View Controller Programming Guide for iOS

User Experience: Windows & Views

2011-01-07

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

Contents

Figures, Tables, and Listings

About View Controllers

For iOS applications, view controllers provide a vital link between an application's data and its visual appearance. Understanding when and how to use view controllers is crucial to the design of iOS applications. View controllers are traditional controller objects in the Model-View-Controller design paradigm but they also do much more. In iOS applications, view controllers provide much of the logic needed to manage basic application behaviors. For example, view controllers manage the presentation and removal of content from the screen and they manage the reorientation of views in response to device orientation changes.

Important: The contents of this document apply to iOS applications only. Although Mac OS X applications also support a type of view controller, they do so using a different class that has different requirements and behaviors.

At a Glance

View controllers exist to make it easier for you to create applications that conform to the platform design guidelines. However, the default view controller classes can only do so much. At some point, your custom code must take over and finish the job. This document shows you the basic behaviors that all view controllers provide and where you can customize that behavior to suit your own needs.

View Controllers Manage a View Hierarchy

Each view controller is responsible for managing a discrete part of your application's user interface. View controllers are directly associated with a single view object but that object is often just the root view of a much larger view hierarchy that is also managed by the view controller. The view controller acts as the central coordinating agent for the view hierarchy, handling exchanges between its views and any relevant controller or data objects. A single view controller typically manages the views associated with a single screen's worth of content, although in iPad applications this may not always be the case.

Relevant chapter: "View Controller Basics" (page 15)

You Manage Your Content Using a Custom View Controller

Whenever you want to present content that is specific to your application, you do so using a custom view controller. You create custom view controllers by subclassing either `UIViewController` or `UITableViewController` directly and implementing the methods necessary to present your content. At a minimum, a custom view controller must be capable of presenting and managing the views associated with your custom content. You may also need to implement other view controller methods to customize behaviors such as rotation, memory management, event handling, and to interact with other view controllers in your application.

Relevant chapter: "Custom View Controllers" (page 27)

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 application. Applications that contain structured content can use navigation controllers to navigate between different 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 chapter: "Navigation Controllers" (page 63)

Tab Bar Controllers Manage Independent Sets of View Controllers

A tab bar controller is an instance of the `UITabBarController` class that you use as-is in your application. Applications 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 application 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 chapter: "Tab Bar Controllers" (page 85)

iPad Supports Special Containers for Your Content

The larger screen size of iPad provides new opportunities for presenting content. The `UISplitViewController` class manages the implementation of a master-detail interface, where both the master and detail portions of the interface are themselves managed by view controllers. Although the `UIPopoverController` class is not a view controller itself, it works with your application's view controllers to present content in a floating view.

Relevant chapter: "iPad-Specific Controllers" (page 99)

Modal View Controllers Temporarily Interrupt the Current Workflow

Any view controller can be presented modally. The purpose of modal view controllers is to interrupt the current workflow temporarily to gather or present information. For example, you might use modal view controllers to display preferences or configuration options, gather data from the user, or even facilitiate transitions between distinct portrait and landscape representations of the current screen. Because they always involve a full-screen transition, the use of modally presented view controllers is used only sparingly in iPad applications but is fairly common in iPhone applications.

Compose view controllers are a specific type of view controller that is almost always presented modally. Compose view controllers are defined by the system and used to present specific interfaces, such as for composing email or SMS messages. On iPhone and iPod touch, you always present compose controllers modally but on iPad you may also present them using a popover.

Relevant chapter: "Modal View Controllers" (page 109)

View Controllers Can Be Combined to Create Sophisticated Layouts

In all but the simplest applications, it is common to find multiple view controllers working 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 modally. 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 chapter: "Combined View Controller Interfaces" (page 119)

Prerequisites

Before you start reading this document, you should have at least a fundamental understanding of the following Cocoa concepts:

- Basic information about Xcode and Interface Builder and their role in developing applications
- How to define new Objective-C classes
- How to manage memory, including how to create and release objects in Objective-C
- The role of delegate objects in managing application behaviors
- A basic understanding of the Model-View-Controller paradigm

Developers who are new to Cocoa and Objective-C can get information about all of these topics in *Cocoa Fundamentals Guide*.

Development of iOS applications requires an Intel-based Macintosh computer running Mac OS X v10.5 or later. You must also download and install the iOS SDK. For information about how to get the iOS SDK, go to the Apple Developer website.

See Also

For additional information about application design, see *iOS Application Programming Guide*.

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

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

View Controller Basics

View controllers provide the fundamental infrastructure you need to implement iOS applications. This chapter provides an overview of the role view controllers play in your application and how you use them to implement different types of user interfaces.

What Is a View Controller?

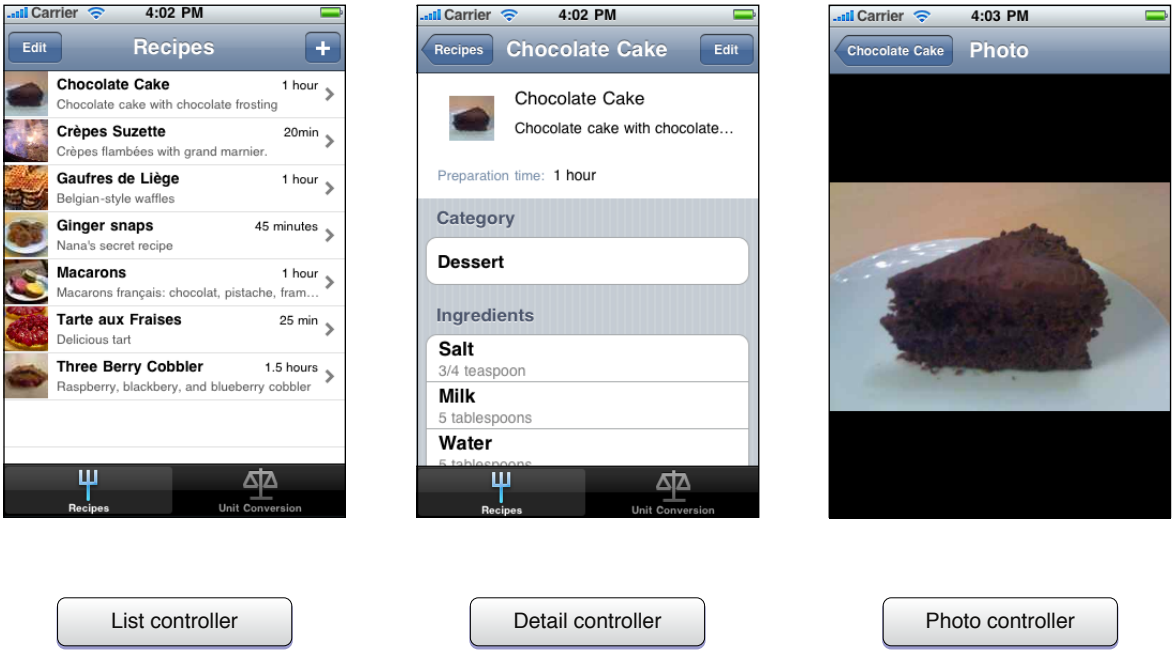
In the Model-View-Controller (MVC) design pattern, a controller object provides the custom logic needed to bridge the application's data to the views and other visual entities used to present that data to the user. In iOS applications, a **view controller** is a specific type of controller object that you use to present and manage a set of views. View controller objects are descendants of the `UIViewController` class, which is defined in the UIKit framework.

View controllers play a very important role in the design and implementation of iOS applications. Applications running on iOS–based devices have a limited amount of screen space for displaying content and therefore must be creative in how they present information to the user. Applications that have lots of content may have to distribute that content across multiple screens or show and hide different parts of content at different times. View controller objects provide the infrastructure for managing your content-related views and for coordinating the showing and hiding of them.

There are many reasons to use view controllers in your application and very few reasons to avoid them. View controllers make it easier for you to implement many of the standard interface behaviors found in iOS applications. They provide default behavior that you can use as is or customize when needed. They also provide a convenient way to organize your application's user interface and content.

Figure 1-1 shows an example of three different (but related) screens from an iPhone application that manages recipes. The first screen lists the recipes that the application manages. Tapping one of the recipes displays the second screen, which shows the details of the recipe. Tapping the recipe's picture in this detail view displays the third screen, which displays a picture of the resulting dish that fills the screen. Managing each of these screens is a distinct view controller object whose job is to present the appropriate view objects, populate those views with data, and respond to interactions with that view.

Figure 1-1 Distinct screens managed by separate view controllers



In addition to displaying and managing views, you can also use view controllers to manage the navigation from screen to screen. In iOS, there are several standard techniques for presenting new screens. All of these techniques are implemented using view controllers and are discussed at different points throughout this document.

Types of View Controllers

Most iOS applications have at least one view controller and some have several. Broadly speaking, view controllers are divided into three general categories that reflect the role the view controller plays in your application.

A **custom view controller** is a controller object that you define for the express purpose of presenting some content on the screen. Most iOS applications present data using several distinct sets of views, each of which handles the presentation of your data in a specific way. For example, you might have a set of views that presents a list of items in a table and another set that displays the details for a single item in that list. The corresponding architecture for such an application would involve the creation of separate view controllers to manage the marshaling and display of each set of views.

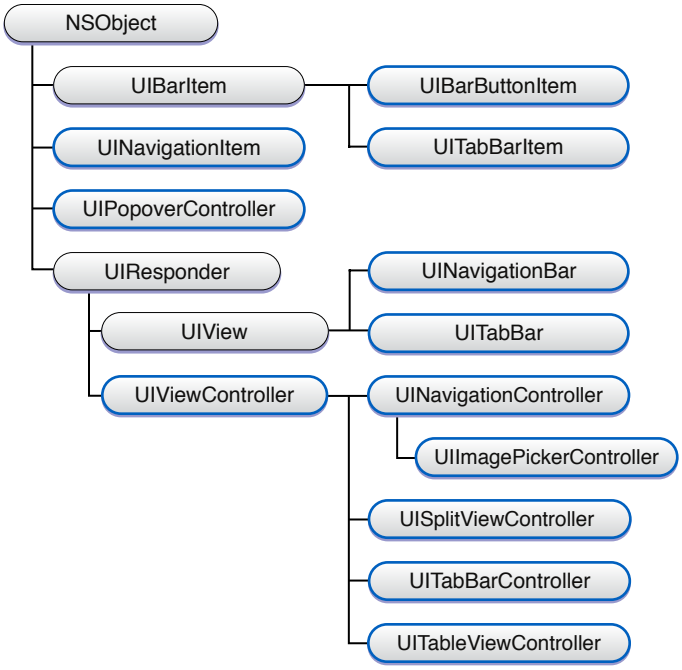
A **container view controller** is a specific type of view controller object that manages other view controllers and defines the navigational relationships among them. Navigation, tab bar, and split view controllers are all examples of container view controllers. You do not define container view controllers yourself. Instead, you use the container view controllers provided by the system as is.

Note: A popover controller is not an actual view controller but does behave similarly to a container view controller. Its job is to present the contents of a view controller that you provide in a popover view.

A **modal view controller** is a view controller (container or custom) that is presented in a specific way by another view controller. Modal view controllers define a specific navigational relationship in your application. The most common reason to present a view controller modally is so that you can prompt the user to input some data. For example, you might present a view controller modally to have the user fill in a form or select an option from a picker interface. However, there are other uses for modal view controllers that are described in more detail in "Modal View Controllers" (page 109).

Figure 1-2 shows the view controller classes available in the UIKit framework along with some of the key classes used in conjunction with view controllers. These additional classes are often used internally by view controller objects to implement special types of interfaces. For example, the `UITabBarController` object manages a `UITabBar` object, which actually displays the tabs associated with the tab bar interface. Other frameworks may also define additional view controller objects in order to present specific types of interfaces.

Figure 1-2 View controller classes in UIKit



The sections that follow provide more detail about the types of view controllers you use to organize and present your application's content.

About Custom View Controllers

Custom view controllers are the primary coordinating objects for your application's content. Nearly every application has at least one custom view controller, and a complex application might have many of them. A custom view controller contains the logic and glue code needed to facilitate interactions between a portion of your application's data and the views used to present that data. The view controller may also interact with other controller objects in your application, including the application delegate and other view controllers.

Each custom view controller object you create is responsible for managing all of the views in a single view hierarchy. In iPhone applications, the views in a view hierarchy traditionally cover the entire screen, but in iPad applications they may cover only a portion of the screen. The one-to-one correspondence between a view controller and the views in its view hierarchy is the key design consideration. You should not use multiple custom view controllers to manage different portions of the same view hierarchy. Similarly, you should not use a single custom view controller object to manage multiple screens worth of content.

Note: If you want to divide a view hierarchy into multiple subareas and manage each one separately, use generic controller objects (custom objects descending from `NSObject`) instead of view controller objects to manage each subarea. Then use a single view controller object to manage the generic controller objects.

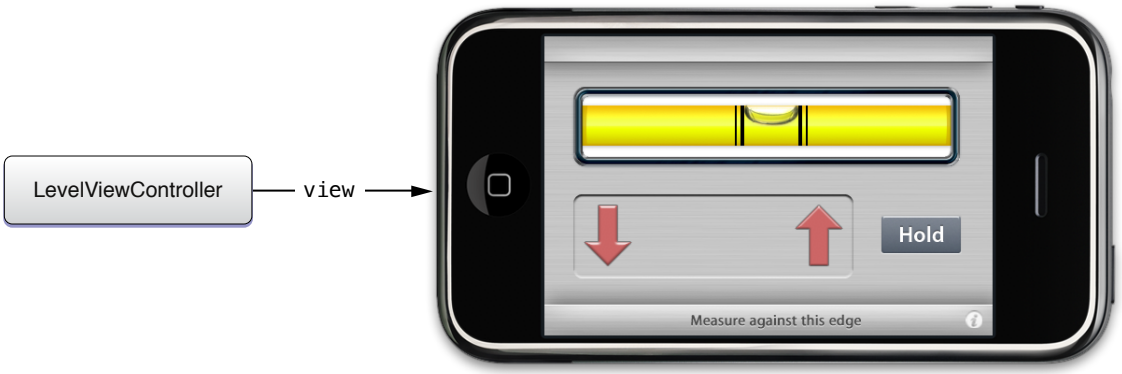
You create a custom view controller by subclassing `UIViewController` directly and adding custom code to your subclass. The declaration of a typical `UIViewController` subclass includes things such as the following:

- Member variables pointing to the objects containing the data to be displayed by the corresponding views
- Member variables (or outlets) pointing to key view objects with which your view controller must interact
- Action methods that perform tasks associated with buttons and other controls in the view hierarchy
- Any additional methods needed to implement your view controller's custom behavior

Because you use it to manage your custom content, most of the code in this type of view controller is going to be specific to your application. However, there are also some common behaviors that all view controllers can support. For these common behaviors, the `UIViewController` class defines methods that you can override and use to implement the desired behavior. Some of the common behaviors include view management, interface rotation management, and low-memory warning support.

Figure 1-3 shows an example of a custom view controller in the sample project *BubbleLevel*. This application defines the `LevelViewController` class, which is a direct descendant of `UIViewController`. This class monitors the accelerometer data for changes in the pitch of the device and uses that data to update its associated view object. The `view` property of the view controller provides a reference to the actual view object presenting the content.

Figure 1-3 Custom view controller in the BubbleLevel application



For information about managing the standard behaviors required of all view controllers, see "Custom View Controllers" (page 27).

