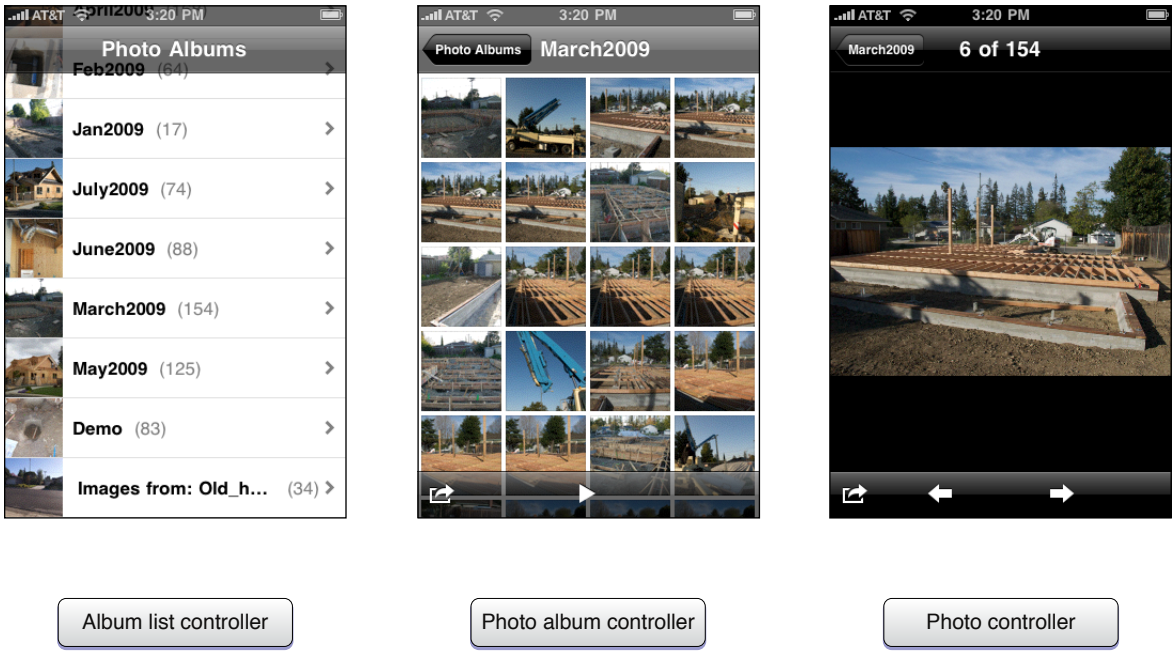
In the first three scenarios, the navigation controller provides a crucial part of your basic interface and stays in place until the app exits. The last two scenarios reflect a more temporary use for navigation controllers, in which case the process for using the navigation controller is identical to the process for other content view controllers. The only difference is that the navigation controller continues to provide additional navigation features not available with a single content view controller. Although the following sections focus on how you create the more permanent types of navigation interface, most of the customization steps and general information apply to all navigation controllers, regardless of how you intend to use them.

Note For a specific example of how to present a navigation controller modally, see "Displaying a Navigation Controller Modally" (page 63).

Defining the Content View Controllers for a Navigation Interface

Every navigation interface has one level of data that represents the root level.This level is the starting point of your interface. For example, the Photos app displays the list of available photo albums at the root level of its data hierarchy. Selecting a photo album then displays the photos in that album and selecting a photo shows a larger version of the photo.

Figure 1-4 Defining view controllers for each level of data



To implement a navigation interface, you must decide what data to present at each level of your data hierarchy. For each level, you must provide a content view controller to manage and present the data at that level. If the presentation at multiple levels is the same, you can create multiple instances of the same view controller class and configure each one to manage its own set of data. For example, the Photos app has three distinct presentation types, as shown in Figure 1-4, so it would need to use three distinct view controller classes.

Each content view controller must provide a way for the user to navigate to the next level of the data hierarchy, except for view controllers managing leaf data. A view controller that displays a list of items can use taps in a given table cell to display the next level of data. For example, when a user selects a photo album from the top-level list, the Photos app creates a new photo album view controller. The new view controller is initialized with enough information about the album for it to present the relevant photos.

For general information and guidance on defining custom view controllers, see "Creating Custom Content View Controllers" in *View Controller Programming Guide for iOS*.

Creating a Navigation Interface Using a Storyboard

If you are creating a new Xcode project, the Master-Detail Application template gives you a navigation controller in the storyboard, set as the first scene.

To create a navigation controller in a storyboard, do the following:

1. Drag a navigation controller from the library.
2. Interface Builder creates a navigation controller and a view controller, and it creates a relationship between them. This relationship identifies the newly created view controller as the root view controller of the navigation controller.
3. Display it as the first view controller by selecting the option Is Initial View Controller in the Attributes inspector (or present the view controller in your user interface in another way.)

Creating a Navigation Interface Programmatically

If you prefer to create a navigation controller programmatically, you may do so from any appropriate point in your code. For example, if the navigation controller provides the root view for your app window, you could create the navigation controller in the `applicationDidFinishLaunching:` method of your application delegate.

When creating a navigation controller, you must do the following:

1. Create the root view controller for the navigation interface.

   This object is the top-level view controller in the navigation stack. The navigation bar displays no back button when its view is displayed and the view controller cannot be popped from the navigation stack.
2. Create the navigation controller, initializing it using the `initWithRootViewController:` method.
3. Set the navigation controller as the root view controller of your window (or otherwise present it in your interface).

Listing 1-1 shows a simple implementation of the `applicationDidFinishLaunching:` method that creates a navigation controller and sets it as the root view controller of the app's main window. The `navigationController` and `window` variables are member variables of the application delegate and the `MyRootViewController` class is a custom view controller class. When the window for this example is displayed, the navigation interface presents the view for the root view controller in the navigation interface.

Listing 1-1 Creating a navigation controller programmatically

```
- (void)applicationDidFinishLaunching:(UIApplication *)application
{
    UIViewController *myViewController = [[MyViewController alloc] init];
    navigationController = [[UINavigationController alloc]
                                initWithRootViewController:myViewController];

    window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    window.rootViewController = navigationController;
    [window makeKeyAndVisible];
}
```

Adopting a Full-Screen Layout for Navigation Views

Typically, a navigation interface displays your custom content in the gap between the bottom of the navigation bar and the top of the toolbar or tab bar (see Figure 1-1 (page 12)). However, a view controller can ask that its view be displayed with a full-screen layout instead. In a full-screen layout, the content view is configured to underlap the navigation bar, status bar, and toolbar as appropriate. This arrangement lets you maximize the visible amount of content for the user and is useful for photo displays or other places where you might want more space.

When determining whether a view should be sized to fill all or most of the screen, a navigation controller considers several factors, including the following:

- Is the underlying window (or parent view) sized to fill the entire screen bounds?
- Is the navigation bar configured to be translucent?
- Is the navigation toolbar (if used) configured to be translucent?
- Is the underlying view controller's `wantsFullScreenLayout` property set to `YES`?

Each of these factors is used to determine the final size of the custom view. The order of the items in the preceding list also reflects the precedence by which each factor is considered. The window size is the first limiting factor; if your app's main window (or the containing parent view in the case of a modally presented view controller) does not span the screen, the views it contains cannot do so either. Similarly, if the navigation bar or toolbar are visible but not translucent, it does not matter if the view controller wants its view to be displayed using a full-screen layout. The navigation controller never displays content under an opaque navigation bar.

If you are creating a navigation interface and want your custom content to span most or all of the screen, here are the steps you should take:

1. Configure the frame of your custom view to fill the screen bounds.

   Be sure to configure the autoresizing attributes of your view as well. The autoresizing attributes ensure that if your view needs to be resized, it adjusts its content accordingly. Alternatively, when your view is resized, you can call the `setNeedsLayout` method of your view to indicate that the position of its subviews should be adjusted.

2. To underlap the navigation bar,set the `translucent` property of your navigation controller to `YES`.
3. To underlap an optional toolbar, set the `translucent` property of the toolbar to `YES`.
4. To underlap the status bar, set the `wantsFullScreenLayout` property of your view controller to `YES`.

When presenting your navigation interface, the window or view to which you add your navigation view must also be sized appropriately. If your app uses a navigation controller as its primary interface, then your main window should be sized to match the screen dimensions. In other words, you should set its size to match the `bounds` property of the `UIScreen` class (instead of the `applicationFrame` property). In fact, for a navigation interface, it is usually better to create your window with the full-screen bounds in all situations because the navigation controller adjusts the size of its views to accommodate the status bar automatically anyway.

If you are presenting a navigation controller modally, the content presented by that navigation controller is limited by the view controller doing the presenting. If that view controller does not want to underlap the status bar, then the modally presented navigation controller is not going to be allowed to underlap the status bar either. In other words, the parent view always has some influence over how its modally presented views are displayed.

For additional information on how to configure the interface to support full-screen layout, see "Creating Custom Content View Controllers" in *View Controller Programming Guide for iOS*.

Modifying the Navigation Stack

You are responsible for creating the objects that reside on the navigation stack. When initializing a navigation controller object, you must provide a content view controller to display the root content of your data hierarchy. You can add or remove view controllers programmatically or in response to user interactions. The navigation controller class provides several options for managing the contents of the navigation stack. These options cover the various scenarios you are likely to encounter in your app. Table 1-1 lists these scenarios and how you respond to them.

Table 1-1 Options for managing the navigation stack

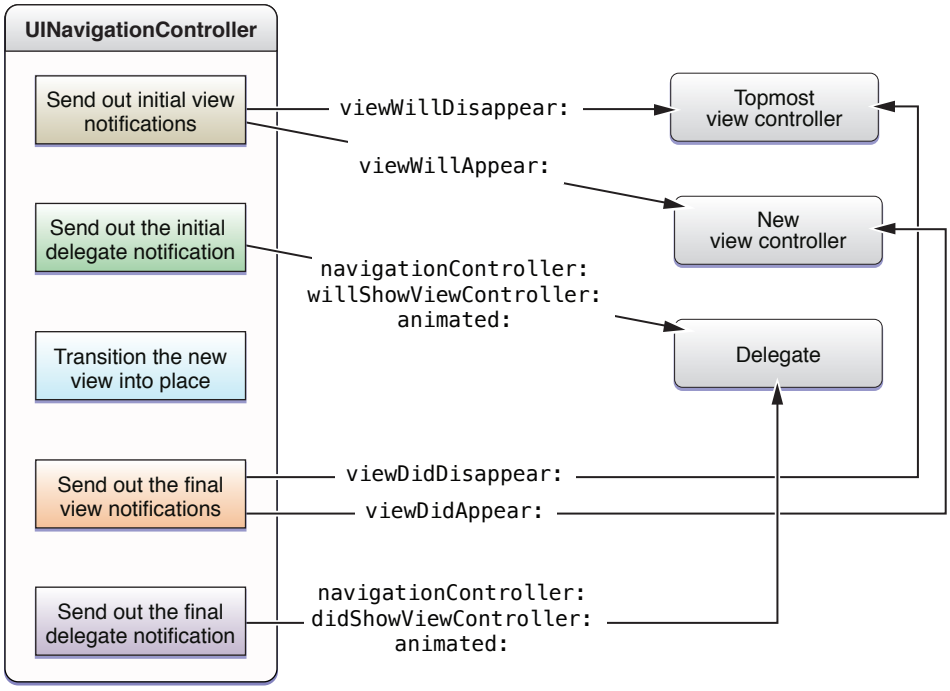
Scenario	Description
Display the next level of hierarchical data.	When the user selects an item displayed by the topmost view controller, you can use a segue or the `pushViewController: animated:` method to push a new view controller onto the navigation stack. The new view controller is responsible for presenting the contents of the selected item.
Back up one level in the hierarchy.	The navigation controller usually provides a back button to remove the topmost view controller from the stack and return to the previous screen. You can also remove the topmost view controller programmatically using the `popViewControllerAnimated:` method.
Restore the navigation stack to a previous state.	When your app launches, you can use the `setViewControllers: animated:` method to restore the navigation controller to a previous state. For example, you would use this method to return the user to the same screen they were viewing when they last quit the app. In order to restore your app to a previous state, you must first save enough state information to recreate the needed view controllers. When the user quits your app, you would need to save some markers or other information indicating the user's position in your data hierarchy. At the next launch time, you would then read this state information and use it to recreate the needed view controllers before calling the `setViewControllers: animated:` method. For more information about saving and restoring state, see "App States and Multitasking" in *iOS App Programming Guide*. You can also use this method to jump to an arbitrary location in your data hierarchy. Jumping to an arbitrary location can easily cause confusion, however, so you should take special care to communicate what is happening clearly to the user.
Return the user to the root view controller.	To return to the top of your navigation interface, use the `popToRootView–ControllerAnimated:` method. This method removes all but the root view controller from the navigation stack.
Back up an arbitrary number of levels in the hierarchy.	To back up more than one level at a time, use the `popToViewController: animated:` method. You might use this method in situations where you use a navigation controller to manage the editing of custom content (rather than presenting content modally). If the user decides to cancel the operation after you have pushed multiple edit screens, you can use this method to remove all of the editing screens at once, rather than one at a time.

When you animate the pushing or popping of view controllers, the navigation controller automatically creates animations that make the most sense. For example, if you pop multiple view controllers off the stack using the `popToViewController:animated:` method, the navigation controller uses an animation only for the topmost view controller. All other intermediate view controllers are dismissed without an animation. If you push or pop an item using an animation, you must wait until the animation is complete before you attempt to push or pop another view controller.

Monitoring Changes to the Navigation Stack

Whenever you push or pop a view controller, the navigation controller sends messages to the affected view controllers. The navigation controller also sends messages to its delegate when its stack changes. Figure 1-5 shows the sequence of events that occurs during a push or pop operation and the corresponding messages that are sent to your custom objects at each stage. The new view controller reflects the view controller that is about to become the topmost view controller on the stack.

Figure 1-5 Messages sent during stack changes



You can use the methods of the navigation controller's delegate to coordinate between content view controllers, for example to update state that is shared between them. If you push or pop multiple view controllers at once, only the view controller that was visible and the view controller that is about to become visible have the

method called. The intermediate view controllers do not get have the method called, unless the chaining happens inside a callback (for example, if your implementation of `viewWillAppear:` calls `pushViewController:animated:`).

You can use the `isMovingToParentViewController` and `isMovingFromParentViewController` methods of `UIViewController` to determine if a view controller is appearing or disappearing as a result of a push or a pop.

Customizing the Navigation Bar Appearance

A navigation bar is a view that manages the controls commonly found in a navigation interface, and it takes on a special role when managed by a corresponding navigation controller object. To ensure consistency, and to reduce the amount of work needed to build navigation interfaces, each navigation controller object creates its own navigation bar and takes on most of the responsibility for managing that bar's content. As needed, the navigation controller interacts with other objects (like your content view controllers) to help in this process.

> **Note** You can also create navigation bars as standalone views and use them however you wish. For more information about using the methods and properties of the `UINavigationBar` class, see *UINavigationBar Class Reference*.

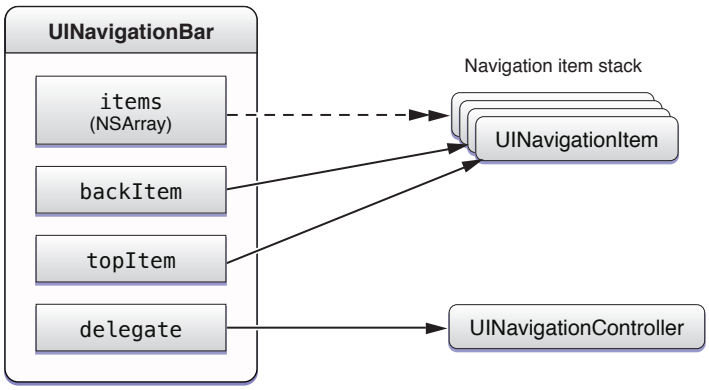
Configuring the Navigation Item Object

The structure of a navigation bar is similar to the structure of a navigation controller in many ways. Like a navigation controller, a navigation bar is a container for content that is provided by other objects. In the case of a navigation bar, the content is provided by one or more `UINavigationItem` objects, which are stored using a stack data structure known as the navigation item stack. Each navigation item provides a complete set of views and content to be displayed in the navigation bar.

Figure 1-6 shows some of the key objects related to a navigation bar at runtime. The owner of the navigation bar (whether it is a navigation controller or your custom code) is responsible for pushing items onto the stack and popping them off as needed. In order to provide proper navigation, the navigation bar maintains pointers

to select objects in the stack. Although most of the navigation bar's content is obtained from the topmost navigation item, a pointer to the back item is maintained so that a back button (with the title of the preceding item) can be created.

Figure 1-6 The objects associated with a navigation bar



Important When used in conjunction with a navigation controller, the delegate of a navigation bar is always set to the owning navigation controller object. Attempting to change the delegate raises an exception.

When used in a navigation interface, each content view controller in the navigation stack provides a navigation item as the value of its `navigationItem` property. The navigation stack and the navigation item stack are always parallel: for each content view controller on the navigation stack, its navigation item is in the same position in the navigation item stack.

A navigation bar has three primary positions for placing items: left, right, and center. Table 1-2 lists the properties of the `UINavigationItem` class that are used to configure each of these positions. When configuring a navigation item for use with a navigation controller, be aware that custom controls in some positions may be ignored in favor of the expected controls. The description of each position includes information about how your custom objects are used.

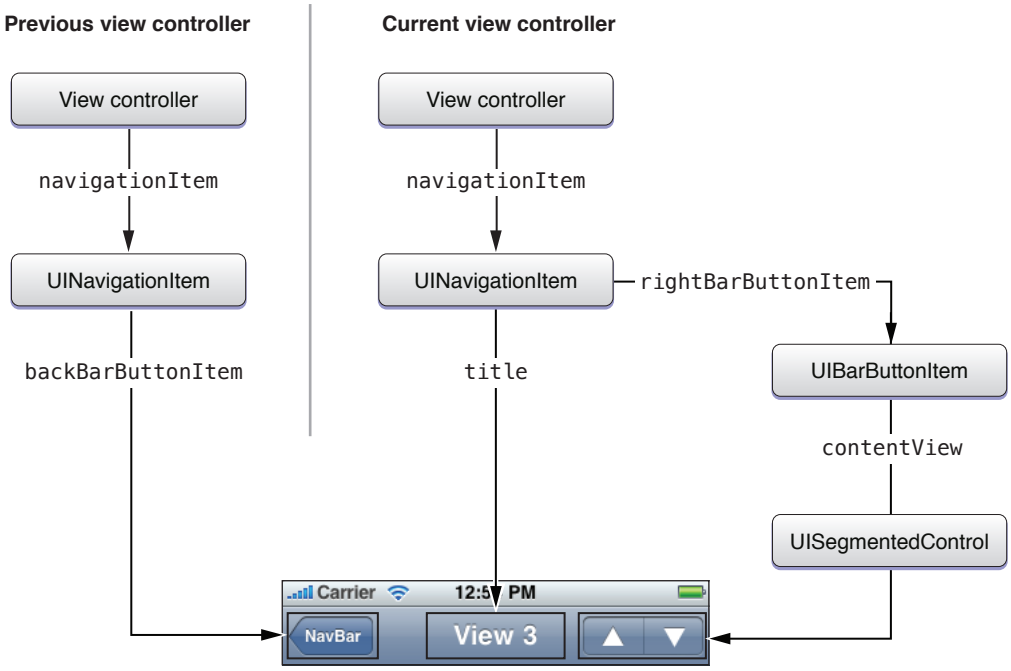
Table 1-2 Item positions on a navigation bar

Position	Property	Description
Left	`backBarButtonItem` `leftBarButtonItem`	In a navigation interface, the navigation controller assigns a Back button to the left position by default. To get the default Back button provided by the navigation controller, get the value of the `backBarButtonItem` property. To assign a custom button or view to the left position, and thereby replace the default Back button, assign a `UIBarButtonItem` object to the `leftBarButtonItem` property.

Position	Property	Description
Center	`titleView`	In a navigation interface, the navigation controller displays a custom view with the title of your content view controller by default. You can replace this view as desired with a custom view of your choosing. If you do not provide a custom title view, the navigation bar displays a custom view with an appropriate title string. The title string is obtained from the navigation item by default, or from the view controller if the navigation item does not provide an appropriate title.
Right	`rightBarButtonItem`	This position is open by default. It is typically used to place buttons for editing or modifying the current screen. You can also place custom views here by wrapping the view in a `UIBarButtonItem` object.

Figure 1-7 shows how the contents of the navigation bar are assembled for a navigation interface. The navigation item associated with the current view controller provides the content for the center and right positions of the navigation bar. The navigation item for the previous view controller provides the content for the left position. Although the left and right items require you to specify a `UIBarButtonItem` object, you can wrap a view in a bar button item as shown in the figure. If you do not provide a custom title view, the navigation item creates one for you using the title of the current view controller.

Figure 1-7 Navigation bar structure



Showing and Hiding the Navigation Bar

When a navigation bar is used in conjunction with a navigation controller, you always use the `setNavigationBarHidden:animated:` method of `UINavigationController` to show and hide the navigation bar. You must never hide the navigation bar by modifying the `UINavigationBar` object's `hidden` property directly. In addition to showing or hiding the bar, using the navigation controller method gives you more sophisticated behaviors for free. Specifically, if a view controller shows or hides the navigation bar in its `viewWillAppear:` method, the navigation controller animates the appearance or disappearance of the bar to coincide with the appearance of the new view controller.