About Table View Controllers

The `UITableViewController` class is another type of custom view controller designed specifically for managing tabular data. Although it is certainly possible to manage tables without a table view controller, the class adds automatic support for many standard table-related behaviors such as selection management, row editing, table configuration, and others. This additional support is there to minimize the amount of code you have to write to create and initialize your table-based interface. You can use a table view controller in the same places you would use a custom view controller. You can also subclass it and implement additional custom behaviors. Of course, any view hierarchy managed by such a view controller should include a table view object.

Figure 1-4 shows an example of the configuration of a table view controller. Because it is a subclass of `UIViewController`, the table view controller still has a pointer to the root view of the interface (through its `view` property) but it also has a separate pointer to the table view displayed in that interface.

Figure 1-4 Managing tabular data

This document covers only the behaviors that are common to all view controllers and does not cover any information specific to table view controllers. For specific information about the table-related behaviors of a table view controller, see *UITableViewController Class Reference*. For more information about managing table views in general, see *Table View Programming Guide for iOS*.

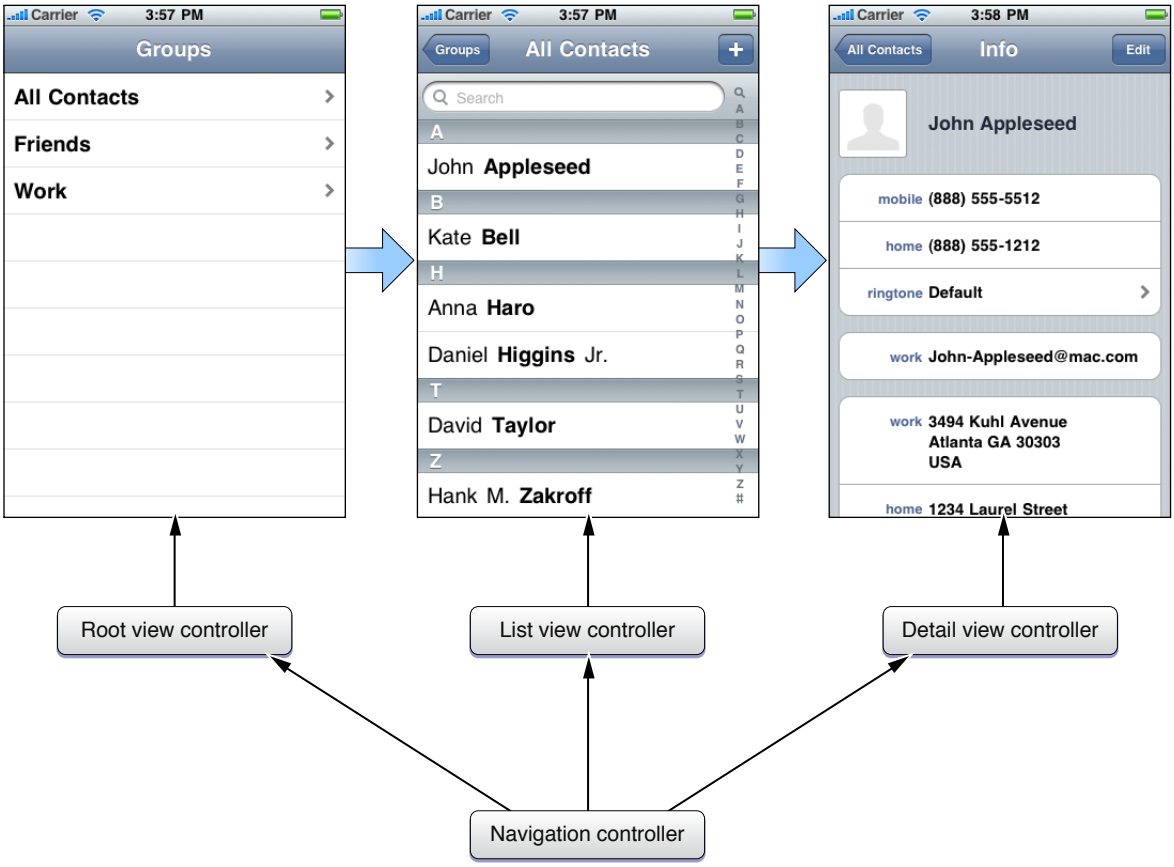
About Navigation Controllers

A navigation controller is a container view controller that you use to present data that is organized hierarchically. A navigation controller is an instance of the `UINavigationController` class, which is a class you use as is and do not subclass. The methods of this class provide support for managing a stack-based collection of custom view controllers. This stack represents the path taken by the user through the hierarchical data, with the bottom of the stack reflecting the starting point and the top of the stack reflecting the user's current position in the data.

Although the navigation controller's primary job is to act as a manager of other view controllers, it also manages a few views. Specifically, it manages a navigation bar that displays information about the user's current location in the data hierarchy, a back button for navigating to previous screens, and any custom controls the current view controller needs. The navigation controller also manages an optional toolbar, which can be used to display commands related to the current screen. You do not modify these views directly in most cases but configure them through support found in the `UIViewController` class.

Figure 1-5 shows some screens from the Contacts application, which uses a navigation controller to present contact information to the user. Each screen displayed to the user is managed by a custom view controller object, which presents information at that specific level of the data hierarchy. For example, the root view controller and list view controllers manage the presentation of tabular contact information in different ways. The detail view controller displays the information for a specific contact using an entirely different type of screen. As the user interacts with controls in the interface, those controls tell the navigation controller to display the next view controller in the sequence or dismiss the current view controller.

Figure 1-5 Navigating hierarchical application data



For information about how to configure and use navigation controller objects, see "Navigation Controllers" (page 63).

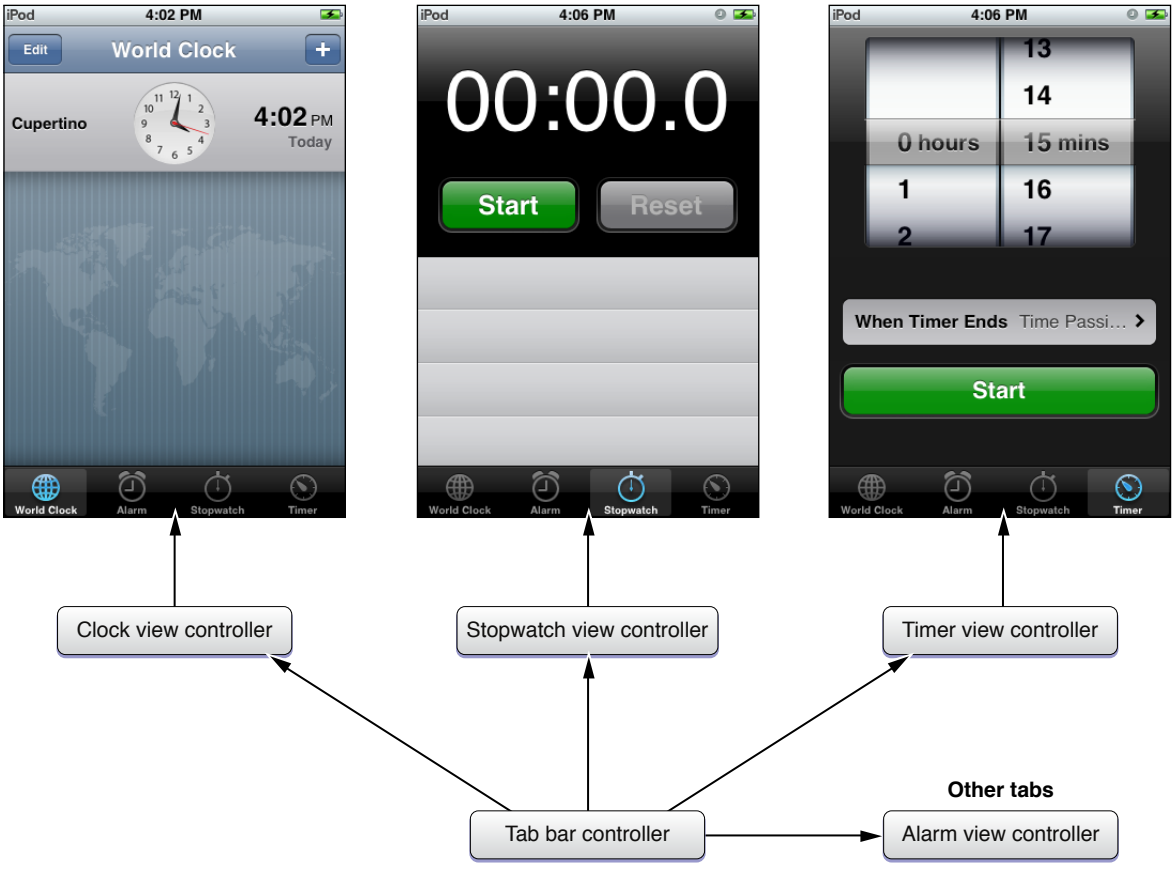
About Tab Bar Controllers

A tab bar controller is a container view controller that you use to divide your application into two or more distinct modes of operation. A tab bar controller is an instance of the `UITabBarController` class, which is a class you use as is and do not subclass. The modes of the tab bar controller are presented using a tab bar view, which displays a tab for each supported mode. Selecting a tab causes an associated view controller to present its interface on the screen.

You use tab bar controllers in situations where your application either presents different types of data or presents the same data in significantly different ways. The tab bar controller facilitates the automatic switching of modes in response to user taps on the tab bar view. If there are more modes than there is space for tabs, the tab bar controller also manages the selection of tabs that are not normally visible and the customization of the tabs that are visible.

Figure 1-6 shows several modes of the Clock application along with the relationships between the corresponding view controllers. Each mode has a root view controller to manage the main content area. In the case of the Clock application, the Clock and Alarm view controllers both display a navigation-style interface to accommodate some additional controls along the top of the screen. The other modes use custom view controllers to present a single screen.

Figure 1-6 Different modes of the Clock application



For information about how to configure and use a tab bar controller, see "Tab Bar Controllers" (page 85).

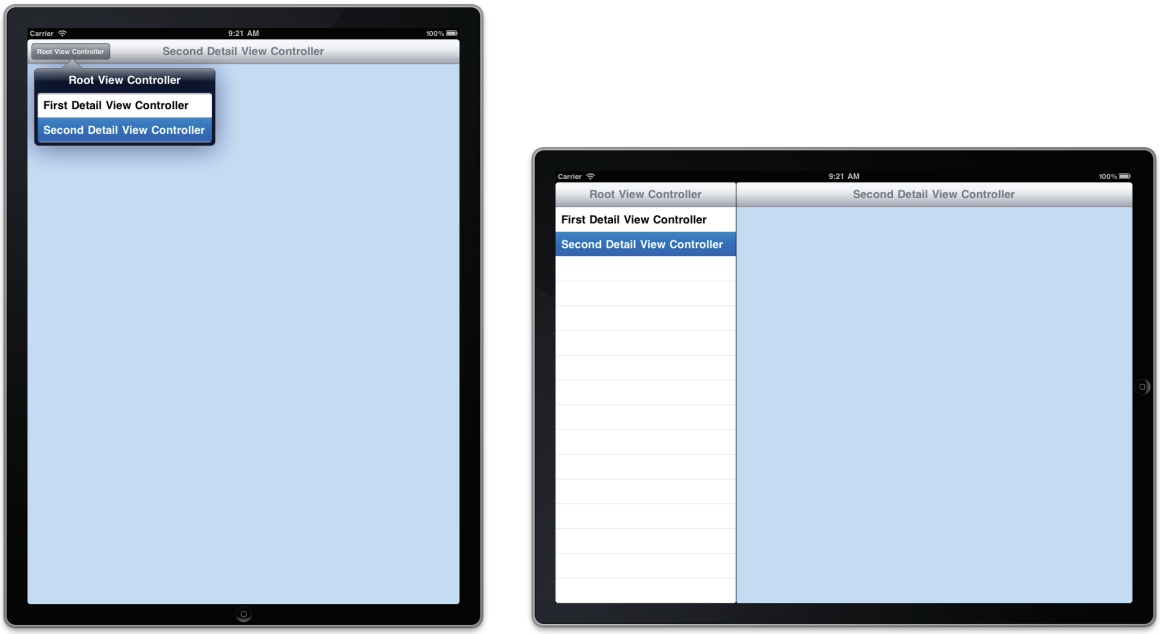
About Split View Controllers

A split-view controller is a container view controller that is typically used to implement master-detail interfaces. A split view controller is an instance of the `UISplitViewController` class, which is a class you use as is and do not subclass. The contents of a split view interface are derived from two view controllers that you provide. In landscape orientations, a split view controller displays the contents of two other view controllers side-by-side. In portrait orientations, it displays only one of the view controllers directly and makes the other one available from a popover.

Note: Split-view controllers are supported on iPad only and are designed to help you take advantage of the larger screen of that device.

Figure 1-7 shows a split view interface from the *MultipleDetailViews* sample application. The landscape version of the interface displays the list view and detail view side-by-side. In portrait mode, only the detail view is displayed and the list view is made available using a popover. Both the list view and detail view are managed by a custom view controller.

Figure 1-7 A master-detail interface in portrait and landscape modes



For information about how to configure and use a split view controller, see "Split View Controller" (page 103).

About Modal View Controllers

A modal view controller is not a specific view controller class but is a way of presenting any view controller to the user. Although container view controllers define specific relationships between the managed view controllers, modal view controllers let you define the relationship. Any view controller object can present any other view controller object modally. Most of the time, you present view controllers modally in order to

gather information from the user or capture the user's attention for some specific purpose. Once that purpose is completed, you dismiss the modal view controller and allow the user to continue navigating through your application.

Figure 1-8 shows an example from the Contacts application. When the user clicks the plus button to add a new contact, the Contacts view controller presents the New Contact view controller modally. This creates a parent-child relationship between the two view controllers. The New Contact screen remains visible until the user cancels the operation or provides enough information about the contact that it can be saved to the contacts database, at which point the Contacts view controller dismisses its child.

Figure 1-8 Presenting a modal view controller

It is worth noting that a view controller presented modally can itself present another view controller modally. This ability to chain modal view controllers can be useful in situations where you might need to perform several modal actions sequentially. For example, if the user taps the Add Photo button in the New Contact screen in the preceding figure and wants to choose an existing image, the New Contact view controller presents an image picker interface modally. The user must dismiss the image picker screen and then dismiss the New Contact screen separately in order to return to the list of contacts.

For more information about the uses for modal view controllers and how to present them in your application, see "Modal View Controllers" (page 109).

Getting Started with the iOS Application Templates in Xcode

Most of the Xcode project templates for iOS applications provide you with at least one view controller class initially, and some may provide you with multiple view controllers. These initial classes provide you with the code found in a typical view controller and are intended to help you start writing your application quickly. It is important to remember though that the templates are just a starting point.

The goal of the template applications is to show you the best way to get started with specific types of applications. It is always easiest to start with the template that most closely matches the interface you are trying to create. For example, if you are creating an application similar to the Stocks or Weather applications, you would start with the Utility Application template. On the other hand, if you plan to use a tab bar to divide your application into different modes, you should start with the Tab Bar Application template.

If you want to explore basic view controller behaviors, the View-based Application template is a good place to start. This type of application uses a single custom view controller to display the contents of the application. You can expand on this basic behavior by presenting additional view controllers modally.

If you want to build your application's user interface from scratch, start with the Window-based Application template. This template provides a minimally configured project that you can modify to include the view controllers you need.

For more information about creating projects in Xcode, see *Xcode Project Management Guide*.

Custom View Controllers

Custom view controllers are what you use to present the content of your application. The job of any view controller is to manage the presentation of some content and coordinate the update and the synchronization of that content with the application's underlying data objects. In the case of a custom view controller, this involves creating a view to present the content and implementing the infrastructure needed to synchronize the contents of that view with your application's data structures.

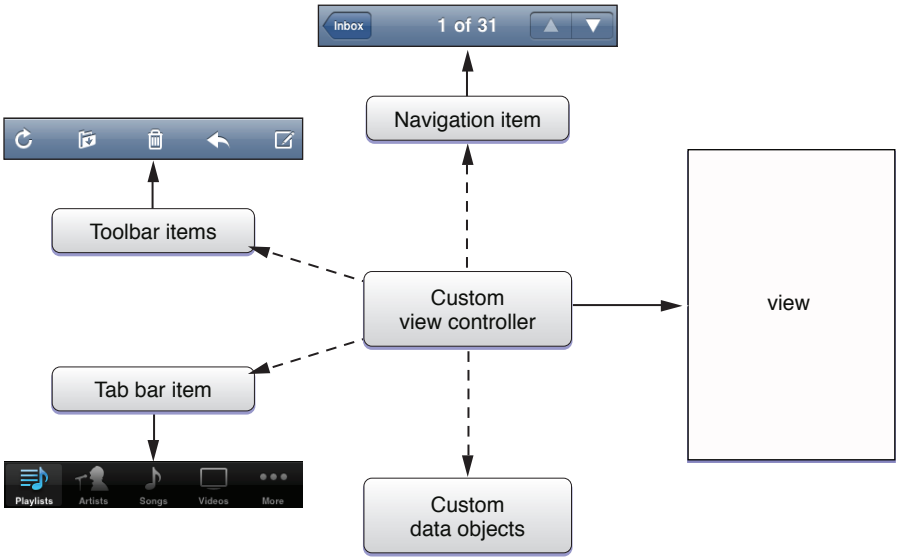
The `UIViewController` class provides the basic behavior required for all view controllers—custom or otherwise. This chapter explains the fundamental behaviors provided by this class and also shows you how to modify those behaviors to suit the needs of your application. Understanding these behaviors and how you can modify them is essential to implementing your application's custom view controllers; it is also useful when interacting with view controllers of any type.

Anatomy of a Custom View Controller

The `UIViewController` class provides the fundamental infrastructure for implementing all custom view controllers. Although you can configure an instance of the `UIViewController` class to display some views, you need to define a custom subclass if you want to do anything interesting. In your subclass, you use custom methods to populate views with data and respond to taps in buttons and other controls. However, when you want to make adjustments to the default behavior of the view controller, you need to override methods of the `UIViewController` class. You may also need to interact with other UIKit classes to implement the behavior you want.

Figure 2-1 shows some of the key objects associated directly with a custom view controller. These are the objects that are essentially owned and managed by the view controller itself. The view (accessible via the `view` property) is the only object that must be provided, although most view controllers also have some custom objects containing the data they need to display. Other objects are used only as needed to support features such as navigation and tab bar interfaces, and even so, most provide default behavior that is often sufficient.

Figure 2-1 Anatomy of a custom view controller



Although view controllers rarely act alone, custom view controllers should always be designed to be independent objects. In other words, a view controller should encapsulate all of the behavior associated with managing the views in its view hierarchy. Your view controller should contain the data it needs (or at least a reference to that data that it controls), the views needed to display that data, the logic to assimilate changes in the system (such as orientation changes), and the code needed to validate or process user interactions. Any custom objects needed to manage portions of the view hierarchy or data model should be created and managed wholly by the view controller.

Implementation Checklist for Custom View Controllers

When you want to implement a custom view controller for your application, you use Xcode to set up your source files. Most iOS project templates include at least one view controller class and you can create new view controllers in Xcode as needed.

For any custom view controllers you create, there are several tasks that you should always handle:

- You must configure the view to be loaded by your view controller; see "Creating the View for Your View Controller" (page 33).
- You must decide which orientations your view controller supports; see "Managing a View Controller's Interface Orientation" (page 43).
- You must clean up the memory that is managed by your view controller; see "Managing Memory Efficiently" (page 41).

As you configure your view controller's view, you will likely discover that you need to define action methods or outlets to use with those views. For example, if your view hierarchy contains a table, you probably want to store a pointer to that table in an outlet so that you can access it later. Similarly, if your view hierarchy

contains buttons or other controls, you may want those controls to call an associated action method in response to user interactions. As you iterate through the definition of your view controller class, you may therefore find that you need to add the following items to your view controller class:

- Member variables pointing to the objects containing the data to be displayed by the corresponding views
- Member variables (or outlets) pointing to key view objects with which your view controller must interact
- Action methods that perform tasks associated with buttons and other controls in the view hierarchy
- Any additional methods needed to implement your view controller's custom behavior

Important: When adding outlets or other member variables to your custom view controllers, it is highly recommended that you include declared properties for each variable you intend to access. Declared properties provide a convenient syntax for accessing member variables and also eliminate the need for you to write much of the glue code required for managing those variables. Especially when referring to other objects, properties can provide tremendous convenience by automatically retaining and releasing objects as needed.

The preceding items are the ones that you are the most likely to include in every custom view controller class you create. However, there are other methods you might add to your view controller to implement specific behaviors. Many of these methods take advantage of hooks in the view controller infrastructure to implement common tasks.

- You can adjust your view hierarchy or application state in response to the view controller's view appearing or disappearing from the screen; see "Responding to Display-Related Notifications" (page 54).
- You can configure objects used by navigation and tab bar controllers, including:
  - The navigation item (if the controller is used in conjunction with a navigation controller interface); see "Customizing the Navigation Bar Appearance" (page 75).
  - The toolbar items (if an associated navigation controller displays a toolbar); see "Specifying the Toolbar Items" (page 82).
  - The tab bar item (if the view controller is used in conjunction with a tab bar controller interface); see "Creating a Tab Bar Interface" (page 88).
- You can adjust your view hierarchy when the interface orientation changes; see "Responding to Orientation Changes" (page 45).
- You can implement event handlers to catch any events not handled by the view or its subviews; see *Event Handling Guide for iOS*.
- You can implement an editable version of your view; see "Enabling Edit Mode for a View" (page 57).

Understanding the View Management Cycle

In a view controller object, management of the corresponding view occurs in two distinct cycles: the load and unload cycles. The load cycle occurs whenever some part of your application asks the view controller for a pointer to its view object and that object is not currently in memory. When that happens, the view controller loads the view into memory and stores a pointer to the view for future reference.

If your application receives a low-memory warning at some point in the future, the view controller may subsequently try to unload the view. During the unload cycle, the view controller attempts to release its view object and return the view controller to its initial viewless state. If it is able to release the view, the view controller remains without a view object until the view is once again requested, at which point the load cycle begins again.

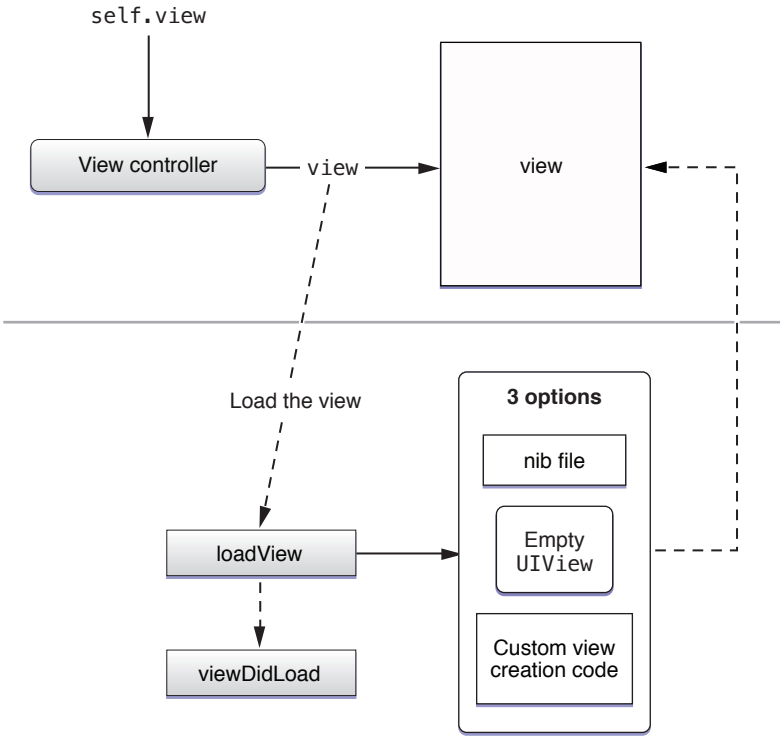
During the load and unload cycles, the view controller does most of the work of loading and unloading the view. However, if your view controller class stores references to views in the view hierarchy or needs to perform some additional configuration of the views at load time, you can override specific methods (which are described in the material that follows) to perform any extra tasks.

The steps that occur during the load cycle are as follows:

1. Some part of your application asks for the view in the view controller's `view` property.
2. If the view is not currently in memory, the view controller calls its `loadView` method.
3. The `loadView` method does one of the following:
    - If you override this method, your implementation is responsible for creating all necessary views and assigning a non-`nil` value to the `view` property.
    - If you do not override this method, the default implementation uses the `nibName` and `nibBundle` properties of the view controller to try to load the view from the specified nib file. If the specified nib file is not found, it looks for a nib file whose name matches the name of the view controller class and loads that file.
    - If no nib file is available, the method creates an empty `UIView` object and assigns it to the `view` property.
4. The view controller calls its `viewDidLoad` method to perform any additional load-time tasks.

Figure 2-2 shows a visual representation of the load cycle, including several of the methods that are called. Your application can override both the `loadView` and `viewDidLoad` methods as needed to facilitate the behavior you want for your view controller.

Figure 2-2 Loading a view into memory

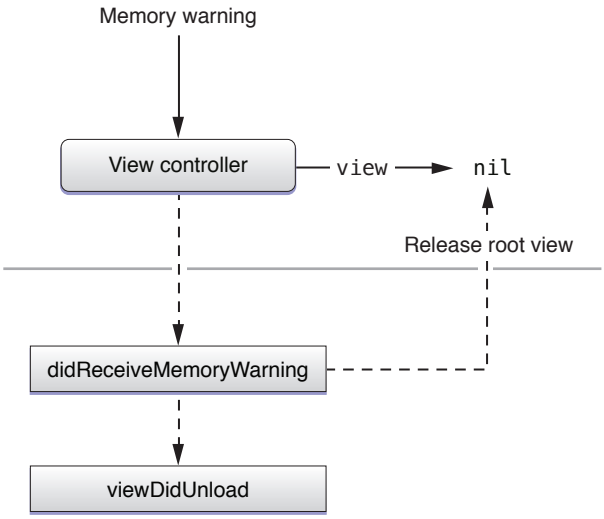


The steps that occur during the unload cycle are as follows:

1. The application receives a low-memory warning from the system.
2. Each view controller calls its `didReceiveMemoryWarning` method:
    - If you override this method, you should use it to release any custom data that your view controller object no longer needs. You should not use it to release your view controller's view. You must call `super` at some point in your implementation to perform the default behavior.
    - The default implementation releases the view only if it determines that it is safe to do so.
3. If the view controller releases its view, it calls its `viewDidUnload` method. You can override this method to perform any additional cleanup required for your views and view hierarchy.

Figure 2-3 shows a visual representation of the unload cycle for a view controller.

Figure 2-3 Unloading a view from memory



Important: In iOS 3.0 and later, the `viewDidUnload` method is the preferred place to put any code related to cleaning up your views. You might also override the `didReceiveMemoryWarning` method to release temporary caches or other private data that is no longer needed when the view is released. If you do override `didReceiveMemoryWarning`, always call `super` to give the inherited version of the method a chance to release the view.

In iOS 2.2 and earlier, you must use the `didReceiveMemoryWarning` method to perform your view-related cleanup and to release any unneeded private data structures. The `viewDidUnload` method is available only in iOS 3.0 and later.

For more information about managing memory during low-memory conditions, see "Managing Memory Efficiently" (page 41).

Defining a Custom View Controller Class

A custom view controller is a subclass of `UIViewController` that you use to present your application's content. Many of the Xcode project templates come with a custom view controller class that you can modify for your needs. If you need to create additional custom view controllers, do the following:

1. Choose File > New File to add a new source file to your project.

   You want to create a new `UIViewController` subclass. There is a template for this type of class in the Cocoa Touch Classes section of the New File dialog.

2. Give your new view controller file an appropriate name and add it to your project.

3. Save your source files.

Once you have your view controller source files, you can implement the behaviors needed to present your content. The following sections describe the key tasks you can perform using a custom view controller. For additional information about creating a view controller, see *UIViewController Class Reference*.

Creating the View for Your View Controller

The main job of a view controller is to load and unload its view as needed. Most view controllers load their views from an associated nib file. The advantage of using nib files is that they allow you to lay out and configure your views graphically, making it easier and faster to adjust your layout. However, you can also create the view programmatically if you prefer.

Creating the View in Interface Builder

Interface Builder provides an intuitive way to create and configure the views for your view controllers. As its name suggests, Interface Builder is a tool for building your application's interface graphically, as opposed to programmatically. Using this application, you assemble views and controls by manipulating them directly, dragging them into the workspace, positioning them, sizing them, and modifying their attributes using an inspector window. The results are then saved in a nib file, which stores the collection of objects you assembled along with information about all the customizations you made.

There are two ways to configure a nib file for use with a view controller:

- Create a **detached nib file** by storing the view in a nib file by itself.
- Create an **integrated nib file** by storing both the view and view controller in the same nib file.

Of the two techniques, using a detached nib file is by far the preferred way to go. Detached nib files offer a more robust solution, especially in the realm of memory management. During a low-memory condition, the contents of a detached nib file can be purged from memory as needed without affecting the state of the owning view controller object. The same is not true in the case of an integrated nib file, the contents of which must stay in memory until all nib-file objects are no longer needed.

Storing the View in a Detached Nib File

The process for creating a detached nib file involves two separate implementation steps:

- You must configure a nib file with the view.
- You must associate that nib file with your view controller object.

The configuration of the nib file itself is relatively straightforward. If you are creating the nib file from scratch, you should use the Cocoa Touch View template in Interface Builder as your starting point. The nib file created by this template contains the File's Owner placeholder and a single custom view object. Add this new nib file to your Xcode project and then configure its contents as follows:

1. Set the class name of the File's Owner placeholder to your view controller class.

   You should have already created the class in your Xcode project. For new nib files, the class of File's Owner is set to `NSObject` by default. (If you are editing a nib file that was provided for you by one of the Xcode project templates, the class may already be set to the correct view controller class name.)

2. Make sure the `view` outlet of the File's Owner placeholder is connected to the top-level View object in the nib file.

This `view` outlet is defined by the `UIViewController` class and inherited by all view controller objects. If you do not see this outlet in the File's Owner placeholder, check to see whether you added the nib file to your Xcode project. Associating the nib file with an Xcode project lets Interface Builder retrieve information about the classes in that project automatically. This is necessary to determine the available outlets and actions of those classes.

If you forget to connect this outlet, the `view` property of your view controller class is set to `nil` when the nib file is loaded, which prevents your view from being presented on the screen.

3. Configure the view itself and add any subviews you need to display your application's content.
4. Save the nib file.

After creating the nib file and adding it to your Xcode project, you need to initialize your view controller object with the name of that nib file. How you initialize your view controller object depends on how you create it. If you are creating your view controller programmatically, pass the name of the nib file to the `initWithNibName:bundle:` method when you initialize your view controller object. If you are loading your view controller object from a separate nib file (one other than the one containing the view), use Interface Builder to set the value NIB Name attribute of the view controller object to the name of your view's nib file.