Because the user needs the back button on the navigation bar to navigate back to the previous screen, you should never hide the navigation bar without giving the user some way to get back to the previous screen. The most common way to provide navigation support is to intercept touch events and use them to toggle the visibility of the navigation bar. For example, the Photos app does this when a single image is displayed full screen. You could also detect swipe gestures and use them to pop the current view controller off the stack, but such a gesture is less discoverable than simply toggling the navigation bar's visibility.

Modifying the Navigation Bar Object Directly

In a navigation interface, a navigation controller owns its `UINavigationBar` object and is responsible for managing it. It is not permissible to change the navigation bar object or modify its bounds, frame, or alpha values directly. However, there are a few properties that you can modify, including the following:

- `barStyle` property
- `translucent` property
- `tintColor` property

Figure 1-8 shows how the `barStyle` and `translucent` properties affect the appearance of the navigation bar. For translucent styles, it is worth noting that if the main view of the underlying view controller is a scroll view, the navigation bar automatically adjusts the content inset value to allow content to scroll out from under the navigation bar. It does not make this adjustment for other types of views.

Figure 1-8 Navigation bar styles

If you want to show or hide the entire navigation bar, you should similarly use the `setNavigationBarHidden:animated:` method of the navigation controller rather than modify the navigation bar directly. For more information about showing and hiding the navigation bar, see "Showing and Hiding the Navigation Bar" (page 25).

Using Custom Buttons and Views as Navigation Items

To customize the appearance of the navigation bar for a specific view controller, modify the attributes of its associated `UINavigationItem` object. You can get the navigation item for a view controller from its `navigationItem` property. The view controller does not create its navigation item until you request it, so you should ask for this object only if you plan to install the view controller in a navigation interface.

If you choose not to modify the navigation item for your view controller, the navigation item provides a set of default objects that should suffice in many situations. Any customizations you make take precedence over the default objects.

For the topmost view controller, the item that is displayed on the left side of the navigation bar is determined using the following rules:

- If you assign a custom bar button item to the `leftBarButtonItem` property of the topmost view controller's navigation item, that item is given the highest preference.
- If you do not provide a custom bar button item and the navigation item of the view controller one level down on the navigation stack has a valid item in its `backBarButtonItem` property, the navigation bar displays that item.

- If a bar button item is not specified by either of the view controllers, a default back button is used and its title is set to the value of the title property of the previous view controller—that is, the view controller one level down on the navigation stack. (If the topmost view controller is the root view controller, no default back button is displayed.)

For the topmost view controller, the item that is displayed in the center of the navigation bar is determined using the following rules:

- If you assign a custom view to the `titleView` property of the topmost view controller's navigation item, the navigation bar displays that view.
- If no custom title view is set, the navigation bar displays a custom view containing the view controller's title. The string for this view is obtained from the `title` property of the view controller's navigation item. If the value of that property is `nil`, the string from the `title` property of the view controller itself is used.

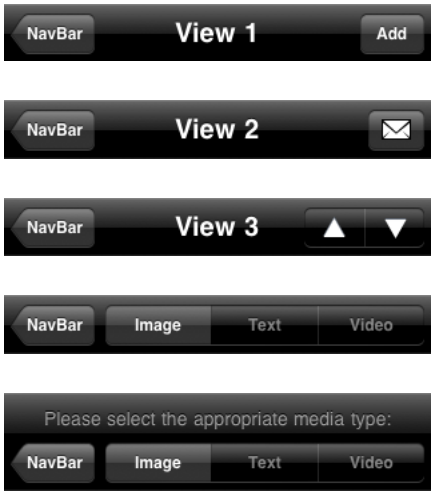
For the topmost view controller, the item that is displayed on the right side of the navigation bar is determined using the following rules:

- If the new top-level view controller has a custom right bar button item, that item is displayed. To specify a custom right bar button item, set the `rightBarButtonItem` property of the navigation item.
- If no custom right bar button item is specified, the navigation bar displays nothing on the right side of the bar.

To add custom prompt text above the navigation bar controls, assign a value to the `prompt` property of the navigation item.

Figure 1-9 shows various navigation bar configurations, including several that use custom views and prompts. The navigation bars in this figure are from the sample project *NavBar*.

Figure 1-9 Custom buttons in the navigation bar



Custom right bar button item

Custom right bar button with an image

Custom right bar button with a view

Custom title view

Custom title view and prompt

Listing 1-2 shows the code from the NavBar app that would be required to create the third navigation bar in Figure 1-9, which is the navigation bar containing the right bar button item with a custom view. Because it is in the right position on the navigation bar, you must wrap the custom view with a `UIBarButtonItem` object before assigning it to the `rightBarButtonItem` property.

Listing 1-2 Creating custom bar button items

```
// View 3 - Custom right bar button with a view
UISegmentedControl *segmentedControl = [[UISegmentedControl alloc] initWithItems:
                                        [NSArray arrayWithObjects:
                                            [UIImage imageNamed:@"up.png"],
                                            [UIImage imageNamed:@"down.png"],
                                            nil]];

[segmentedControl addTarget:self action:@selector(segmentAction:)
forControlEvents:UIControlEventValueChanged];
segmentedControl.frame = CGRectMake(0, 0, 90, kCustomButtonHeight);
segmentedControl.segmentedControlStyle = UISegmentedControlStyleBar;
segmentedControl.momentary = YES;
```

```
defaultTintColor = segmentedControl.tintColor;    // Keep track if this if you
need it later.

UIBarButtonItem *segmentBarItem = [[UIBarButtonItem alloc]
initWithCustomView:segmentedControl];

self.navigationItem.rightBarButtonItem = segmentBarItem;
```

Configuring your view controller's navigation item programmatically is the most common approach for most apps. Although you could create bar button items using Interface Builder, it is often much simpler to create them programmatically. You should create the items in the `viewDidLoad` method of your view controller.

Using Edit and Done Buttons

Views that support in-place editing can include a special type of button in their navigation bar that allows the user to toggle back and forth between display and edit modes. The `editButtonItem` method of `UIViewController` returns a preconfigured button that when pressed toggles between an Edit and Done button and calls the view controller's `setEditing:animated:` method with appropriate values. To add this button to your view controller's navigation bar, you would use code similar to the following:

```
myViewController.navigationItem.rightBarButtonItem = [myViewController
editButtonItem];
```

If you include this button in your navigation bar, you must also override your view controller's `setEditing:animated:` method and use it to adjust your view hierarchy. For more information on implementing this method, see "Creating Custom Content View Controllers" in *View Controller Programming Guide for iOS*.

Displaying a Navigation Toolbar

In iOS 3.0 and later, a navigation interface can display a toolbar and populate it with items provided by the currently visible view controller. The toolbar itself is managed by the navigation controller object. Supporting a toolbar at this level is necessary in order to create smooth transitions between screens. When the topmost view controller on the navigation stack changes, the navigation controller animates changes between different sets of toolbar items. It also creates smooth animations in cases where you want to toggle the visibility of the toolbar for a specific view controller.

To configure a toolbar for your navigation interface, you must do the following:

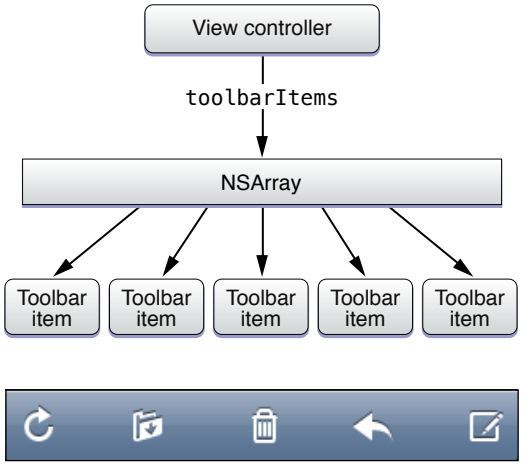
- Show the toolbar by setting the `toolbarHidden` property of the navigation controller object to `NO`.

- Assign an array of `UIBarButtonItem` objects to the `toolbarItems` property of each of your content view controllers, as described in "Specifying the Toolbar Items" (page 30).

  If you do not want to show a toolbar for a particular content view controller, you can hide the toolbar as described in "Showing and Hiding the Toolbar" (page 32).

Figure 1-10 shows an example of how the objects you associate with your content view controller are reflected in the toolbar. Items are displayed in the toolbar in the same order they are provided in the array. The array can include all types of bar button items, including fixed and flexible space items, system button items, or any custom button items you provide. In this example, the five items are all button items from the Mail app.

Figure 1-10 Toolbar items in a navigation interface



Specifying the Toolbar Items

When configuring bar button items, remember to associate an appropriate target and action with the button. The target and action information is what you use to respond to taps in the toolbar. In most cases, the target should be the view controller itself, since it is responsible for providing the toolbar items. Figure 1-11 shows a sample toolbar that places a segmented control in the center of the toolbar and

To specify the toolbar items using a storyboard:

1. Drag a toolbar from the library.
2. Add two flexible space bar button items to the toolbar, by dragging them from the library.
3. Add a segmented control from the library, between the flexible space bar buttons, by dragging it from the library.

Use the Inspector to configure the segmented control.

Figure 1-11 A segmented control centered in a toolbar

Listing 1-3 shows the code needed to specify the toolbar items programmatically. You would implement the method in your view controller and call it at initialization time.

Listing 1-3 Configuring a toolbar with a centered segmented control

```
- (void)configureToolbarItems
{
   UIBarButtonItem *flexibleSpaceButtonItem = [[UIBarButtonItem alloc]

initWithBarButtonSystemItem:UIBarButtonSystemItemFlexibleSpace
                        target:nil action:nil];

   // Create and configure the segmented control
   UISegmentedControl *sortToggle = [[UISegmentedControl alloc]
                        initWithItems:[NSArray arrayWithObjects:@"Ascending",
                                            @"Descending", nil]];
   sortToggle.segmentedControlStyle = UISegmentedControlStyleBar;
   sortToggle.selectedSegmentIndex = 0;
   [sortToggle addTarget:self action:@selector(toggleSorting:)
               forControlEvents:UIControlEventValueChanged];

   // Create the bar button item for the segmented control
   UIBarButtonItem *sortToggleButtonItem = [[UIBarButtonItem alloc]
                                     initWithCustomView:sortToggle];

   // Set our toolbar items
   self.toolbarItems = [NSArray arrayWithObjects:
                         flexibleSpaceButtonItem,
                         sortToggleButtonItem,
                         flexibleSpaceButtonItem,
                         nil];
```

```
}
```

In addition to setting toolbar items during initialization, a view controller can also change its existing set of toolbar items dynamically using the `setToolbarItems:animated:` method. This method is useful for situations where you want to update the toolbar commands to reflect some other user action. For example, you could use it to implement a set of hierarchical toolbar items, whereby tapping a button on the toolbar displays a set of related child buttons.