Listing 2-1 shows an example of how you would create and initialize a view controller programmatically. In this case, the custom class uses a nib file with the same name to store its view. After initializing the view controller, you can use the view controller as you see fit, including presenting it to the user, as is done in the example.

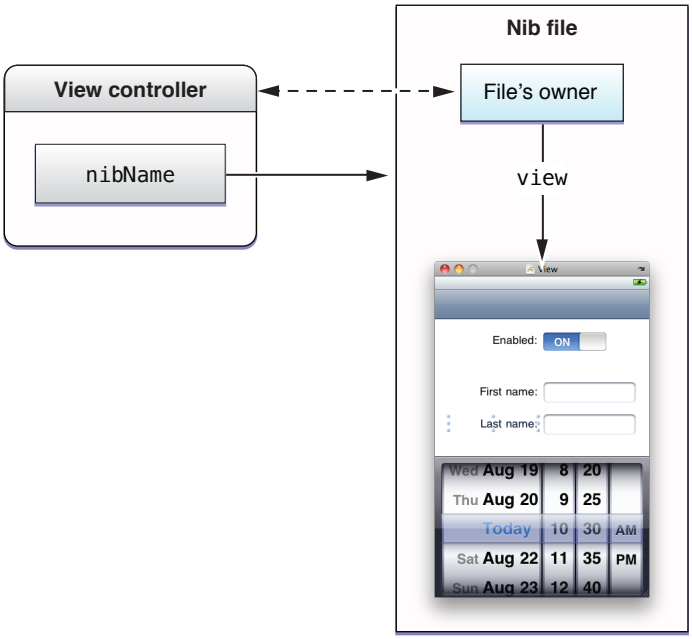
Listing 2-1 Creating a view controller object programmatically

```
- (void)displayModalView
{
   MyViewController* vc = [[[MyViewController alloc]
initWithNibName:@"MyViewController"
                                bundle:nil] autorelease];
   [self presentModalViewController:vc animated:YES];
}
```

For a detached nib file, the actual loading of the nib file occurs automatically when the `view` property of the view controller object is accessed and the view is not currently in memory. The default `loadView` method uses the `nibName` and `nibBundle` properties to locate the desired nib file and load its contents into memory.

Figure 2-4 shows the runtime configuration of a view controller and its detached nib file prior to loading. The `nibName` property of the view controller stores a string with the name of the nib file. This string is used to locate the nib file in the application's bundle. Inside the nib file, the File's Owner placeholder stands in for the view controller object and is used to connect the view controller's outlets and actions to objects in the nib file. After the nib file is loaded, the `view` property of the view controller object points to the view from the nib file.

Figure 2-4 Loading a view from a detached nib file



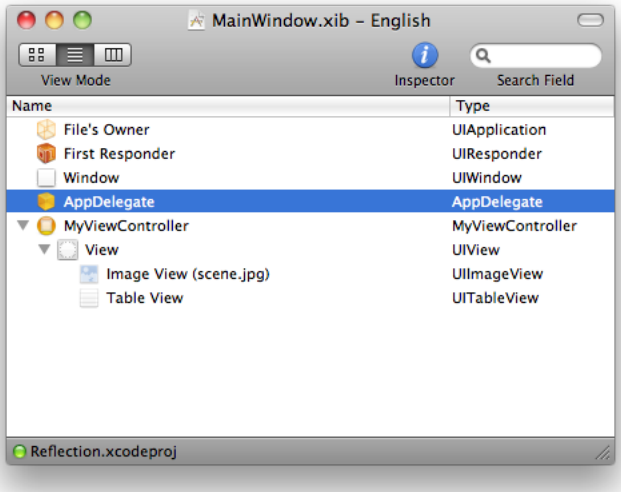
For more information about how to create a nib file or configure its contents, see *Interface Builder User Guide*. For information about configuring custom outlets and actions for your view controller, see "Configuring Actions and Outlets for Your View Controller" (page 38).

Storing the View and View Controller in the Same Nib File

If your application has only one screen, you can include both the views for that screen and the view controller that manages them in the same nib file. Storing views and custom view controller objects in the same nib file is generally not recommended because it often prevents the system from unloading the views in low-memory situations. However, if the view itself is never going to be unloaded, including it in the same nib file as its view controller object might make the most sense.

Figure 2-5 shows the main nib file of an application that presents a single screen in its window. In this case, the nib file includes both a custom view controller object (`MyViewController`) and the view managed by that view controller. Notice that the view object is nested inside the view controller object in the document window. Nesting the view in this manner is preferred because it allows Interface Builder to keep both the view and its view controller in sync.

Figure 2-5 Embedding a view controller in a nib file



To configure the nib file shown in Figure 2-5, you would do the following:

1. Drag a View Controller (`UIViewController`) object from the library to your Interface Builder document window.
2. Add a generic View object to the view controller in one of the following ways:
    - Drag the view to the View Controller's workspace window.
    - Drag the view to the view controller object in the Interface Builder document window.

    Important: Always drag the view to the view controller object in order to nest it inside the view controller (as shown in Figure 2-5 (page 36)). In Interface Builder, view controller objects take into account the presence of the status bar, translucency effects, and other factors that can affect the position of the view inside its parent window. If you do not embed the view inside the view controller object, these factors are not taken into account and your view may be positioned incorrectly.
3. Drag an Image View from the library to the generic view.
4. Drag a Table View from the library to the generic view.
5. Save the nib file.

Note: Although the preceding example used a generic `UIView` object as the view controller's root view, you can add any single view you want, including custom subclasses of `UIView` that you define. To use one of your own subclasses, add a generic `UIView` object and change its class name using the Identity Inspector.

Whenever you add objects to the top-level of your nib file, you should always connect those objects to outlets somewhere else in the nib file. In the case of the preceding nib file, this would mean defining an outlet in your application delegate object and connecting that outlet to the custom view controller object. Without the outlet, your code would have no way to access the view controller at runtime. In addition, because top-level objects are retained and then autoreleased, if you did not retain the object, it might possibly be released before you had a chance to use it.

Because they are stored in the same nib file, both the view controller and its view are ready to be used after the nib file has been loaded into memory. In the case of the main nib file, you would typically include some code in your application delegate's `applicationDidFinishLaunching:` method to add the view controller's view to your window, as described in "Presenting a View Controller's View" (page 53).

Configuring the View Display Attributes in Interface Builder

To help you lay out the contents of your view properly, Interface Builder provides controls that let you specify whether the view has a navigation bar, a toolbar, or other objects that might affect the position of your custom content. Table 2-1 lists the items that you can configure and the impact they have on your view controller or views. For a detached nib file, you configure these items by modifying attributes of the view object. For integrated nib files, you configure these items by modifying attributes of the view controller object.

Table 2-1 Configurable items for a view controller

Configurable items	Description
Status bar	You can specify whether the status bar is visible and what type of status bar your application displays by changing the Status Bar attribute in the Attributes inspector. This attribute is for design purposes only and is provided so that you can get a complete picture of how your views and controls will look when they are displayed with the status bar. The value of Status Bar attribute is not saved with your nib file. The actual style of the status bar (and whether it is present) must be set programmatically by your application at runtime.
Navigation bar	You can specify whether your view has a navigation bar by changing the value of the Top Bar attribute in the Attributes inspector. This attribute is for design purposes only and is provided so that you can get a complete picture of how your views and controls will look when displayed with a navigation bar. Interface Builder lets you configure several different navigation bar styles, including navigation bars with extra space for prompt text. The value of the Top Bar attribute is not saved with your nib file. The actual style of the navigation bar (and whether it is present) is controlled by the owning navigation controller. For information on how to configure the navigation bar, see "Customizing the Navigation Bar Appearance" (page 75).

Configurable items	Description
Tab bar	You can specify whether your view has a tab bar by changing the value of the Bottom Bar attribute in the Attributes inspector. This attribute is primarily for design purposes and is provided so that you can get a complete picture of how your views and controls will look in the presence of a tab bar. If you are actually configuring a tab bar controller, tab bar items may be added to the individual view controllers associated with the tabs of the tab bar interface. For more information about configuring tab bar controllers and tab bar items, see "Creating a Tab Bar Interface" (page 88).
Toolbar items	To specify that your view uses the toolbar provided by a navigation controller, set the Bottom Bar attribute in the Attributes inspector to Toolbar. This attribute is primarily for design purposes when creating a navigation interface. You can use it to see how your views and controls will look in the presence of a tab bar. If you are actually configuring a navigation controller, you can also include the bar button items for your view controller's toolbar in your nib file. For more information on configuring toolbars in a navigation interface, see "Displaying a Navigation Toolbar" (page 81).
Title	In an integrated nib file, you can specify a title for your view controller by assigning an appropriate value to the Title attribute. Navigation and tab bar controllers use the value of this attribute as a default value to use when displaying your view controller.
Nib name	For view controllers embedded in a nib file, you use the NIB Name attribute to specify the name of the detached nib file containing the view controller's view. For information on how to configure a detached nib file, see "Storing the View in a Detached Nib File" (page 33). When the view controller is loaded into memory at runtime, the value for this attribute is used to set the value of the view controller's `nibName` property.

The configuration of navigation and tab bar controllers in nib files is a little more involved and is discussed in the following places:

- For information about how to configure a navigation controller object by itself, see "Creating a Navigation Interface" (page 66).
- For information about how to configure a tab bar controller by itself, see "Creating a Tab Bar Interface" (page 88).
- For information about how to configure combinations of navigation and tab bar controllers, see "Adding a Navigation Controller to a Tab Bar Interface" (page 119).

Configuring Actions and Outlets for Your View Controller

Regardless of whether you use a detached or integrated nib file, the way you configure the actions and outlets of your view controller is essentially the same. Using Interface Builder, you create connections between the views and controls in your interface to the object representing your view controller. In an integrated nib file, you can make connections directly to the view controller object. However, in a detached nib file, you make connections to the File's Owner placeholder, which stands in for your view controller object.

Listing 2-2 shows the definition of a custom `MyViewController` class that defines two custom outlets (designated by the `IBOutlet` keyword) and a single action method (designated by the `IBAction` return type). The outlets store references to a button and a text field in the nib file, while the action method responds to taps in the button.

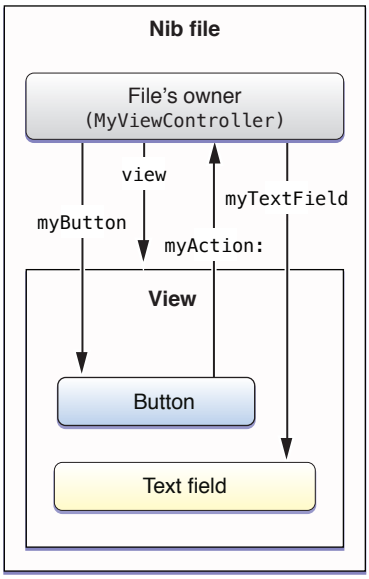
Listing 2-2 Custom view controller class declaration

```
@interface MyViewController : UIViewController
{
  id myButton;
  id myTextField;
}
@property (nonatomic) IBOutlet id myButton;
@property (nonatomic) IBOutlet id myTextField;

- (IBAction)myAction:(id)sender;
```

Figure 2-6 shows the connections you would create among the objects in such a `MyViewController` class. This nib file is configured as per the instructions in "Configuring Actions and Outlets for Your View Controller" (page 38), with the class of the File's Owner placeholder set to the view controller class and the `view` outlet of File's Owner connected to the top-level view object. The nib file also contains connections between the outlets and actions of the `MyViewController` class and the corresponding nib file objects.

Figure 2-6 Contents of `MyViewController.nib`



When the previously configured `MyViewController` class is created and presented modally, the view controller infrastructure loads the nib file automatically and reconfigures any outlets or actions. Thus, by the time the view is presented to the user, the outlets and actions of your view controller are set and ready to be used. This ability to bridge between your runtime code and your design-time resource files is one of the things that makes nib files very powerful.

Creating the View Programmatically

If you prefer to create views programmatically, instead of using a nib file, you do so from your view controller's `loadView` method. You must override this method if you plan to create your views programmatically. Your implementation of this method should do the following:

1. Create a root view object that is sized to fit the screen.

   The root view acts as the container for all other views associated with your view controller. You typically define the frame for this view to match the size of the application window, which itself should fill the screen. However, the view controller also adjusts the frame size as needed to accommodate the presence of assorted views, such as the system status bar, a navigation bar, or a tab bar.

   You can use a generic `UIView` object, a custom view you define, or any other view that can scale to fill the screen.

2. Create any additional subviews and add them to the root view. For each view, you should do the following:

   a. Create and initialize the view. For system views, you typically use the `initWithFrame:` method to specify the initial size and position of the view.

   b. Add the view to a parent view using the `addSubview:` method.

   c. Release the view by calling its `release` method.

3. Assign the root view to the `view` property of your view controller.

4. Release the root view.

The notion of releasing each view shortly after creating it might sound strange, but once you add a view to your view hierarchy or save a reference to it, that is exactly what you want to do. The initial retain count of any object is 1 at creation time. Because parent views automatically retain their subviews, it is safe to release child views after they have been added to their parent. Similarly, the `view` property used to store your root view uses retain semantics to prevent the view from being released. Therefore, releasing each view transfers ownership to the appropriate place and prevents the object from being leaked later.

Listing 2-3 shows the `loadView` method from the `MetronomeViewController` class of the sample project *Metronome*. This method creates a custom view and does some basic setup, including assigning the custom view to the view controller's `view` property, which retains the view. In this example, the `metronomeView` property of the view controller is an additional property that stores a pointer to the view; however, this property uses assignment semantics to avoid potential retention problems.

Listing 2-3 Creating views programmatically in the Metronome application

```
- (void)loadView {

    self.wantsFullScreenLayout = YES;

    MetronomeView *view = [[MetronomeView alloc]
                        initWithFrame:[UIScreen mainScreen].applicationFrame];
    view.metronomeViewController = self;
    self.view = view;
    self.metronomeView = view;
```

```
    [view release];
}
```

Note: When overriding the `loadView` method to create your views programmatically, you should not call `super`. Doing so initiates the default view-loading behavior and is usually just a waste of CPU cycles. Your own implementation of the `loadView` method should do all the work that is needed to create a root view and subviews for your view controller. For more information on the view loading process, see "Understanding the View Management Cycle" (page 30).

If you retain any view objects using properties or custom setter methods, you should always remember to implement a `viewDidUnload` method to set those properties to `nil`. This is especially true if you use outlets to store references to your views and those outlets use a property or other setter method with retain semantics. For more information about managing the memory associated with your views, see "Managing Memory Efficiently" (page 41).

Cleaning Up After Unloading a View

After it is loaded into memory, a view controller's view remains in memory until a low-memory condition occurs or the view controller itself is deallocated. In the case of a low-memory condition, the default `UIViewController` behavior is to release the view object stored in the `view` property if that view is not currently being used. However, if your custom view controller class stores outlets or pointers to any views in the view hierarchy, you must also release those references when the top-level view object is released. Failure to do so prevents those objects from being removed from memory right away and could potentially cause memory leaks later if you subsequently overwrite any pointers to them.

There are two places where your view controller should always clean up any references to view objects:

- The `dealloc` method
- The `viewDidUnload` method

If you use a declared property to store a reference to your view, and that property uses retain semantics, assigning a `nil` value to it is enough to release the view. Properties are by far the preferred way to manage your view objects because of their convenience. If you do not use properties, you must send a `release` message to any view that you explicitly retained before setting the corresponding pointer value to `nil`.

For information about memory management best practices, see "Managing Memory Efficiently" (page 41).