Showing and Hiding the Toolbar

To hide the toolbar for a specific view controller, set the `hidesBottomBarWhenPushed` property of that view controller to `YES`. When the navigation controller encounters a view controller with this property set to `YES`, it generates an appropriate transition animation whenever the view controller is pushed onto (or removed from) the navigation stack.

If you want to hide the toolbar sometimes (but not always), you can call the `setToolbarHidden:animated:` method of the navigation controller at any time. A common way to use this method is to combine it with a call to the `setNavigationBarHidden:animated:` method to create a temporary full-screen view. For example, the Photos app toggles the visibility of both bars when it is displaying a single photo and the user taps the screen.

Tab Bar Controllers

You use tab bar controller to organize your app into one or more distinct modes of operation. The view hierarchy of a tab bar controller is self contained. It is composed of views that the tab bar controller manages directly and views that are managed by content view controllers you provide. Each content view controller manages a distinct view hierarchy, and the tab bar controller coordinates the navigation between the view hierarchies.

This chapter describes how you configure and use tab bar controllers in your app. For information about ways in which you can combine tab bar controllers with other types of view controller objects, see "Combined View Controller Interfaces" (page 61).

Anatomy of a Tab Bar Interface

A tab bar interface is useful in situations where you want to provide different perspectives on the same set of data or in situations where you want to organize your app along functional lines. The key component of a tab bar interface is the presence of a tab bar view along the bottom of the screen. This view is used to initiate the navigation between your app's different modes and can also convey information about the state of each mode.

The manager for a tab bar interface is a tab bar controller object. The tab bar controller creates and manages the tab bar view and also manages the view controllers that provide the content view for each mode. Each content view controller is designated as the view controller for one of the tabs in the tab bar view. When a tab is tapped by the user, the tab bar controller object selects the tab and displays the view associated with the corresponding content view controller.

Figure 2-1 shows the tab bar interface implemented by the Clock app. The tab bar controller has its own container view, which encompasses all of the other views, including the tab bar view. The custom content is provided by the view controller of the selected tab.

Figure 2-1 The views of a tab bar interface

When a tab bar view is part of a tab bar interface, it must not be modified. In a tab bar interface, the tab bar view is considered to be part of a private view hierarchy that is owned by the tab bar controller object. If you do need to change the list of active tabs, you must always do so using the methods of the tab bar controller itself. For information on how to modify a tab bar interface at runtime, see "Managing Tabs at Runtime" (page 40).

> **Note** You can also create tab bars as standalone views and use them however you wish. For more information about using the methods and properties of the `UITabBar` class, see *UITabBar Class Reference*.

The Objects of a Tab Bar Interface

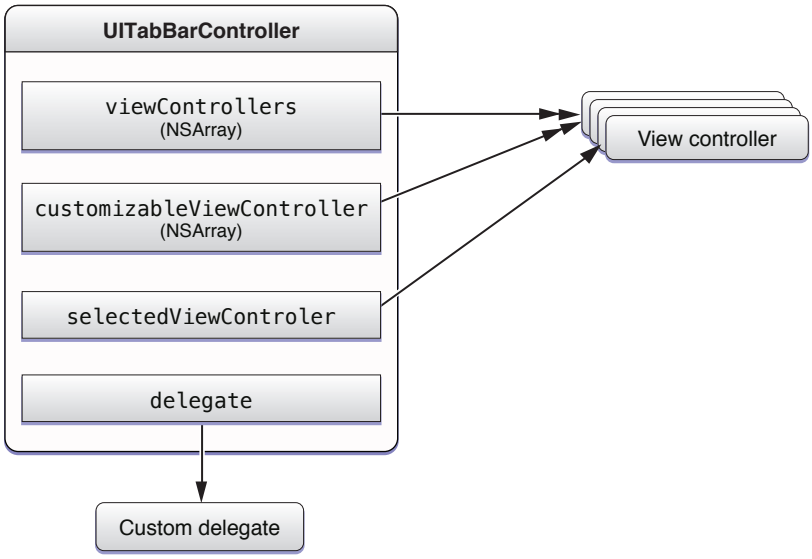
A standard tab bar interface consists of the following objects:

- A `UITabBarController` object

- One content view controller object for each tab
- An optional delegate object

Figure 2-2 shows the relationship of the tab bar controller to its associated view controllers. Each view controller in the tab bar controller's `viewControllers` property is a view controller for a corresponding tab in the tab bar.

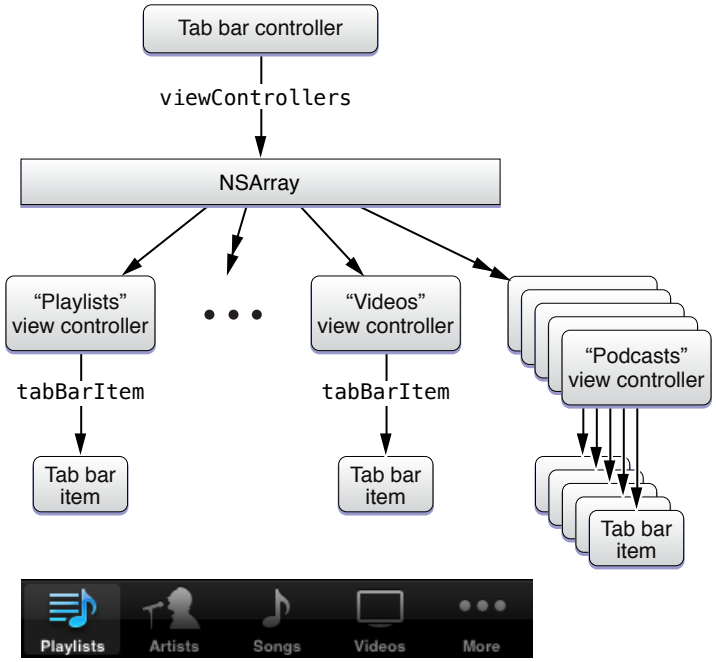
Figure 2-2 A tab bar controller and its associated view controllers



If you add more than five items to the `viewControllers` property, the tab bar controller automatically inserts a special view controller (called the **More view controller**) to handle the display of the additional items. The More view controller provides a custom interface that lists the additional view controllers in a table, which can expand to accommodate any number of view controllers. The More view controller cannot be customized or selected and does not appear in any of the view controller lists managed by the tab bar controller. It appears automatically when it is needed and is separate from your custom content. You can get a reference to it though by accessing the `moreNavigationController` property of `UITabBarController`.

You do not modify the tab bar view directly. The tab bar controller object assembles the contents of the tab bar from the `UITabBarItem` objects provided by your content view controllers. Figure 2-3 shows the relationship between the tab bar controller, view controllers, and the tab bar item objects in the iPod app. Because there are more view controllers than can be displayed at once, tab bar items from only the first four view controllers are displayed. The final tab bar item is provided by the More view controller.

Figure 2-3 Tab bar items of the iPod app



Because tab bar items are used to configure the tab bar, you must configure the tab bar item of each view controller before displaying the tab bar interface. If you are using Interface Builder to assemble your interface, you can specify the title and image as described in "Creating a Tab Bar Interface Using a Storyboard" (page 38). If you are creating your tab bar interface programmatically, you must create a new `UITabBarItem` object for each of your content view controllers, as described in "Creating a Tab Bar Interface Programmatically" (page 39).

A tab bar controller also supports an optional delegate object that can be used to respond to tab bar selections and customizations. For information about responding to delegate-related messages, see "Managing Tabs at Runtime" (page 40).

Creating a Tab Bar Interface

Before creating a tab bar interface, you need to decide how you intend to use a tab bar interface. Because it imposes an overarching organization on your data,you should use one only in these specific ways:

- Install it directly as a window's root view controller.
- Install it as one of the two view controllers in a split view interface. (iPad only)
- Present it modally from another view controller.
- Display it from a popover. (iPad only)

Installing a tab bar interface in your app's main window is by far the most common way to use it. In such a scenario, the tab bar interface provides the fundamental organizing principle for your app's data, with each tab leading the user to a distinct part of the app. Because it provides access to your entire app, it must be the root view controller of the window.

It is also possible to present a tab bar controller modally if a very specific need makes doing so worthwhile. For example, you could present a tab bar controller modally in order to edit some complex data set that had several distinct sets of options. Because a modal view fills all or most of the screen (depending on the device), the presence of the tab bar would simply reflect the choices available for viewing or editing the modally presented data. Avoid using a tab bar in this way if a simpler design approach is available.

Defining the Content View Controllers for a Tab Bar Interface

Because each mode of a tab bar interface is separate from all other modes, the view controller for each tab defines the content for that tab. Thus, the view controller you choose for each tab should reflect the needs of that particular mode of operation. If you need to present a relatively rich set of data, you could install a navigation controller to manage the navigation through that data. If the data being presented is simpler, you could install a content view controller with a single view.

Figure 2-4 shows several screens from the Clock app. The World Clock tab uses a navigation controller primarily so that it can present the buttons it needs to edit the list of clocks. The Stopwatch tab requires only a single screen for its entire interface and therefore uses a single view controller. The Timer tab uses a custom view controller for the main screen and presents an additional view controller modally when the When Timer Ends button is tapped.

Figure 2-4 Tabs of the Clock app

The tab bar controller handles all of the interactions associated with presenting the content view controllers, so there is very little you have to do with regard to managing tabs or the view controllers in them. Once displayed, your content view controllers should simply focus on presenting their content.

For general information and guidance on defining custom view controllers, see "Creating Custom Content View Controllers" in *View Controller Programming Guide for iOS*.

Creating a Tab Bar Interface Using a Storyboard

If you are creating a new Xcode project, the Tabbed Application template gives you a tab bar controller in the storyboard, set as the first scene.

To create a tab bar controller in a storyboard, do the following:

1. Drag a tab bar controller from the library.
2. Interface Builder creates a tab bar controller and two view controllers, and it creates relationships between them. These relationships identify each of the newly created view controllers as the view controller for one tab of the tab bar controller.

3. Display it as the first view controller by selecting the option Is Initial View Controller in the Attributes inspector (or present the view controller in your user interface in another way.)

Creating a Tab Bar Interface Programmatically

If you prefer to create a tab bar controller programmatically, the most appropriate place to do so is in the `applicationDidFinishLaunching:` method of your application delegate. Because a tab bar controller usually provides the root view of your app's window, you need to create it immediately after launch and before you show the window. The steps for creating a tab bar interface are as follows:

1. Create a new `UITabBarController` object.
2. Create a content view controller for each tab.
3. Add the view controllers to an array and assign that array to your tab bar controller's `viewControllers` property.
4. Set the tab bar controller as the root view controller of your window (or otherwise present it in your interface).