Managing Memory Efficiently

When it comes to view controllers and memory management, there are two issues to consider:

- How do you allocate memory efficiently?
- When and how do you release memory?

Although some aspects of memory allocation are strictly yours to decide, there are a handful of methods in the `UIViewController` class that usually have some connection to memory management tasks. Table 2-2 lists the places in your view controller object where you are likely to allocate or deallocate memory along with information about what you should be doing in each place.

Table 2-2 Places to allocate and deallocate memory

Task	Methods	Discussion
Allocating critical data structures required by your view controller	Initialization methods	Your custom initialization method (whether it is named `init` or something else) is always responsible for putting your view controller object in a known good state. This includes allocating whatever data structures are needed to ensure proper operation.
Creating your view objects	`loadView`	Overriding the `loadView` method is required only if you intend to create your views programmatically. If you are loading your views from a nib file, all you have to do is use the `initWithNibName: bundle:` method to initialize your view controller with the appropriate nib file information.
Allocating or loading data to be displayed in your view	`viewDidLoad`	Any data that is tied to your view objects should be created or loaded in the `viewDidLoad` method. By the time this method is called, your view objects are guaranteed to exist and be in a known good state.
Releasing references to view objects	`viewDidUnload` `dealloc`	If you retain any view objects in your view hierarchy using outlets or other member variables in your class, you must always release those outlets when the views are no longer needed. After releasing a view object, always be sure to set your outlet or variable to `nil` for safety. For more information about when views get released, see "Understanding the View Management Cycle" (page 30).
Releasing data that is not needed when your view is not displayed	`viewDidUnload`	You can use the `viewDidUnload` method to deallocate any data that is view-specific and that can be recreated easily enough if the view is loaded into memory again. If recreating the data might be too time-consuming, though, you do not have to release the corresponding data objects here. Instead, you should consider releasing those objects in your `didReceiveMemoryWarning` method.
Responding to low-memory notifications	`didReceiveMemory-Warning`	Use this method to deallocate all noncritical custom data structures associated with your view controller. Although you would not use this method to release references to view objects, you might use it to release any view-related data structures that you did not already release in your `viewDidUnload` method. (The view objects themselves should always be released in the `viewDidUnload` method.)

Task	Methods	Discussion
Releasing critical data structures required by your view controller	`dealloc`	Use this method to release all data structures associated with your view controller. If your view controller still has outlets or other variables with non-`nil` values, you should release them here.

Managing a View Controller's Interface Orientation

The accelerometers in iOS–based devices make it possible to determine the current orientation of the device. The UIKit framework takes advantage of this information and uses it to orient your application's user interface to match the device orientation when appropriate. Although applications support only a portrait orientation by default, you can configure your view controllers to support other orientations as needed.

Supporting alternate orientations requires additional configuration for both your views and the view controllers that manage them. The simplest way to support multiple interface orientations is to do the following:

- Override your view controller's `shouldAutorotateToInterfaceOrientation:` method and declare the orientations it supports; see "Declaring the Supported Interface Orientations" (page 44).
- Configure the autoresizing mask for each view in your view controller's view hierarchy; see "Configuring Your Views to Support Multiple Orientations" (page 45).

These two steps should be sufficient for many applications. However, if the autoresizing behavior of views does not yield the layout you need for each orientation, you can override additional view controller methods and use them to tweak your layout when orientation changes occur. The `UIViewController` class provides a series of notifications that let you respond to different phases of the orientation change and make adjustments to your views (or other parts of your application) as needed. These notifications are described in more detail in "Responding to Orientation Changes" (page 45)

Understanding the Rotation Process

When the orientation of an iOS–based device changes, the system sends out a `UIDeviceOrientationDidChangeNotification` notification to let any interested parties know that the change occurred. By default, the UIKit framework intercepts this notification and uses it to update your interface orientation automatically. This means that with only a few exceptions, you should not need to handle this notification at all. Instead, all you need to do is implement the appropriate methods in your view controller classes to respond to orientation changes.

In an iOS application, the window object does much of the work associated with changing the current orientation. However, it works in conjunction with the application's view controllers to determine whether an orientation change should occur at all, and if so, what additional methods should be called to respond to the change. Specifically, it works with the view controller whose root view was most recently added to, or presented in, the window. In other words, the window object works only with the frontmost view controller whose view was displayed using one of the mechanisms described in "Presenting a View Controller's View" (page 53).

The actual rotation process proceeds along one of two paths depending on the implementation of the associated view controller. The most common path is to perform a one-step rotation but view controllers can also support a two-step rotation if it provides a better experience. The one-step rotation process is available in iOS 3.0 and later and is preferred because it is more efficient than the two-step process. The choice as to which path is taken is dependent on your view controller subclass and which methods you override. If you override any of the methods associated with the one-step process, the window object uses that process; otherwise, it uses the two-step process.

Regardless of which rotation process is used, methods of the view controller are called at various stages of the rotation to give the view controller a chance to perform additional tasks. You might use these methods to hide or show views, reposition or resize views, or notify other parts of your application about the orientation change. Because your custom methods are called during the rotation operation, you should avoid performing any time-consuming operations there. You should also avoid replacing your entire view hierarchy with a new set of views. There are better ways to provide unique views for different orientations, such as presenting a new view controller modally (as described in "Creating an Alternate Landscape Interface" (page 50)).

For detailed information about the sequence of steps that occur during the one-step and two-step rotation processes, see "Responding to Orientation Changes" (page 45).

Declaring the Supported Interface Orientations

If the view managed by your view controller supports orientations other than the default portrait orientation, you must override the `shouldAutorotateToInterfaceOrientation:` method and indicate which orientations your view supports. You should always choose the orientations your view supports at design time and implement your code with those orientations in mind. There is no benefit to choosing which orientations you want to support dynamically based on runtime information. Even if you did so, you would still have to implement the necessary code to support all possible orientations, and so you might as well just choose to support the orientation or not up front.

Listing 2-4 shows a fairly typical implementation of the `shouldAutorotateToInterfaceOrientation:` method for a view controller that supports the default portrait orientation and the landscape-left orientation. Your own implementation of this method should be just as simple.

Listing 2-4 Implementing the `shouldAutorotateToInterfaceOrientation:` method

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)orientation
{
    if ((orientation == UIInterfaceOrientationPortrait) ||
        (orientation == UIInterfaceOrientationLandscapeLeft))
       return YES;

    return NO;
}
```

Important: You must always return YES for at least one interface orientation.

If your application supports both landscape orientations, you can use the `UIInterfaceOrientationIsLandscape` macro as a shortcut, instead of explicitly comparing the *`orientation`* parameter against both landscape constants. The UIKit framework similarly defines a `UIInterfaceOrientationIsPortrait` macro to identify both variants of the portrait orientation.

Configuring Your Views to Support Multiple Orientations

When the user interface changes orientation, the bounds of the affected views are modified automatically according to their autoresizing mask. The `autoresizingMask` property of every view contains constants that describe how the bounds of that view change in relation to its superview. Each view adjusts its own bounds and then asks each of its subviews to resize itself based on its autoresizing behaviors. The net result is that if you configure the autoresizing behaviors of your views appropriately, your views can adjust to orientation changes automatically.

If the autoresizing behaviors of views do not provide the precise layout you need, you may want to replace or supplement their behavior by providing your own custom layout code. The `UIViewController` class defines several methods that are called before, during, and after an orientation change. You can use these methods to modify the layout of your views as needed.

For information about the methods that are called during the rotation process, see "Responding to Orientation Changes" (page 45). For more information on the autoresizing properties of views and how they affect the view, see *View Programming Guide for iOS*.

Responding to Orientation Changes

When the orientation of a device changes, view controllers can respond by making a corresponding change to the orientation of their view. If the new orientation is supported, the view controller generates notifications as it makes the change to give your code a chance to respond. Rotation notifications can occur as a one-step or two-step process.

The reason you might want to respond to orientation changes is to make adjustments to your view hierarchy. For example, you might use these notifications to make the following types of changes:

- Show or hide views that are specific to a particular orientation.
- Adjust the position or size of views based on the new orientation.
- Update other parts of your application to reflect the orientation change.

Whenever possible, the UIKit framework uses the one-step rotation process to rotate your views. However, whether it actually uses the one-step or two-step process is up to you. There are methods used by the two-step process that are not used by the one-step process, and vice versa; if you override any of the one-step methods, the one-step process is used. If you override only methods that are used by the two-step process, that process is used. The following sections describe the methods associated with each process. You can also find information about these methods (including whether they trigger the one-step or two-step process) in *UIViewController Class Reference*.

Responding to Orientation Changes in One Step

In iOS 3.0 and later, you can use one-step rotation methods to make changes just before and just after the orientation change occurs. During this process, the following sequence of events occurs:

1. The window detects that a change in the device orientation has occurred.
2. The window looks for an appropriate view controller and calls its `shouldAutorotateToInterfaceOrientation:` method to determine if it supports the new orientation.

Container view controllers may intercept this method and use their own heuristics to determine whether the orientation change should occur. For example, the tab bar controller allows orientation changes only if all of its managed view controllers support the new orientation.

3. If the new orientation is supported, the window calls the view controller's `willRotateToInterfaceOrientation:duration:` method.

   Container view controllers forward this message on to the currently displayed custom view controller. You can override this method in your custom view controllers to hide views or make other changes to your view layout before the interface is rotated.

4. The window adjusts the bounds of the view controller's view.

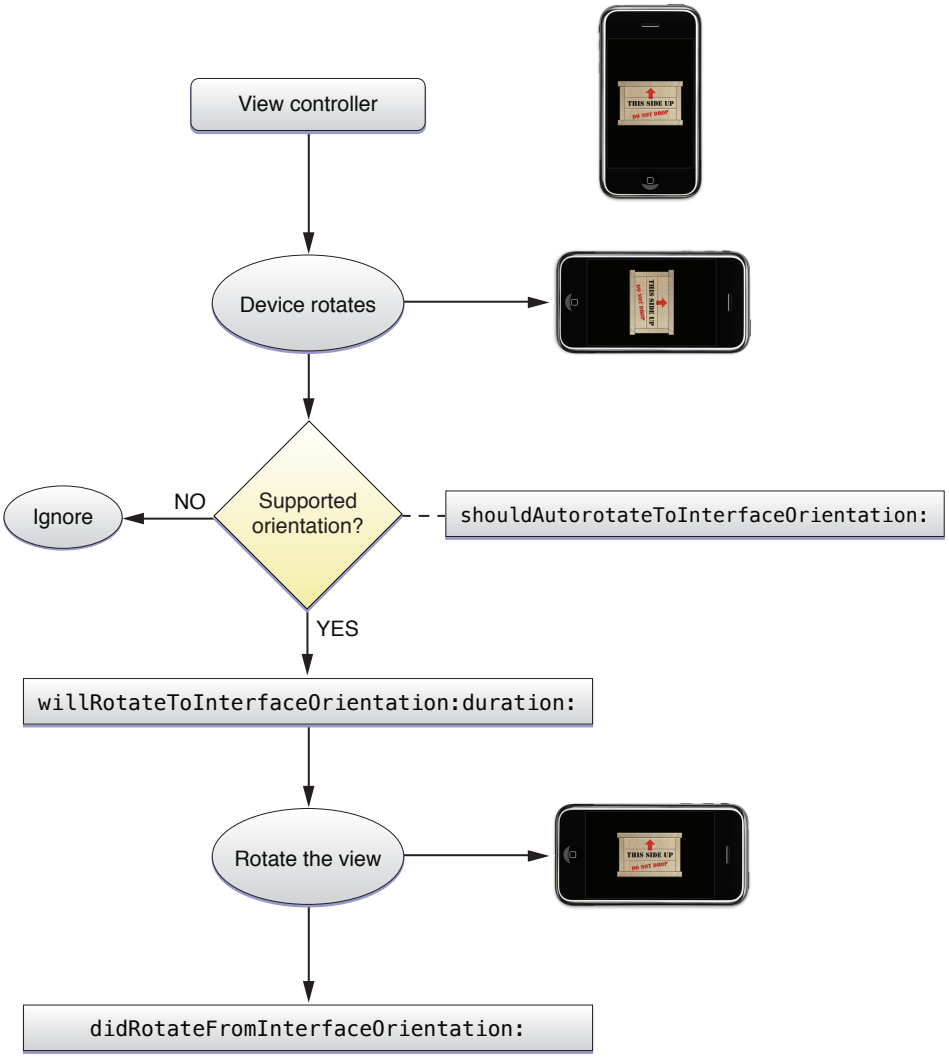
   This causes each view in the view hierarchy to be resized based on its autoresizing mask.

5. The window calls the view controller's `didRotateFromInterfaceOrientation:` method.

   Container view controllers forward this message to the currently displayed custom view controller. This marks the end of the rotation process. You can use this method to show views, change the layout of views, or make other changes to your application.

Figure 2-7 shows a visual representation of the preceding steps. It also shows how the interface would look at various stages of the process.

Figure 2-7 Processing a one-step interface rotation



Responding to Orientation Changes in Two Steps

In all versions of iOS, you can use two-step notifications to respond to interface orientation changes. In the two-step process, two separate rotations occur. During the first step, the interface is rotated only halfway to its destination. During the second step, the rotation is rotated from this midpoint to its final orientation. Your application receives notifications throughout the process that allow you to respond before, during, and after the rotation.

The following sequence of events occurs during a two-step rotation:

1. The window detects that a change in the device orientation has occurred.
2. The window looks for an appropriate view controller and calls its `shouldAutorotateToInterfaceOrientation:` method to determine if it supports the new orientation.

Container view controllers may intercept this method and use their own heuristics to determine whether the orientation change should occur. For example, the tab bar controller allows orientation changes only if all of its managed view controllers support the new orientation.

3. If the new orientation is supported, the window calls the view controller's `willRotateToInterfaceOrientation:duration:` method.

   Container view controllers forward this message to the currently displayed custom view controller. You can use this method to hide views or make other changes to your view layout before the interface is rotated.

4. The window calls the view controller's `willAnimateFirstHalfOfRotationToInterfaceOrientation:duration:` method.

   Container view controllers forward this message to the currently displayed custom view controller. You can use this method to hide views or make other changes to your view layout before the interface is rotated.

5. The window performs the first half of the rotation.

   This causes the bounds of each view in the view hierarchy to be adjusted based on its autoresizing behaviors. Although most rotations involve moving from portrait to landscape mode (and thus rotating 45 degrees to the halfway point), it is possible to rotate from a landscape left to landscape right orientation or from a portrait to upside down portrait orientation. In these latter cases, the first half of the rotation would be 90 degrees.

6. The window calls the view controller's `didAnimateFirstHalfOfRotationToInterfaceOrientation:` method.

   Container view controllers forward this message to the currently displayed custom view controller.

7. The window calls the view controller's `willAnimateSecondHalfOfRotationFromInterfaceOrientation:duration:` method.

   Container view controllers forward this message to the currently displayed custom view controller.

8. The window performs the second half of the rotation.

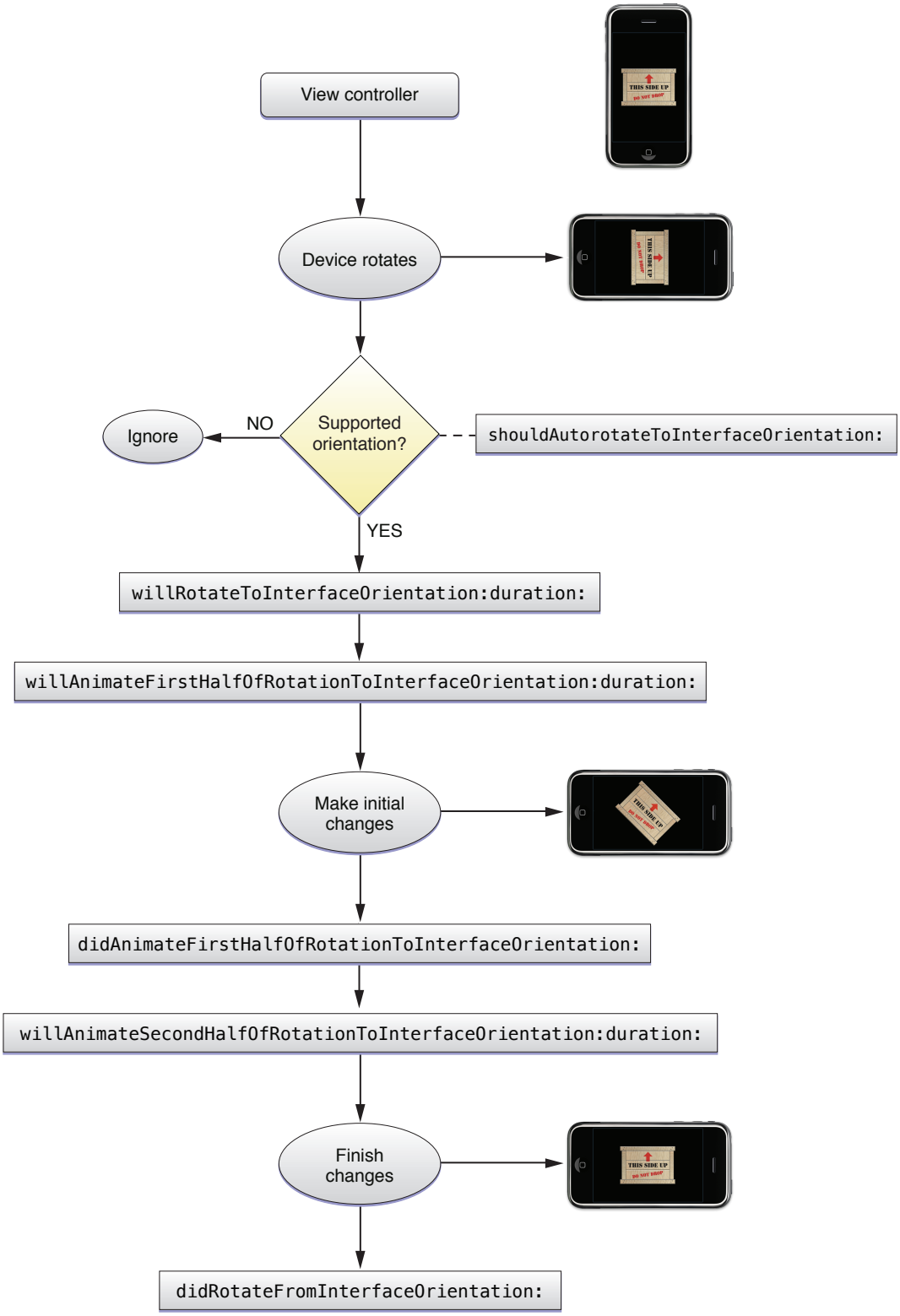
   By the end of this rotation, the autoresizing behaviors for all views have been applied and the views are in their "final" position.

9. The window calls the view controller's `didRotateFromInterfaceOrientation:` method.

   Container view controllers forward this message to the currently displayed custom view controller. You can use this method to show views, reposition or resize views, or make other changes once the rotation has finished.

Figure 2-8 shows a visual representation of the preceding steps. Along with each step, it shows how the view would appear to the user.

Figure 2-8 Processing a two-step interface rotation



Creating an Alternate Landscape Interface

If you want to present the same data differently based on whether a device is in a portrait or landscape orientation, the way to do so is using two separate view controllers. One view controller should manage the display of the data in the primary orientation (typically portrait) while the other manages the display of the data in the alternate orientation. Using two view controllers is simpler and more efficient than making major changes to your view hierarchy each time the orientation changes. It allows each view controller to focus on the presentation of data in one orientation and to manage things accordingly. It also eliminates the need to litter your view controller code with conditional checks for the current orientation.

In order to support an alternate landscape interface, you have to do the following:

- Implement two view controller objects:
    - One should present a portrait-only interface.
    - One should present a landscape-only interface.
- Register for the `UIDeviceOrientationDidChangeNotification` notification. In your handler method, present or dismiss the alternate view controller based on the current device orientation.

Because view controllers normally manage orientation changes internally, you have to tell each view controller to display itself in one orientation only. The implementation of the primary view controller then needs to detect device orientation changes and present the alternate view controller when the appropriate orientation change occurs. When the orientation returns to the primary orientation, the primary view controller would then dismiss the alternate view controller.

Listing 2-5 shows the key methods you would need to implement in a primary view controller that supports a portrait orientation. As part of its initialization, this view controller registers to receive orientation changed notifications from the shared `UIDevice` object. When such a notification arrives, the `orientationChanged:` method then presents or dismisses the landscape view controller depending on the current orientation.

Listing 2-5 Presenting the landscape view controller