Listing 2-1 shows the basic code needed to create and install a tab bar controller interface in the main window of your app. This example creates only two tabs, but you could create as many tabs as needed by creating more view controller objects and adding them to the `controllers` array. You need to replace the custom view controller names `MyViewController` and `MyOtherViewController` with classes from your own app.

Listing 2-1 Creating a tab bar controller from scratch

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
    tabBarController = [[UITabBarController alloc] init];

    MyViewController* vc1 = [[MyViewController alloc] init];
    MyOtherViewController* vc2 = [[MyOtherViewController alloc] init];

    NSArray* controllers = [NSArray arrayWithObjects:vc1, vc2, nil];
    tabBarController.viewControllers = controllers;

     window.rootViewController = tabBarController;
}
```

Creating a Tab Bar Item Programmatically

For each content view controller in your tab bar interface, you must provide a `UITabBarItem` object with the image and text to be displayed in the corresponding tab. You can associate tab bar items with your view controllers at any time before displaying your tab bar interface. You do this by assigning the tab bar item to the `tabBarItem` property of the corresponding view controller. The recommended time to create a tab bar item is during the initialization of the view controller itself, but this is typically feasible only for custom view controllers. You can also create and initialize the view controller object, create the tab bar item, and then make the association.

Listing 2-2 shows an example of how to create a tab bar item for a custom view controller. Because it is associated with a custom view controller, the view controller creates the tab bar item itself as part of its initialization process. In this example, the tab bar item includes both a custom image (which is stored in the application bundle) and a custom title string.

Listing 2-2 Creating the view controller's tab bar item

```
UIImage* anImage = [UIImage imageNamed:@"MyViewControllerImage.png"];
UITabBarItem* theItem = [[UITabBarItem alloc] initWithTitle:@"Home" image:anImage
 tag:0];
```

Managing Tabs at Runtime

After creating your tab bar interface, there are several ways to modify it and respond to changes in your app. You can add and remove tabs or use a delegate object to prevent the selection of tabs based on dynamic conditions. You can also add badges to individual tabs to call the user's attention to that tab.

Adding and Removing Tabs

If the number of tabs in your tab bar interface can change dynamically, you can make the appropriate changes at runtime as needed. You change the tabs at runtime in the same way you specify tabs at creation time, by assigning the appropriate set of view controllers to your tab bar controller. If you are adding or removing tabs in a way that might be seen by the user, you can animate the tab changes using the `setViewControllers:animated:` method.

Listing 2-3 shows a method that removes the currently selected tab in response to a tap in a specific button in the same tab. This method is implemented by the view controller for the tab. You might use something similar in your own code if you want to remove a tab that is no longer needed. For example, you could use it to remove a tab containing some user-specific data that needs to be entered only once.

Listing 2-3 Removing the current tab

```
- (IBAction)processUserInformation:(id)sender
{
   // Call some app-specific method to validate the user data.
   // If the custom method returns YES, remove the tab.
   if ([self userDataIsValid])
   {
      NSMutableArray* newArray = [NSMutableArray
arrayWithArray:self.tabBarController.viewControllers];
      [newArray removeObject:self];

      [self.tabBarController setViewControllers:newArray animated:YES];
   }
}
```

Preventing the Selection of Tabs

If you need to prevent the user from selecting a tab, you can do so by providing a delegate object and implementing the `tabBarController:shouldSelectViewController:` method on that object. Preventing the selection of tabs should be done only on a temporary basis, such as when a tab does not have any content. For example, if your app requires the user to provide some specific information, such as a login name and password, you could disable all tabs except the one that prompts the user for the required information. Listing 2-4 shows an example of what such a method looks like. The `hasValidLogin` method is a custom method that you implement to validate the provided information.

Listing 2-4 Preventing the selection of tabs

```
- (BOOL)tabBarController:(UITabBarController *)aTabBar
         shouldSelectViewController:(UIViewController *)viewController
{
   if (![self hasValidLogin] && (viewController != [aTabBar.viewControllers
objectAtIndex:0]) )
   {
      // Disable all but the first tab.
      return NO;
   }
```

```
    return YES;
}
```

Monitoring User-Initiated Tab Changes

There are two types of user-initiated changes that can occur on a tab bar:

- The user can select a tab.
- The user can rearrange the tabs.

Both types of changes are reported to the tab bar controller's delegate, which is an object that conforms to the `UITabBarControllerDelegate` protocol. You might provide a delegate to keep track of user changes and update your app's state information accordingly. However, you should not use these notifications to perform work that would normally be handled by the view controllers being hidden and shown. For example, you would not use your tab bar controller delegate to change the appearance of the status bar to match the style of the currently selected view. Visual changes of that nature are best handled by your content view controllers.

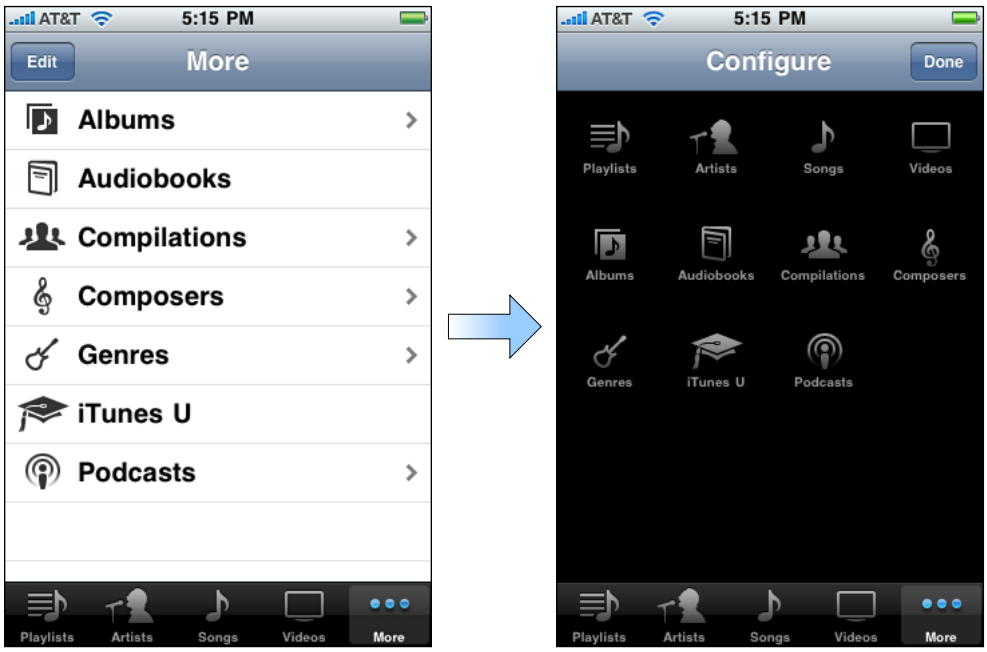
For more information about the methods of the `UITabBarControllerDelegate` protocol and how you use them, see *UITabBarControllerDelegate Protocol Reference* .

Preventing the Customization of Tabs

The More view controller provides built-in support for the user to modify the items displayed in the tab bar. For apps with lots of tabs, this support allows the user to pick and choose which screens are readily accessible and which require additional navigation to reach. The left side of Figure 2-5 shows the More selection screen

displayed by the iPod app. When the user taps the Edit button in the upper-left corner of this screen, the More controller automatically displays the configuration screen shown on the right. From this screen, the user can replace the contents of the tab bar by dragging new items to it.

Figure 2-5 Configuring the tab bar of the iPod app



While it is a good idea to let the user rearrange tabs most of the time, there may be situations where you might not want the user to remove specific tabs from the tab bar or place specific tabs on the tab bar. In these situations, you can assign an array of view controller objects to the `customizableViewControllers` property. This array should contain the subset of view controllers that it is all right to rearrange. View controllers not in the array are not displayed on the configuration screen and cannot be removed from the tab bar if they are already there.

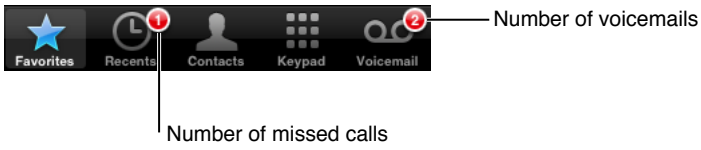
Important Adding or removing view controllers in your tab bar interface also resets the array of customizable view controllers to the default value, allowing all view controllers to be customized again. Therefore, if you make modifications to the `viewControllers` property (either directly or by calling the `setViewControllers:animated:` method) and still want to limit the customizable view controllers, you must also update the array of objects in the `customizableViewControllers` property.

Changing a Tab's Badge

The appearance of a tab in a tab bar interface normally does not change, except when it is selected. To call attention to a specific tab, perhaps because there is new content on that tab for the user to look at, you can use a badge.

A badge is a small red marker displayed in the corner of the tab. Inside the badge is some custom text that you provide. Typically, badges contain numerical values reflecting the number of new items available on the tab, but you can also specify very short character strings too. Figure 2-6 shows badges for tabs in the Phone app.

Figure 2-6 Badges for tab bar items



To assign a badge to a tab, assign a non-nil value to the `badgeValue` property of the corresponding tab bar item. Listing 2-5 shows an example of how a view controller that displays the number of new items in its badge might set the badge value.

Listing 2-5 Setting a tab's badge