```
@implementation PortraitViewController
- (id)init
{
   self = [super initWithNibName:@"PortraitView" bundle:nil];
   if (self)
   {
       isShowingLandscapeView = NO;
       self.landscapeViewController = [[[LandscapeViewController alloc]
                           initWithNibName:@"LandscapeView" bundle:nil]
autorelease];

      [[UIDevice currentDevice] beginGeneratingDeviceOrientationNotifications];
       [[NSNotificationCenter defaultCenter] addObserver:self
                                selector:@selector(orientationChanged:)
                                name:UIDeviceOrientationDidChangeNotification
                                object:nil];
   }
   return self;
}
```

```
- (void)orientationChanged:(NSNotification *)notification
{
    UIDeviceOrientation deviceOrientation = [UIDevice currentDevice].orientation;
    if (UIDeviceOrientationIsLandscape(deviceOrientation) &&
        !isShowingLandscapeView)
    {
        [self presentModalViewController:self.landscapeViewController
                                 animated:YES];
        isShowingLandscapeView = YES;
    }
    else if (UIDeviceOrientationIsPortrait(deviceOrientation) &&
             isShowingLandscapeView)
    {
        [self dismissModalViewControllerAnimated:YES];
        isShowingLandscapeView = NO;
    }
}
```

Tips for Implementing Your Rotation Code

Depending on the complexity of your views, you may need to write a lot of code to support rotations or none at all. When figuring out what you need to do, you can use the following tips as a guide for writing your code.

- **Disable event delivery temporarily during rotations.** Disabling event delivery for your views prevents unwanted code from executing while an orientation change is in progress.
- **Store the visible map region.** If your application contains a map view, save the visible map region value prior to the beginning of rotations. When the rotations finish, use the saved value as needed to ensure that the displayed region is approximately the same as before.
- **For complex view hierarchies, replace your views with a snapshot image.** If animating large numbers of views is causing performance issues, temporarily replace those views with an image view containing an image of the views instead. Once the rotations are complete, reinstall your views and remove the image view.
- **Reload the contents of any visible tables after a rotation.** Forcing a reload operation when the rotations are finished ensures that any new table rows exposed are filled appropriately.
- **Use rotation notifications to update your application's state information.** If your application uses the current orientation to determine how to present content, use the rotation methods of your view controller (or the corresponding device orientation notifications) to note those changes and make any necessary adjustments.

Creating Custom View Controller Objects at Runtime

There are two ways to create a custom view controller object: programmatically or using a nib file. Which technique you should use depends on the structure of your user interface. For complex interfaces that incorporate tab bar controllers and navigation controllers, you usually include at least a few custom view controllers in your application's main nib file and create the rest programmatically. In most other situations, you should create view controllers programmatically and only as they are needed.

To create a custom view controller programmatically, you could use code similar to the following:

```
MyViewController* vc = [[MyViewController alloc]
initWithNibName:@"MyViewController"
                              bundle:nil];
```

Such an example assumes that your view controller does not perform any significant initialization, or if it does, it does so in an overridden version of the `initWithNibName:bundle:` method. A common approach to take when designing custom view controllers though is to define one or more custom initialization methods. Doing so allows you to hide some of the more immutable aspects of the view controller initialization (such as specifying the nib file and bundle names) and focus instead on the data you want to use to initialize the view controller. For example, a custom initialization method that takes an array of objects might look something like the following:

```
- (id)initWithData:(NSArray*)data {
    if ((self = [super initWithNibName:@"MyViewController" bundle:nil])) {
        // Initialize the view controller with the starting data
    }
    return self;
}
```

Using such an initialization method, you would create and initialize the view controller so that it is immediately ready for use. You could then present the view controller or add it to a navigation interface. For example, to present the new view controller modally, you might define a method similar to the following on the current view controller:

```
- (void)presentModalViewControllerWithData:(NSArray*)data {
   MyViewController* vc = [[MyViewController alloc] initWithData:data];
   [self presentModalViewController:vc animated:YES];
}
```

When creating view controllers programmatically, it is your responsibility to set the frame of the view controller's view appropriately before using the view. A programmatically created view controller that loads its view from a nib file does not try to change the size or position of that view. If the view controller is presented modally or used with a container view controller, the parent or container view controller often adjusts the view for you. But in cases where you add the view to a window yourself, the view's existing frame is used as is. If your application has a status bar, failing to adjust the view's frame could cause the misplacement of the view underneath the status bar.

The process for creating view controllers using a nib file is somewhat more involved and is described in "Storing the View and View Controller in the Same Nib File" (page 35). For information on how to display your view controller's view at runtime, see "Presenting a View Controller's View" (page 53).

Presenting a View Controller's View

There are several options for displaying the view associated with a view controller:

- Display the view directly by adding it to a window using the `addSubview:` method.
- Display the view indirectly using one of the following techniques:
  - Present the owning view controller modally using the `presentModalViewController:animated:` method.
  - Push the owning view controller onto the navigation stack of a navigation controller object.
  - Make the owning view controller the root view controller of a tab in a tab bar interface.
  - On iPad, present the view controller using a popover.

Listing 2-6 shows an example of how to display a view directly in the application's main window. In this example, the `viewController` variable is an outlet that stores a pointer to a view controller loaded from the main nib file. Similarly, the `window` variable stores a pointer to the application window. (This exact code is created for you in your application delegate when you use the View-based Application project template.) Adding the view to the window loads the view and causes it to be displayed when the window is subsequently shown.

Listing 2-6 Adding a view controller's view to a window

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
    // At this point, the main nib file is loaded.
    // It is a good idea to set the view's frame before adding it to a window.
    [viewController.view setFrame:[[UIScreen mainScreen] applicationFrame]];

    [window addSubview:viewController.view];
    [window makeKeyAndVisible];
}
```

Important: If you plan to add a view to a window using the `addSubview:` method, it is recommended that you explicitly set the frame of the view after loading it from a nib file. Although nib files store the most appropriate size for root views, setting the frame explicitly guarantees that your view is sized and positioned correctly within the window. If you plan to present a view modally or use it in conjunction with a container view controller, you do not need to set the frame explicitly. However, if you create your view controller object programmatically, you should always set the frame of its view before use.

It is recommended that you use only the suggested techniques for displaying the views of your view controllers. In order to present and manage views properly, the system makes a note of each view (and its associated view controller) that you display directly or indirectly. It uses this information later to report view controller-related events to your application. For example, when the device orientation changes, a window uses this information to identify the frontmost view controller and notify it of the change. If you incorporate a view controller's view into your hierarchy by other means (by adding it as a subview to some other view perhaps), the system assumes you want to manage the view yourself and does not send messages to the associated view controller object.

Apart from your setting up your application's initial interface, most other views are presented indirectly through their view controller objects. For information about how to present views indirectly, consult the following sections:

- For information about how to present a view modally, see "Presenting a View Controller Modally" (page 114).
- For information about how to display views in a navigation interface, see "Modifying the Navigation Stack" (page 73).
- For information about displaying a view in a tab, see "Creating a Tab Bar Interface" (page 88).
- For information about displaying a view using a popover, see "Creating and Presenting a Popover" (page 101).

Responding to Display-Related Notifications

When the visibility of a view controller's view changes, the view controller calls some built-in methods to notify subclassers of the changes. You can use these built-in methods to respond to the change in visibility appropriately. For example, you could use these notifications to change the color and orientation of the status bar so that it matches the presentation style of the view that is about to be displayed. A view controller calls different methods depending on whether a view is about to appear or disappear from the screen.

Figure 2-9 shows the basic sequence of events that occurs when a view controller's view is added to a window. (If the view is already in a window but currently hidden by another view, this same sequence of events occurs when those obstructions are removed and the view is once again revealed.) The `viewWillAppear:` and `viewDidAppear:` methods give subclasses a chance to perform any additional actions related to the appearance of the view.

Figure 2-9 Responding to the appearance of a view

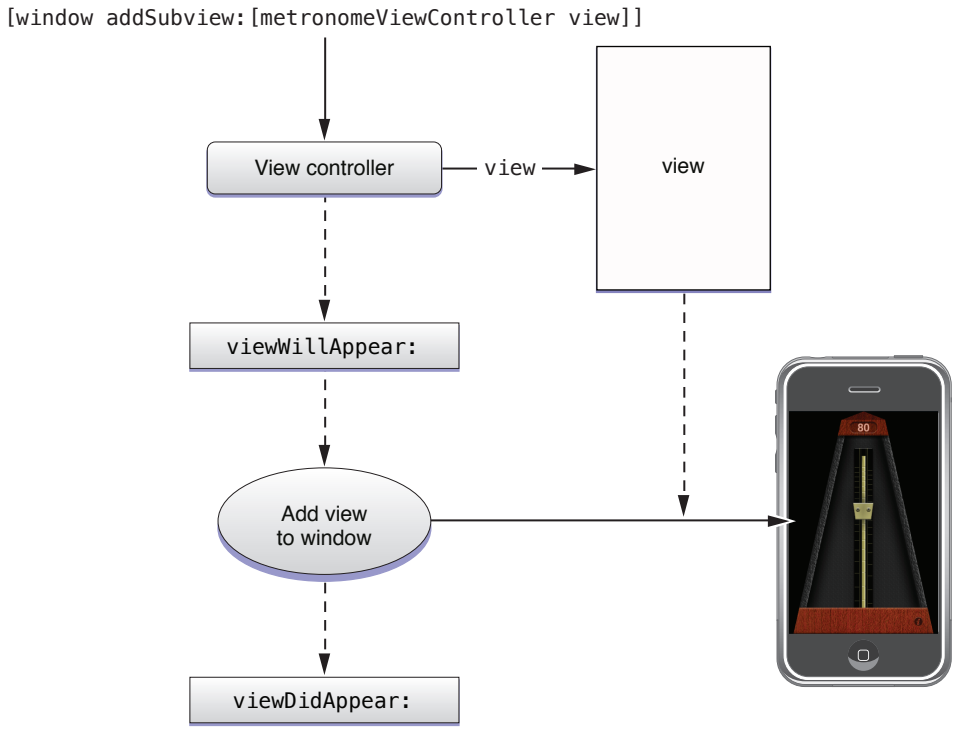
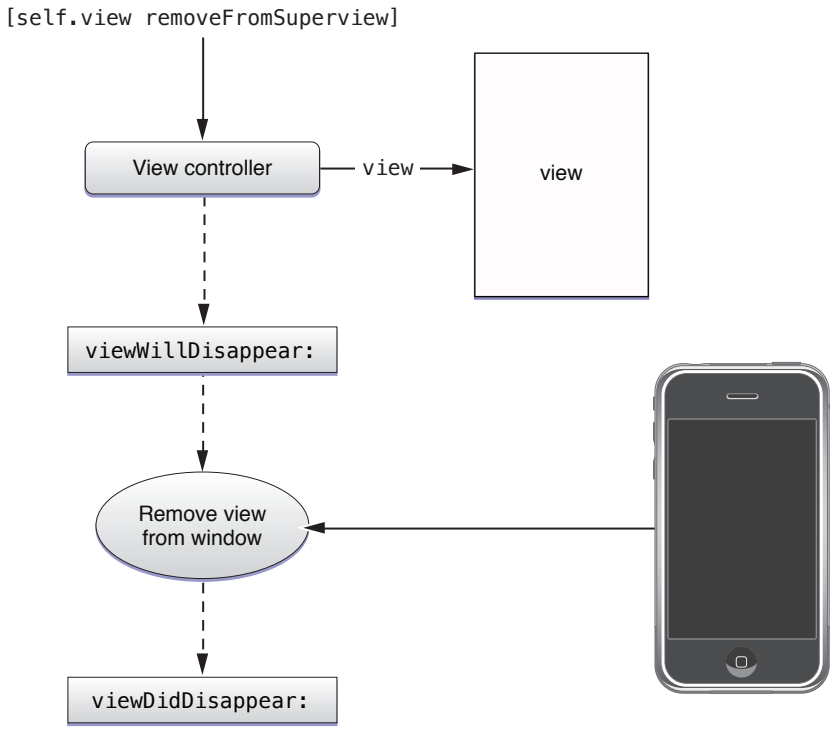


Figure 2-10 shows the basic sequence of events that occurs when a view is removed from its window. (This same sequence of events occurs when a view becomes completely hidden by another view, such as can occur when presenting new view controllers on top of the existing one.) When the view controller detects that its view is about to be removed or hidden, it calls the `viewWillDisappear:` and `viewDidDisappear:` methods to give subclasses a chance to perform any relevant tasks.