```
if (numberOfNewItems == 0)
   self.tabBarItem.badgeValue = nil;
else
   self.tabBarItem.badgeValue = [NSString stringWithFormat:@"%d", numberOfNewItems];
```

It is up to you to decide when to display badge values and to update the badge value at the appropriate times. However, if your view controller contains a property with such a value, you can use key-value observing (KVO) notifications to detect changes to the value and update the badge accordingly. For information about setting up and handling KVO notifications, see *Key-Value Observing Programming Guide*.

Tab Bar Controllers and View Rotation

Tab bar controllers support a portrait orientation by default and do not rotate to a landscape orientation unless all of the contained view controllers support such an orientation. When a device orientation change occurs, the tab bar controller queries its array of view controllers. If any one of them does not support the orientation, the tab bar controller does not change its orientation.

Tab Bars and Full-Screen Layout

Tab bar controllers support full-screen layout differently from the way most other controllers support it. You can still set the `wantsFullScreenLayout` property of your content view controller to `YES` if you want its view to underlap the status bar or a navigation bar (if present). However, setting this property to `YES` does not cause the view to underlap the tab bar view. The tab bar controller always resizes your view to prevent it from underlapping the tab bar.

For more information on full-screen layout for custom views, see "Creating Custom Content View Controllers" in *View Controller Programming Guide for iOS*.

Page View Controllers

You use a page view controller to present content in a page-by-page manner. A page view controller manages a self-contained view hierarchy. The parent view of this hierarchy is managed by the page view controller, and the child views are managed by the content view controllers that you provide.

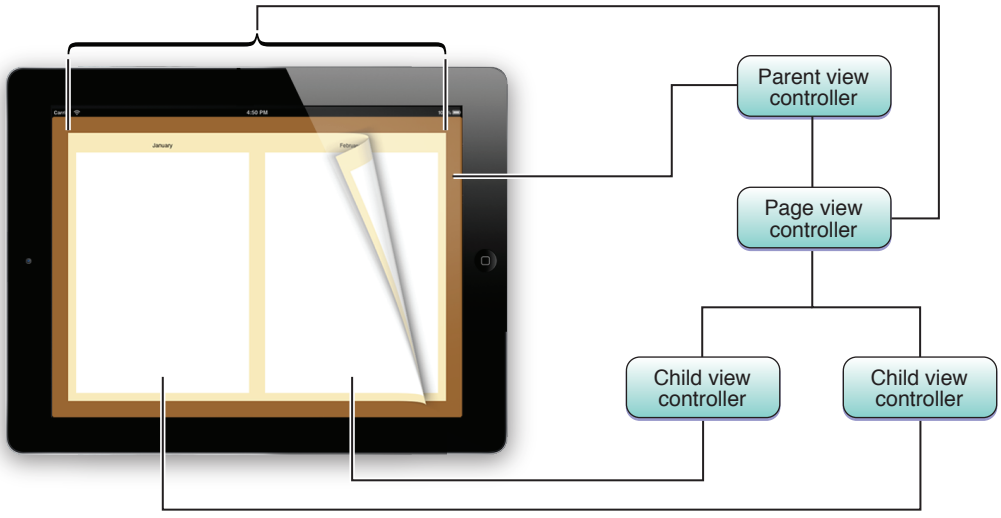
Anatomy of a Page View Controller Interface

A page view controller has a single view in which it hosts your content. The UI provided by a page view controller is visible when the user is turning the page. The page view controller applies a page curl to its view controler's view, providing the visual appearance of a page being turned.

The navigation provided by a page view controller is a linear series of pages. It is well suited for presenting content that is accessed in a linear fashion, such as the text of a novel, and for presenting content that has natural page breaks, such as a calendar. For content that the user needs to access in a nonlinear fashion, such as a reference book, you are responsible for providing the navigation logic and UI.

Figure 3-1 shows the page view interface implemented by a sample application. The outermost brown view is associated with the parent view controller, not the page view controller itself. The page view controller has no UI of its own; however, it does add a page curl effect to its children while the user is turning the page. The custom content is provided by the view controllers that are children of the page view controller.

Figure 3-1 The views of a page view interface

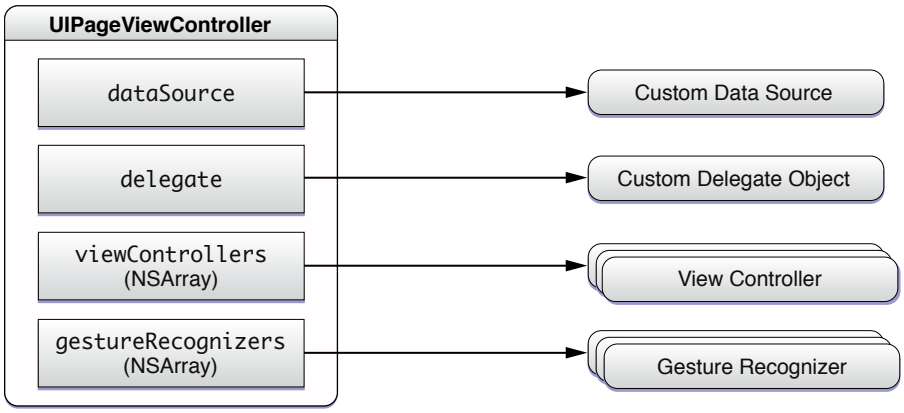


The Objects of a Page View Controller Interface

A page view interface consists of the following objects:

- An optional delegate
- An optional data source
- An array of the current view controllers
- An array of gesture recognizers

Figure 3-2 A page view controller and its associated objects



The data source provides view controllers on demand.

The delegate provides methods that are called in response to gesture-based navigation and orientation changes.

The array of view controllers contains the content view controllers that are currently being displayed. The number of items in this array varies depending on the options passed to the page view controller.

The array of gesture recognizers is populated only if a data source is set. These gesture recognizers let the user turn pages by tapping, flicking, or dragging.

Creating a Page View Controller Interface

A page view controller's view can be resized and embedded in a view hierarchy. This means that, unlike a navigation controller or tab bar controller, a page view controller can be used in a wide variety of settings, not just a few specific cases.

Creating a Page View Controller Interface Using a Storyboard

The Page-Based Application Xcode template creates a new project with a page view controller as the initial scene.

To add a page view controller to an existing storyboard, do the following:

1. Drag a page view controller out of the library. Add a page view controller scene to your storyboard.
2. In the Attributes inspector, set up the appropriate options.
3. Optionally, set a delegate, a data source, or both by connecting the corresponding outlets.
4. Display it as the first view controller by selecting the option Is Initial View Controller in the Attributes inspector (or present the view controller in your user interface in another way.)

Creating a Page View Controller Interface Programmatically

To create a page view controller programmatically:

1. Allocate and initialize a page view controller using the `initWithTransitionStyle:navigationOrientation:options:` method. For initialization-time customization, see "Customizing Behavior at Initialization" (page 49).
2. Optionally, set a delegate, a data source, or both.
3. Set the initial content view controllers.
4. Present the page view controller's view on the screen.

Setting an Initial View Controller

Whether you create a page view controller in Interface Builder or programmatically, you need to set its initial view controllers before displaying it on the screen.

To set the initial view controller, call the `setViewControllers:direction:animated:completion:` method with an array containing the appropriate number of view controllers.

Debugging tip If a page view controller has no initial view controller, it returns NO from `shouldAutorotateToInterfaceOrientation:` for all orientations, raising an exception.

Customizing Behavior at Initialization

You pass parameter values and options to the `initWithTransitionStyle:navigationOrientation:options:` method to customize a page view controller when initializing it. They are accessible as read-only properties after initialization. You can customize:

- The direction in which navigation occurs.
- The location of the spine.
- The transition style. In iOS 5, the only valid transition style is `UIPageViewControllerTransitionStylePageCurl`.

For example, Listing 3-1 initializes a page view controller with horizontal navigation and the spine in the center:

Listing 3-1 Customizing a page view controller

```
NSDictionary * options = [NSDictionary dictionaryWithObject:
            [NSNumber numberWithInt:UIPageViewControllerSpineLocationMid]
    forKey:UIPageViewControllerOptionSpineLocationKey];

UIPageViewController *pageViewController = [[UIPageViewController alloc]
    initWithTransitionStyle:UIPageViewControllerTransitionStylePageCurl
      navigationOrientation:UIPageViewControllerNavigationOrientationHorizontal
                    options:options];
```

Customizing Behavior at Run Time with a Delegate

The page view controller's delegate implements the `UIPageViewControllerDelegate` protocol. It can perform actions when the device orientation changes and when the user navigates to a new page, and it can update the spine location in response to a change in the interface orientation.

Supplying Content by Providing a Data Source

Supplying a data source enables gesture-driven navigation. Without a data source, you must provide your own UI for navigation and supply content as described in "Supplying Content by Setting the Current View Controller" (page 50). The data source you provide must implement the `UIPageViewControllerDataSource` protocol.

The methods of the data source are called with the currently displayed view controller, and return the view controller that come before or after them. To simplify the process of finding the previous or next view controller, you can store additional information in your view controllers, such as a page number.

If a data source is supplied, the page view controller associates gesture recognizers to its view. These gesture recognizers let the user turn the pages by tapping, flicking, and dragging; they are accessible as the `gestureRecognizers` property.

To move the gesture recognizers to another view, pass the value of the `gestureRecognizers` property to the `addGestureRecognizer:` method of the target view. Moving the gesture recognizers to another view is particularly useful if you embed the page view controller into a larger view hierarchy.

For example, if the page view controller doesn't fill the entire bounds of the screen, placing the gesture recognizers on a larger superview can make it easier for users to initiate page turns. Moving the gesture recognizers lets the users start their gestures anywhere within the superview's bounds, not just within the bounds of the page view controller's view.

Supplying Content by Setting the Current View Controller

To directly control what content is being displayed, call the `setViewControllers:direction:animated:completion:` method, passing an array of content view controllers to display. The number of view controllers to pass varies; see the method's reference documentation for specific details.

This approach is how you would let the user jump to a specific location in the content, such as the first page or an overview: You set the view controllers directly, in response to the user interacting with the UI.

If you do not provide a data source, you need to provide UI for moving between pages, such as forward and backward buttons. Gesture-driven navigation is available only when a you provide a datasource.

Special Consideration for Right-to-Left and Bottom-to-Top Content

The page view controller understands content to the left and to the top as being before the current page, and it understands content to the right and to the bottom as being after the current page. This matches the usage for left-to-right and top-to-bottom content.

To use a page view controller to display right-to-left or bottom-to-top content from a data source, just reverse the behavior of two methods:

- In your data source, implement `pageViewController:viewControllerBeforeViewController:` to return the view controller *after* the given view controller.
- In your data source, implement `pageViewController:viewControllerAfterViewController:` to return the view controller *before* the given view controller.

For right-to-left or top-to-bottom content, you typically want to set the spine location to `UIPageViewControllerSpineLocationMax`.

Split View Controllers

The `UISplitViewController` class is a container view controller that manages two panes of information. The first pane has a fixed width of 320 points and a height that matches the visible window height. The second pane fills the remaining space. Figure 4-1 shows a split view controller interface.

Figure 4-1 A split view interface

The panes of a split view interface contain content that is managed by view controllers you provide. Because the panes contain application-specific content, it is up to you to manage interactions between the two view controllers. However, rotations and other system-related behaviors are managed by the split view controller itself.

A split view controller must always be the root of any interface you create. In other words, you must always install the view from a `UISplitViewController` object as the root view of your application's window. The panes of your split view interface may then contain navigation controllers, tab bar controllers, or any other type of view controller you need to implement your interface. Split view controllers cannot be presented modally.

The easiest way to integrate a split view controller into your application is to start from a new project. The Split View-based Application template in Xcode provides a good starting point for building an interface that incorporates a split view controller. Everything you need to implement the split view interface is already provided. All you have to do is modify the array of view controllers to present your content. The process for

modifying these view controllers is virtually identical to the process used in iPhone applications. The only difference is that you now have more screen space available for displaying your detail-related content. However, you can also integrate split view controllers into your existing interfaces.

Creating a Split View Controller Using a Storyboard

If you are creating a new Xcode project, the Master-Detail Application template gives you a split view in the storyboard, set as the first scene.

To add a split view controller to an existing app:

1. Open your application's main storyboard.
2. Drag a split view controller out of the library.

   Interface Builder creates a split view controller a navigation controller and a view controller, and it creates relationships between them. These relationships identify the newly created view controllers as the left and right panes of the split view controller.
3. Display it as the first view controller by selecting the option Is Initial View Controller in the Attributes inspector (or present the view controller in your user interface in another way.)

The contents of the two view controllers you embed in the split view are your responsibility. You configure these view controllers just as you would configure any other view controllers in your application. For example, for embedded navigation and tab bar controllers, you may need to specify additional view controller information.

Creating a Split View Controller Programmatically

To create a split view controller programmatically, create a new instance of the `UISplitViewController` class and assign view controllers to its two properties. Because its contents are built on-the-fly from the view controllers you provide, you do not have to specify a nib file when creating a split view controller. Therefore, you can just use the `init` method to initialize it. Listing 4-1 shows an example of how to create and configure a split view interface at launch time. You would replace the first and second view controllers with the view controller objects that present your application's content. The `window` variable is assumed to be an outlet that points to the window loaded from your application's main nib file.

Listing 4-1 Creating a split view controller programmatically

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions

{
```

```
    MyFirstViewController* firstVC = [[MyFirstViewController alloc] init];
    MySecondViewController* secondVC = [[MySecondViewController alloc] init];

    UISplitViewController* splitVC = [[UISplitViewController alloc] init];
    splitVC.viewControllers = [NSArray arrayWithObjects:firstVC, secondVC, nil];

     window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
     window.rootViewController = splitVC;
    [window makeKeyAndVisible];

    return YES;
}
```

Supporting Orientation Changes in a Split View

A split view controller relies on its two contained view controllers to determine what orientations are supported. It only supports an orientation if both contained view controllers do. Even if one of the contained view controllers isn't currently being displayed, it must support the orientation. When the orientation changes, the split view controller handles most of the rotation behaviors automatically.

In a landscape orientation, the split view controller presents the two panes side-by-side with a small divider separating them. In a portrait orientation, the split view controller either shows both panes or it shows only the second, larger pane and provides a toolbar button for displaying the first pane using a popover, depending on the value returned by the `splitViewController:shouldHideViewController:inOrientation:` delegate method.

Popovers

Although not a view controller itself, the `UIPopoverController` class manages the presentation of view controllers. You use a popover controller object to present content using a **popover**, which is a visual layer that floats above your app's window. Popovers provide a lightweight way to present or gather information from the user and are commonly used in the following situations:

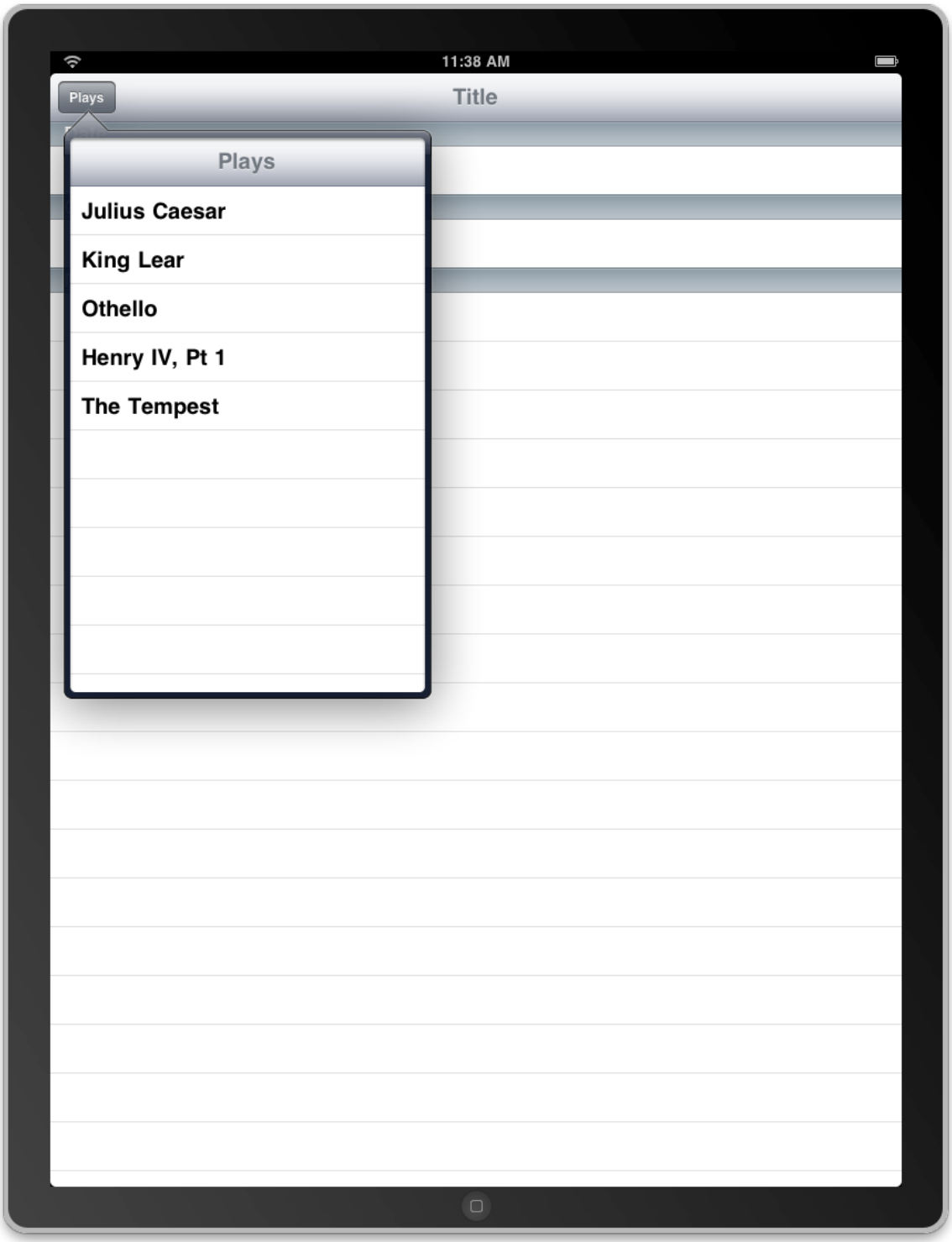
- To display information about an object on the screen
- To manage frequently accessed tools or configuration options
- To present a list of actions to perform on objects inside one of your views
- To present one pane from a split view controller when the device is in a portrait orientation

The use of a popover for the preceding actions is less intrusive and cumbersome than a modal view. In iPad apps, modal views should be reserved for situations where you require the user to explicitly accept or cancel some action or information. For example, you would use a modal view to ask the user for a password that granted access to the rest of your app. For most other cases, you would use a popover instead. The advantage of popovers is that they do not cover the entire screen and can be dismissed by simply tapping outside the popover view. Thus, they are an excellent choice in situations where user interactions with your content are not required but provide information or additional features for the user.

Note You should always dismiss the visible popover before displaying another view controller modally. For specific guidelines on when and how to use popovers in your app, see "Popover (iPad Only)" in *iOS Human Interface Guidelines*.

Figure 5-1 shows an example of a popover used to display a pane from a split view interface. Selecting a play from the popover causes the app's main view to display information about that play. (For more information about creating a split view interface, see "Split View Controllers" (page 52).)

Figure 5-1 Using a popover to display a master pane



Creating and Presenting a Popover

The content of a popover is derived from a view controller object that you provide. Popovers are capable of presenting most types of view controllers. When you are ready to present that view controller in a popover, do the following:

1. Create an instance of the `UIPopoverController` class and initialize it with your view controller object.
2. Specify the size of the popover, which you can do in one of two ways:
    - Assign a value to the `contentSizeForViewInPopover` property of the view controller you want to display in the popover.
    - Assign a value to the `popoverContentSize` property of the popover controller itself.
3. (Optional) Assign a delegate to the popover. For more information about the responsibilities of the delegate, see "Implementing a Popover Delegate" (page 60).
4. Present the popover.

When you present a popover, you associate it with a particular portion of your user interface. Popovers are commonly associated with toolbar buttons, so the `presentPopoverFromBarButtonItem:permittedArrowDirections:animated:` method is a convenient way to present popovers from your app's toolbar. You can also associate a popover with a particular part of one of your views using the `presentPopoverFromRect:inView:permittedArrowDirections:animated:` method.

The popover normally derives its initial size from the `contentSizeForViewInPopover` property of the view controller being presented. The default size stored in this property is 320 pixels wide by 1100 pixels high. You can customize the default value by assigning a new value to the `contentSizeForViewInPopover` property. Alternatively, you can assign a value to the `popoverContentSize` property of the popover controller itself. If you change the view controller displayed by a popover, any custom size information you put in the `popoverContentSize` property is replaced by the size of the new view controller. Changes to the content view controller or its size while the popover is visible are automatically animated. You can also change the size (with or without animations) using the `setPopoverContentSize:animated:` method.

Note The actual placement of a popover on the screen is determined by the popover controller itself and is based on several factors, including the size of your view controller's content, the location of the button or view used as the source of the popover, and the permitted arrow directions.

Listing 5-1 shows a simple action method that presents a popover in response to user taps in a toolbar button. The popover is stored in a property (defined by the owning class) that retains the popover object. The size of the popover is set to the size of the view controller's view, but the two need not be the same. Of course, if the two are not the same, you must use a scroll view to ensure the user can see all of the popover's contents.

Listing 5-1 Presenting a popover programmatically

```
- (IBAction)toolbarItemTapped:(id)sender
{
   MyViewController* content = [[MyViewController alloc] init];
   UIPopoverController* aPopover = [[UIPopoverController alloc]
        initWithContentViewController:content];
   aPopover.delegate = self;

   // Store the popover in a custom property for later use.
   self.popoverController = aPopover;

   [self.popoverController presentPopoverFromBarButtonItem:sender
        permittedArrowDirections:UIPopoverArrowDirectionAny animated:YES];
}
```

Popovers are dismissed automatically when the user taps outside the popover view. Taps within the popover do not cause it to be automatically dismissed, but you can dismiss it programmatically using the `dismissPopoverAnimated:` method. You might do this when the user selects an item in your view controller's content or performs some action that warrants the removal of the popover from the screen. If you do dismiss the popover programmatically, you need to store a reference to the popover controller object in a place where your view controller can access it. The system does not provide a reference to the currently active popover controller.

Implementing a Popover Delegate

When a popover is dismissed due to user taps outside the popover view, the popover automatically notifies its delegate of the action. If you provide a delegate, you can use this object to prevent the dismissal of the popover or perform additional actions in response to the dismissal. The `popoverControllerShouldDismissPopover:` delegate method lets you control whether the popover should actually be dismissed. If your delegate does not implement the method, or if your implementation returns `YES`, the controller dismisses the popover and sends a `popoverControllerDidDismissPopover:` message to the delegate.