Figure 2-10 Responding to the disappearance of a view



Adopting a Full-Screen Layout for Custom Views

If your application displays the status bar, the `UIViewController` class automatically shrinks its view so that the view does not underlap the status bar. After all, if the status bar is opaque, there is no way to see the content lying underneath or interact with it. However, if your application displays a translucent status bar, you can set the value of your view controller's `wantsFullScreenLayout` property to `YES` to allow your view to underlap the status bar.

Underlapping the status bar is useful in situations where you want to maximize the amount of space available for displaying your content. When displaying content under the status bar, you should be sure to put that content inside a scroll view so that the user can scroll it out from under the status bar. Being able to scroll your content is important because the user cannot interact with content that is positioned behind the status bar or any other translucent views (such as translucent navigation bars and toolbars). Navigation bars automatically add a scroll content inset to your scroll view (assuming it is the root view of your view controller) to account for the height of the navigation bar; otherwise, you must modify the `contentInset` property of your scroll view manually.

For more information about adopting a full-screen layout for view controllers used in conjunction with a navigation controller, see "Adopting a Full-Screen Layout for Navigation Views" (page 72).

Enabling Edit Mode for a View

If you want to use the same view controller to display and edit content, you can override the `setEditing:animated:` method and use it to toggle the view controller's view between display and edit modes. When called, your implementation of this method should add, hide, and adjust the view controller's views to match the specified mode. For example, you might want to change the content or appearance of views to convey that the view is now editable. If your view controller manages a table, you can also call the table's own `setEditing:animated:` method to put the table into the appropriate mode.

Note: You typically do not swap out your entire view hierarchy when toggling back and forth between display and edit modes. In fact, the whole point of using the `setEditing:animated:` method is so that you can make small changes to existing views. If you would prefer to display a new set of views for editing, you should either present a new view controller modally or use a navigation controller to present the new views.

Figure 2-11 shows a view from the Contacts application that supports in-place editing. Tapping the Edit button in the upper-right corner tells the view controller to update itself for editing; the Done button returns the user to display mode. In addition to modifying the table, the view also changes the content of the image view and the view displaying the user's name. It also configures the assorted views and cells so that tapping them edits their contents instead of performing other actions.

Figure 2-11 Display and edit modes of a view

Display mode **Edit mode**

The implementation of your own `setEditing:animated:` method is relatively straightforward. All you have to do is check to see which mode your view controller is entering and adjust the contents of your view accordingly.

```
- (void)setEditing:(BOOL)flag animated:(BOOL)animated
{
    [super setEditing:flag animated:animated];
    if (flag == YES){
        // change views to edit mode
    }
    else {
        // save the changes if needed and change views to noneditable
    }
}
```

A common place in which to use an editable view is in a navigation interface. When implementing your navigation interface, you can include a special Edit button in the navigation bar when your editable view controller is visible. (You can get this button by calling the `editButtonItem` method of your view controller.) When tapped, this button automatically toggles between an Edit and Done button and calls your view controller's `setEditing:animated:` method with appropriate values. You can also call this method from your own code (or modify the value of your view controller's `editing` property) to toggle between modes.

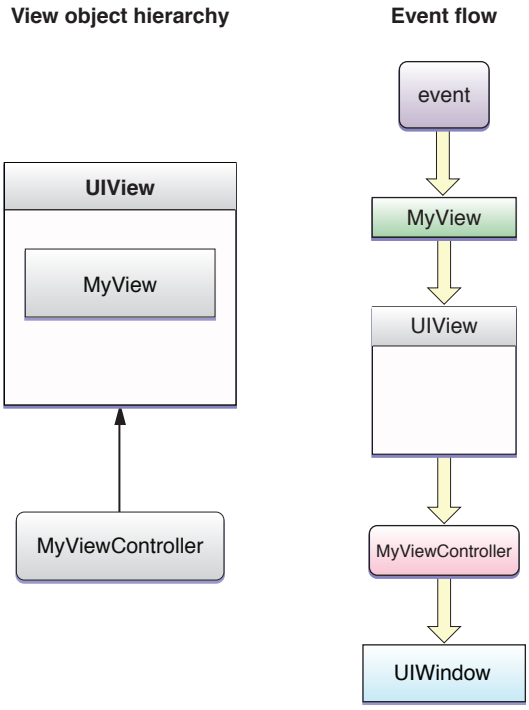
For more information about adding an Edit button to a navigation bar, see "Using Edit and Done Buttons" (page 81). For more information about how you support the editing of table views, see *Table View Programming Guide for iOS*.

Handling Events

View controllers are themselves descendants of the `UIResponder` class and are therefore capable of handling all sorts of events. Normally, when a view does not respond to a given event, it passes that event to its superview. However, if the view is being managed by a view controller, it passes the event to the view controller object first. This gives the view controller the opportunity to absorb any events that are not handled by its views. If the view controller does not handle the event, then that event moves on to the view's superview as usual.

Figure 2-12 demonstrates the flow of events within a view hierarchy. Suppose you have a custom view that is embedded inside a screen-sized generic view object, which is in turn managed by your view controller. Touch events arriving in your custom view's frame are delivered to that view for processing. If your view does not handle an event, it is passed along to the parent view. Because the generic view does not handle events, it passes those events along to its view controller first. If the view controller does not handle the event, the event is further passed along to the superview of the generic `UIView` object, which in this case would be the window object.

Figure 2-12 Responder chain for view controllers



Note: The message-passing relationship between a view controller and its view is managed privately by the view controller and cannot be programmatically modified by your application.

Although you might not want to handle touch events specifically in your view controller, you could use it to handle motion-based events. You might also use it to coordinate the setting and changing of the first responder. For more information about how events are distributed and handled in iOS applications, see *Event Handling Guide for iOS*.

Accessing Related View Controller Objects

Although a view controller's main responsibility is to provide and manage its view hierarchy, view controllers often work in concert with other view controllers to present more sophisticated interfaces. For example, a navigation controller manages the navigation bar view, which provides a back button and information specific to the current view controller. For the specific content, the navigation controller relies on your custom view controllers to provide that content. Tab bar controllers similarly expect your custom view controllers to provide information related to tabs.

A custom view controller is responsible for providing any specific objects needed by higher-level view controllers. Table 2-3 lists the types of objects your view controller should be prepared to provide and the situations in which you need to provide it. If you do not specify a custom object for any of these properties, the view controller provides an appropriate default item for you.

Table 2-3 Supporting objects managed by your custom view controllers

Object	Property	Description
Navigation toolbar items	`toolbarItems`	Used in conjunction with a navigation controller that provides a custom toolbar. You can use this property to specify any items you want displayed in that toolbar. As the user transitions from one screen to the next, the toolbar items associated with the new view controller are animated into position. The default behavior is to provide no custom toolbar items. Available in iOS 3.0 and later. For information about configuring a navigation toolbar, see "Displaying a Navigation Toolbar" (page 81).
Navigation bar content	`navigationItem`	Used in conjunction with a navigation controller. The navigation item object provides the objects to be displayed in the navigation bar when your view controller is displayed. You can use this to provide a custom title or additional controls for your view. The default navigation item uses the title of your view controller as the title of the navigation bar and does not provide any custom buttons. The navigation controller adds the back button automatically. For information about specifying the contents of the navigation bar, see "Customizing the Navigation Bar Appearance" (page 75).
Tab bar item	`tabBarItem`	Used in conjunction with a tab bar controller. The tab bar item provides the image and text to display in the tab associated with your view controller. The default tab bar item contains the title of your view controller and no image. For information about specifying the tab contents for your view controller, see "Creating a Tab Bar Interface" (page 88).

For more information about these properties, see *UIViewController Class Reference*.

Navigation Controllers

You use navigation controllers to manage the presentation of hierarchical data in your application. A navigation controller manages a self-contained view hierarchy (known as a navigation interface) whose contents are composed partly of views managed directly by the navigation controller and partly of views managed by custom view controllers you provide. Each custom view controller manages a distinct view hierarchy but a navigation controller coordinates the navigation between different 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 provides an overview of how you configure and use navigation controllers in your application. For information about ways in which you can combine navigation controllers with other types of view controller objects, see "Combined View Controller Interfaces" (page 119).

Anatomy of a Navigation Interface

Although its primary job is to manage the presentation of your custom view controllers, a navigation controller is also responsible for presenting some custom views of its own. Specifically, it presents a navigation bar, which contains a back button along with some buttons you can customize. In iOS 3.0 and later, a navigation controller can also present a navigation toolbar view and populate it with custom buttons; display of the toolbar is optional though.

Figure 3-1 shows the key views of a navigation interface. The navigation view in this figure is the view stored in the navigation controller's `view` property. This is the view that you get and embed in a window or present using another view controller. All of the other views in the interface are part of what is essentially an opaque view hierarchy managed by the navigation controller.

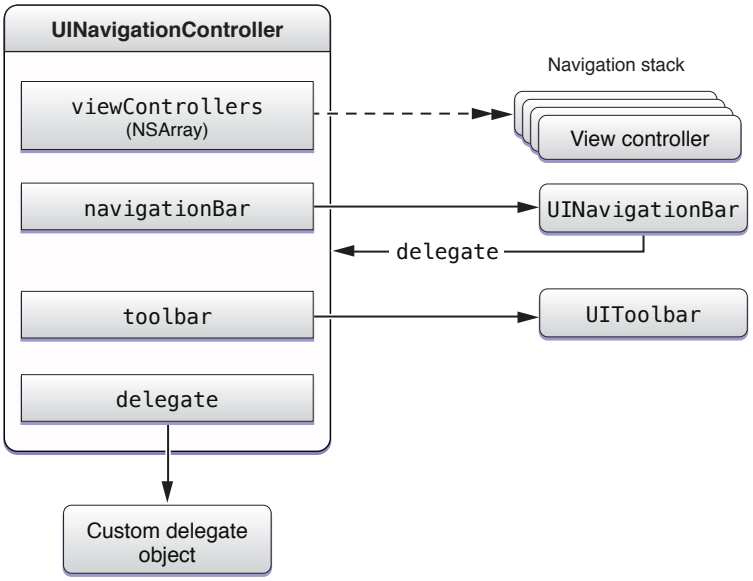
Figure 3-1 The views of a navigation interface

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 specific interfaces 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 75). For information about how to display and configure custom toolbar items in a navigation interface, see "Displaying a Navigation Toolbar" (page 81).

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 but the rest are created by the navigation controller itself. Specifically, you are responsible for providing the view controllers with the custom content you want to present. If you want to respond to notifications from the navigation controller, you can also provide a delegate object. However, the navigation controller creates the additional views (such as the navigation bar and toolbar) that are used for the navigation interface and is responsible for managing those views. Figure 3-2 shows the relationship between the navigation controller and these key objects.

Figure 3-2 Objects managed by the navigation controller

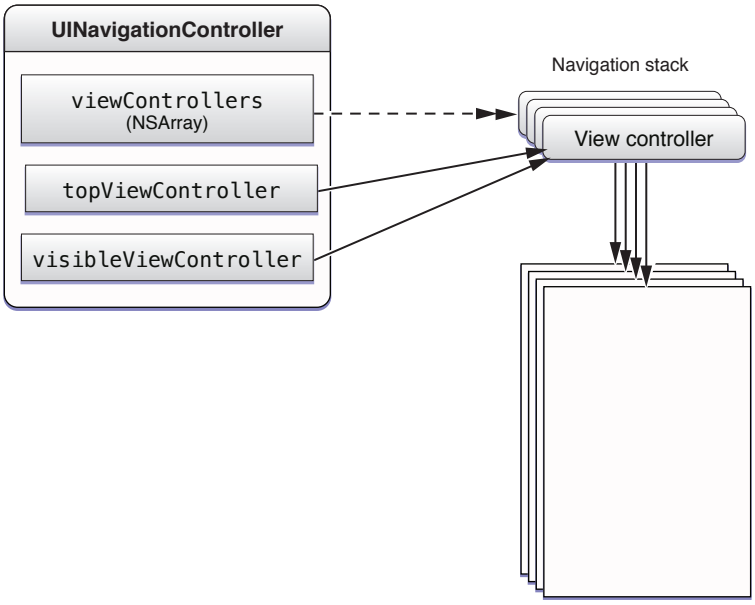


Except for modifying some aspects of their appearance, you should not modify the navigation bar or toolbar objects associated with a navigation controller. The navigation controller alone is responsible for configuring and displaying them. In addition, a navigation controller object automatically assigns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

Among the objects it is all right for you to modify are 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 can never be removed. Additional items can be added to the stack using the methods of the `UINavigationController` class.

Figure 3-3 shows the relevant relationships between the navigation controller and the objects on the navigation stack. It is important to note that the objects in the `topViewController` and `visibleViewController` properties need not be the same. If you present a view controller modally from the topmost object in the stack, the `topViewController` property does not change but the object in the `visibleViewController` property does. Specifically, the value in the `visibleViewController` property changes to reflect the modal view controller that was presented.

Figure 3-3 The navigation stack



Your main responsibility is to push new view controllers onto the stack in response to user actions. Each view controller you push on the navigation stack is responsible for presenting some portion of your application's data. Typically, when the user selects an item in the currently visible view, you create a new view controller object, assign the data for the selected item to it, and push the new view controller onto the stack. Doing so is how you present the selected data to the user. For example, when the user selects a photo album, the Photos application pushes a view controller that displays the photos in that album. 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 when tapped, pops the topmost view controller automatically.

For more information about how to customize the navigation bar, see "Customizing the Navigation Bar Appearance" (page 75). For information about pushing view controllers onto the navigation stack (and removing them later), see "Modifying the Navigation Stack" (page 73). For information on how to customize the contents of the toolbar, see "Displaying a Navigation Toolbar" (page 81).