In most situations, you should not need to override the `popoverControllerShouldDismissPopover:` method at all. The method is provided for situations where dismissing the popover might cause problems for your app. Rather than returning `NO` from this method, though, it is better to avoid designs that require keeping the popover active. For example, it might be better to present your content modally and force the user to enter the required information or accept or cancel the changes.

By the time the `popoverControllerDidDismissPopover:` method of your delegate is called, the popover itself has been removed from the screen. At this point, it is safe to release the popover controller if you do not plan to use it again. You can also use this message to refresh your user interface or update your app's state.

Tips for Managing Popovers in Your App

Consider the following when writing popover-related code for your app:

- Dismissing a popover programmatically requires a pointer to the popover controller. The only way to get such a pointer is to store it yourself, typically in the content view controller. This ensures that the content view controller is able to dismiss the popover in response to appropriate user actions.
- You can cache popover controllers and reuse them rather than creating new ones from scratch. Popover controllers are very malleable and so you can specify a different view controller and configuration options each time you use them.
- When presenting a popover, specify the `UIPopoverArrowDirectionAny` constant for the permitted arrow direction whenever possible. Specifying this constant gives the UIKit the maximum flexibility in positioning and sizing the popover. If you specify a limited set of permitted arrow directions, the popover controller may have to shrink the size of your popover before displaying it.

Combined View Controller Interfaces

You can use the view controllers that the UIKit framework provides by themselves or in conjunction with other view controllers to create even more sophisticated interfaces. When combining view controllers, however, the order of containment is important; only certain arrangements are valid. The order of containment, from child to parent, is as follows:

- Content view controllers, and container view controllers that have flexible bounds (such as the page view controller)
- Navigation view controller
- Tab bar controller
- Split view controller

Modal view controllers represent an interruption, they follow slightly different rules. You can present almost any view controller modally at any time. It is far less confusing to present a tab bar controller or navigation controller modally from a custom view controller.

The following sections show you how to combine table view, navigation, and tab bar controllers in your iOS apps. For more information on using split view controllers in iPad apps, see "Split View Controllers" (page 52). For more information on table views, see *Table View Programming Guide*.

Adding a Navigation Controller to a Tab Bar Interface

An app that uses a tab bar controller can also use navigation controllers in one or more tabs. When combining these two types of view controller in the same user interface, the tab bar controller always acts as the wrapper for the navigation controllers.

The most common way to use a tab bar controller is to embed its view in your app's main window. The following sections show you how to configure your app's main window to include a tab bar controller and one or more navigation controllers. There are examples for doing this both programmatically and using Interface Builder.

Tab bar views do not support translucency and tab bar controllers never display content underneath their associated tab bar. Therefore, if your navigation interface is embedded in a tab of a tab bar controller, your content may underlap the navigation bar if you adopt a full-screen layout as described in "Adopting a Full-Screen Layout for Navigation Views" (page 18), but it does not underlap the tab bar.

When embedding navigation controllers in a tab bar interface, you should embed only instances of the `UINavigationController` class, and not system view controllers that are subclasses of the `UINavigationController` class. Although the system provides custom navigation controllers for selecting contacts, picking images, and implementing other behaviors, these view controllers are generally designed to be presented modally. For information about how to use a specific view controller, see the reference documentation for that class.

Creating the Objects Using a Storyboard

To create a combined interface with three tabs that contain custom view controllers and one tab that contains a navigation controller:

1. Create three custom view controllers, one for each tab, and a navigation controller.
2. Select the three custom view controllers and the navigation controller (only the navigation controller scene, not it's root view controller).
3. Choose Editor > Embed In > Tab Bar Controller.
4. Display the tab bar controller as the first view controller by selecting the option Is Initial View Controller in the Attributes inspector (or present the view controller in your user interface in another way.)

Creating the Objects Programmatically

If you are creating a combined tab bar and navigation interface programmatically for your app's main window, the most appropriate place to do so is in the `applicationDidFinishLaunching:` method of your application delegate. The following steps explain how to create a combined interface where three tabs contain custom view controllers and one contains a navigation controller:

1. Create the `UITabBarController` object.
2. Create three custom root view controller objects, one for each tab.
3. Create an additional custom view controller to act as the root view controller for your navigation interface.
4. Create the `UINavigationController` object and initialize it with its root view controller.
5. Add the navigation controller and three custom view controllers to your tab bar controller's `viewControllers` property.

Listing 6-1 shows the template code needed to create and install three custom view controllers and a navigation controller as tabs in a tab bar interface. The class names of the custom view controllers are placeholders for classes you provide yourself. The `init` method of each custom view controller is also a method that you provide to initialize the view controllers. The `tabBarController` and `window` variables are declared properties of the class that retain their values.

Listing 6-1 Creating a tab bar controller programmatically

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
    self.tabBarController = [[[UITabBarController alloc] init];

    MyViewController1* vc1 = [[MyViewController1 alloc] init];
    MyViewController2* vc2 = [[MyViewController2 alloc] init];
    MyViewController3* vc3 = [[MyViewController3 alloc] init];
    MyNavRootViewController* vc4 = [[MyNavRootViewController alloc] init];
    UINavigationController* navController = [[UINavigationController alloc]
                            initWithRootViewController:vc4];

   NSArray* controllers = [NSArray arrayWithObjects:vc1, vc2, vc3, navController,
 nil];
    tabBarController.viewControllers = controllers;

    window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    window.rootViewController = tabBarController;
    [window makeKeyAndVisible];
}
```

Even if you create your custom view controllers programmatically, there are no restrictions on how you create the view for each one. The view management cycle is the same for a view controller regardless of how it is created, so you can create your views either programmatically or using Interface Builder as described in "Creating Custom Content View Controllers" in *View Controller Programming Guide for iOS*.

Displaying a Navigation Controller Modally

It is perfectly reasonable (and relatively common) to present navigation controllers modally from your app. In fact, many of the standard system view controllers (including `UIImagePickerController` and `ABPeoplePickerNavigationController`) are navigation controllers that are specifically designed to be presented modally.

When you want to present your own custom navigation interfaces modally, you always pass the navigation controller object as the first parameter to the `presentModalViewController:animated:` method. You must always configure the view controller appropriately before presenting it. At a minimum, your navigation

controller should have a root view controller. And if you want to start the user at a different point in the navigation hierarchy, you must add those view controllers to the navigation stack (without animations) before presenting the navigation controller.

Listing 6-2 shows an example of how to create and configure a navigation controller and display it modally. In the example, the view controllers pushed onto the navigation stack are custom objects you need to define and configure with views. And the `currentViewController` object is a reference to the currently visible view controller that you also need to provide.

Listing 6-2 Displaying a navigation controller modally

```
MyViewController1*  rootVC = [[MyViewController1 alloc] init];
MyViewController2*  nextVC = [[MyViewController2 alloc] init];
NSArray * viewControllers = [NSArray arrayWithObjects:rootVC, nextVC, nil];

// Create the nav controller and set the view controllers.
UINavigationController*  theNavController = [[UINavigationController alloc]
                          initWithRootViewController:rootVC];
[theNavController setViewControllers:viewControllers animated:NO];

// Display the nav controller modally.
[currentViewController presentModalViewController:theNavController animated:YES];
```

As with all other modally presented view controllers, the presenting view controller is responsible for dismissing its modally presented child view controller in response to an appropriate user action. When dismissing a navigation controller though, remember that doing so removes not only the navigation controller object but also the view controllers currently on its navigation stack. The view controllers that are not visible are simply released, but the topmost view controller also receives the usual `viewWillDisappear:` message.

For information on how to present view controllers (including navigation controllers) modally, see "Presenting View Controllers From Other View Controllers" in *View Controller Programming Guide for iOS*. For additional information on how to configure a navigation controller for use in your app, see "Navigation Controllers" (page 11).

Displaying a Tab Bar Controller Modally

It is possible (although uncommon) to present a tab bar controller modally in your app. Tab bar interfaces are normally installed in your app's main window and updated only as needed. However, you could present a tab bar controller modally if the design of your interface seems to warrant it. For example, to toggle from your app's primary operational mode to a completely different mode that uses a tab bar interface, you could present the secondary tab bar controller modally using a crossfade transition.

When presenting a tab bar controller modally, you always pass the tab bar controller object as the first parameter to the `presentModalViewController:animated:` method. The tab bar controller must already be configured before you present it. Therefore, you must create the root view controllers, configure them, and add them to the tab bar controller just as if you were installing the tab bar interface in your main window.

As with all other modally presented view controllers, the parent view controller is responsible for dismissing its modally presented child view controller in response to an appropriate user action. When dismissing a tab bar controller though, remember that doing so removes not only the tab bar controller object but also the view controllers associated with each tab. The view controllers that are not visible are simply released, but the view controller displayed in the currently visible tab also receives the usual `viewWillDisappear:` message.

For information on how to present view controllers (including navigation controllers) modally, see "Presenting View Controllers From Other View Controllers" in *View Controller Programming Guide for iOS*. For information on how to configure a tab bar controller for use in your app, see "Tab Bar Controllers" (page 33).

Using Table View Controllers in a Navigation Interface

It is very common to combine table views with a navigation controller to create a navigation interface. Because navigation controllers facilitate the navigation of a data hierarchy, tables are often used to provide the user with the choice of where to navigate next. Tapping a row in one table takes the user to a new screen displaying the data associated with that row. For example, selecting a playlist in the iPod app takes the user to a list of songs in that playlist.

One way to manage tables is with a `UITableViewController` object. Although this class makes the management of tabular data much easier, you still need to implement custom code to facilitate navigation. Specifically, when the user taps a row in the table, you need to push an appropriate new view controller object onto the navigation stack.

Listing 6-3 shows an example of how to navigate to the next level of data in a navigation interface. Whenever the user taps a specific row in the current table, you use the information associated with that row to initialize a new view controller, which you then push onto the navigation stack. The `initWithTable:andDataAtIndexPath:` method is a custom method that you have to implement yourself; its job is to get the data object for the row and use it to initialize the next level view controller.

Listing 6-3 Navigating data using table views

```
// Implement something like this in your UITableViewController subclass
// or in the delegate object you use to manage your table.
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
*)indexPath
{
   // Create a view controller with the title as its
   // navigation title and push it.
   NSUInteger row = indexPath.row;
   if (row != NSNotFound)
   {
      // Create the view controller and initialize it with the
      // next level of data.
      MyViewController *viewController = [[MyViewController alloc]
                  initWithTable:tableView andDataAtIndexPath:indexPath];
      [[self navigationController] pushViewController:viewController
                  animated:YES];
   }
}
```

For more detailed information about managing tables and using table view controllers, see *Table View Programming Guide for iOS*.

Document Revision History

This table describes the changes to *View Controller Catalog for iOS*.

Date	Notes
2012-02-16	Editorial corrections throughout.
2012-01-09	Now document that describes the view controllers available in iOS, and how to use them. This content was previously part of *View Controller Programming Guide for iOS*.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010