Creating a Navigation Interface

You use a navigation interface in situations where the information you want to present is organized hierarchically. Typically, you use a navigation controller to manage the presentation of hierarchical data but you could also use one to manage multilevel editing or some other interface that required multiple successive screens.

Before creating a navigation interface, you need to decide how you intend to use it. There are a handful of places where you might install a navigation interface in your application:

- Install it directly in your application's main window.
- Install it as the root 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 or otherwise use it as a standalone view controller that you can display and dismiss as needed.
- 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 application exits. However, 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 modal and custom view controllers. The only difference is that the navigation controller continues to provide additional navigation features not available with a single custom 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.

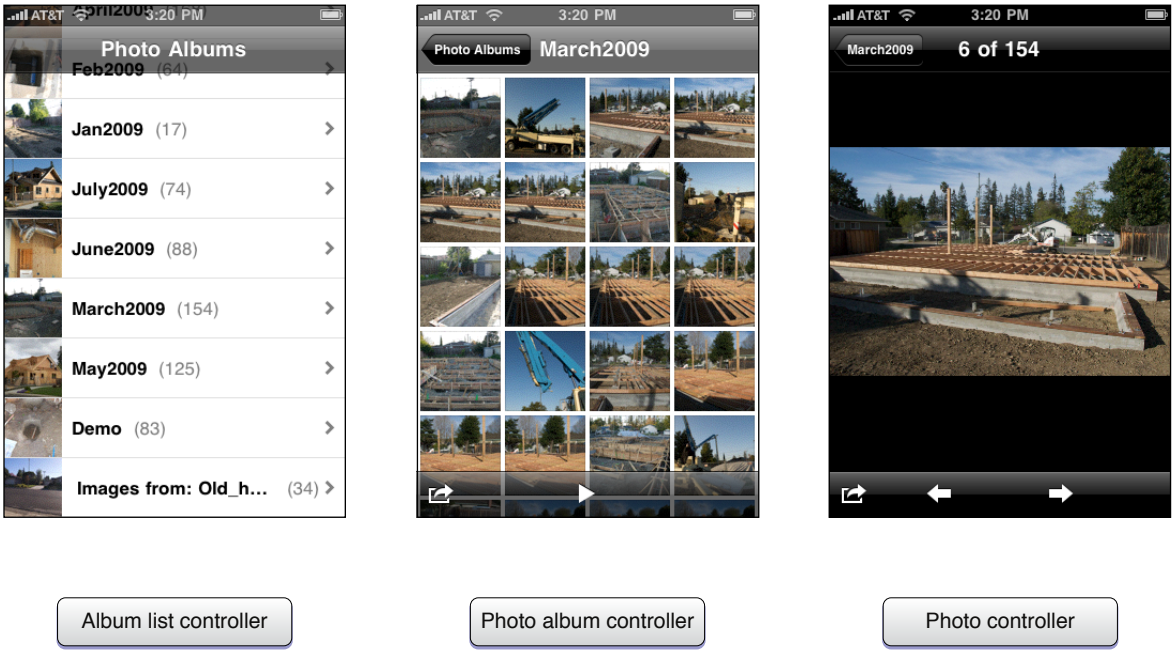
Note: For a specific example of how to present a navigation controller modally, see "Displaying a Navigation Controller Modally" (page 123).

Defining the Custom View Controllers for a Navigation Interface

One of the key things you must do to implement a navigation interface is decide what data you plan to present at each stage. Every navigation interface has at least one level of data that represents the root level of data. This is the starting point of your interface. For example, the Photos application 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.

For each level of your data hierarchy, you must provide a custom view controller object to manage and present the data at that level. If the presentation at multiple levels is essentially the same, you can reuse the same view controller class if you want. However, at runtime, you must still create separate instances of that class and configure each one to manage its own set of data. For example, the Photos application has three distinct presentation types, so it would need to be three separate view controller objects, such as the ones shown in Figure 3-4.

Figure 3-4 Defining view controllers for each level of data



With the exception of view controllers managing leaf data, each custom view controller must provide a way for the user to navigate to the next level of the data hierarchy. 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 application 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.

If your application has highly structured data, with a known progression from one level to the next, defining the view controllers at each level (and the relationships between them) should be relatively straightforward. (The Photos application always displays albums, followed by the contents of an album, followed by a single photo.) But if your application presents data differently depending on which item is selected at the current level, you might want to use additional information in your data model to determine how to present that data. You could use that information to determine whether to use different view controller classes or to customize the presentation style of a single view controller class. For example, viewing songs by Composer in the iPod application displays a list of albums or a list of songs depending on whether the songs for that composer come from the same album or from several different albums. In both cases though, the data is presented in list form using a table; therefore, you could use a single view controller but simply provide different table cells for each type of data.

For general information and guidance on defining custom view controllers, see "Custom View Controllers" (page 27).

Loading Your Navigation Interface from a Nib File

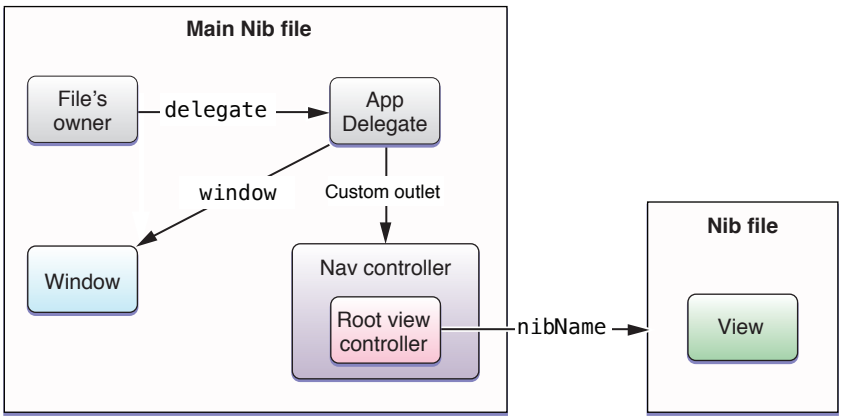
Although Interface Builder supports the inclusion of navigation controller objects in nib files, the semantics are slightly different than they are for custom view controllers. With a custom view controller, you use a nib file to store the views associated with the view controller, but you typically create the view controller separately—either programmatically or by loading it from a different nib file. However, with a navigation controller, the view is always created programmatically so there is no view to put in a separate nib file. One

consequence of this behavior is that a navigation controller never manages a nib file—in other words, you never assign a navigation controller to the File's Owner placeholder. Instead, the only time you mix navigation controllers and nib files is when the navigation controller itself is stored in the nib file. For this reason, a navigation controller is almost always a passenger in a nib file managed by some other object.

It makes the most sense to include navigation controllers in your application's main nib file. You do this when the navigation controller itself provides the main view for your application's window or in situations where the navigation controller provides the root view for a tab bar interface. Although you could also load standalone or modally presented navigation controllers from your main nib file (or any other nib file), doing so is not optimal. In those scenarios, it is usually easier to create the navigation controller programmatically at the point of use.

Figure 3-5 shows the configuration of the main nib file for an application that displays a navigation interface in its main window. The nib file includes the window object, the navigation controller, and the root view controller for the navigation interface, which is embedded inside the navigation controller itself. (The view managed by the root view controller typically would not need to be in the same nib file, and in this example is shown in a separate nib file.)

Figure 3-5 Nib file containing a navigation interface



Because the navigation controller creates its view programmatically, you cannot install that view in a window using Interface Builder. Instead, you must also add the view to a window programmatically. If the window and navigation controller themselves are in a nib file, then you must remember to store references to those objects so that you can access them later.

To configure a navigation controller in a nib file, you do the following:

1. Drag a navigation controller object from the library to your Interface Builder document window.

   When you add a navigation controller, Interface Builder also adds a navigation bar, an embedded root view controller object, and a navigation item for your root view controller. You can access some of these objects by selecting them in the navigation controller edit surface. You can also select any of the objects from the document window when in outline and browser modes.

2. Save a reference to the navigation controller using an outlet.

   In order to access the navigation controller at runtime, you either need to use an outlet or you must explicitly retrieve the nib file's top-level objects when you load the nib file. Using an outlet is generally much simpler. To add an outlet, add a variable to the declaration for your application delegate class that is similar to the following:

```
@interface MyAppDelegate : NSObject <UIApplicationDelegate> {
   IBOutlet UINavigationController*  myNavigationController;
}
@end
```

After adding the outlet definition, create a connection from that outlet to the navigation controller object in Interface Builder.

3. Set the class of the root view controller to a custom class from your Xcode project.

   The class you choose should be the one responsible for displaying the highest-level data in your navigation hierarchy. If there are multiple levels of data, the view presented by your view controller class should include controls for navigating to the next level of data. For information about how to design your view controllers, see "Defining the Custom View Controllers for a Navigation Interface" (page 67).

4. Configure the view for your root view controller.

   You can include the view in the same nib file or store it in a different nib file. Using a different nib file is recommended because it gives the system the option of removing the view from memory during low-memory conditions.

   To specify a separate nib file, set the NIB Name attribute of the root view controller to the name of your nib file. To include the views in the same nib file as the navigation controller, drag your view objects to the navigation controller edit surface.

5. In the `applicationDidFinishLaunching:` method of your application delegate, add the navigation controller's view to your main window.

   The navigation controller does not install itself automatically in your application's window. You must do this programmatically with code similar to the following:

   ```
   - (void)applicationDidFinishLaunching:(UIApplication *)application {
       [window addSubview:myNavigationController.view];
   }
   ```

6. Save your nib file.

In addition to configuring the root view controller object, you can also use the inspector to change the attributes of the navigation item and navigation bar that Interface Builder creates. For the navigation bar, you can change some basic stylistic attributes. For the navigation item, you can specify a custom title and prompt text. If you do not specify a custom string for the navigation item, Interface Builder uses the string in the custom view controller's title property.

The navigation controller's edit surface provides a convenient way to edit the configuration of the navigation bar. To add additional buttons or views to the navigation bar, drag the appropriate objects from the library to the navigation bar in the edit surface. Doing so adds the corresponding objects to the root view controller's navigation item. You can then configure the items you dragged as you would other Interface Builder objects. For example, you could connect buttons in the navigation item to action methods of your view controller in order to facilitate the handling of user taps at runtime.

For information about using Interface Builder to configure the contents of your nib files, including how to connect views and other controls to action methods, see *Interface Builder User Guide*.

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 application window, you could create the navigation controller in the `applicationDidFinishLaunching:` method of your application delegate.

There are other situations where it might make more sense to create a view controller programmatically. For example, if you plan to present a navigation interface modally, it is usually simpler to create the navigation controller object at the point of use, present it, and then release it when it is no longer needed. Loading such a view controller from a nib file would require extra overhead for loading the nib file and might require you to store a pointer to the navigation controller object for longer than necessary.

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. Add the navigation controller's view to your window (or otherwise present it in your interface).

After you create the navigation controller, you can use it as appropriate. For example, you could add its view to your window, use it to initialize another view controller, or present it modally.

Listing 3-1 shows a simple implementation of the `applicationDidFinishLaunching:` method that creates a navigation controller and adds it to the application'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 3-1 Creating a navigation controller programmatically

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

    window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    [window addSubview:navigationController.view];
    [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 3-1 (page 64)). 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 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 application'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 hope to 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.

Note: 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 still underlap the navigation bar but will not underlap the tab 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. Set the `translucent` property of your navigation controller to `YES`. This allows your content to underlap the navigation bar.

3. To underlap the status bar, set the `wantsFullScreenLayout` property of your view controller to `YES`. (The navigation bar must be translucent in order for this attribute to be recognized.)

4. To underlap an optional toolbar, set the `translucent` property of the toolbar 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 application 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 "Adopting a Full-Screen Layout for Custom Views" (page 56).

Modifying the Navigation Stack

Although the navigation controller manages the navigation stack, you are responsible for creating the objects that reside on that stack. When initializing a navigation controller object, you must provide a custom view controller to display the root content of your data hierarchy. This root view controller always sits at the bottom of the navigation stack and can never be removed. After that, you can add or remove other view controllers either programmatically or in response to user interactions. To add and remove these view controllers, you use the methods of the `UINavigationController` class.

The navigation controller provides several options for managing the contents of the navigation stack. These options cover the various scenarios you are likely to encounter in your application. Table 3-1 lists these scenarios and how you respond to them.

Table 3-1 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 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.

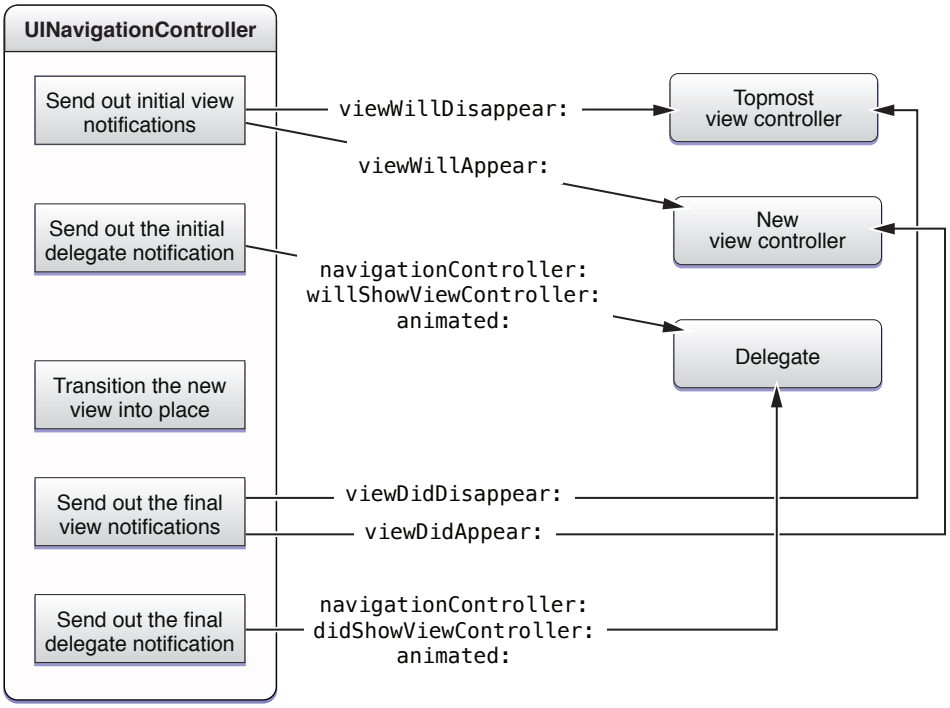
Scenario	Description
Restore the navigation stack to a previous state.	When your application launches, you can use the `setViewControllers:animated:` method to restore the navigation controller to a previous state. For example, you would use this method to return the user to the same screen they were viewing when they last quit the application. In order to restore your application to a previous state, you must first save enough state information to recreate the needed view controllers. When the user quits your application, 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.
Return the user to the root view controller.	To return to the top of your navigation interface, use the `popToRootViewControllerAnimated:` 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.
Jump to an arbitrary location in your data hierarchy.	You can use the `setViewControllers: animated:` to jump seamlessly to anywhere in your data hierarchy. Although jumping around your data hierarchy is not a great idea (because of the potential confusion it might cause), using this method to provide a trail back to the root view controller is the best way to do it.

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

As changes occur to the navigation stack, the navigation controller sends appropriate messages to its delegate. Specifically, whenever you push or pop a view controller, the navigation controller sends messages to the affected view controllers. Figure 3-6 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 3-6 Messages sent during stack changes



You can use the methods of the navigation controller's delegate to update the state of your navigation interface and modify your application's data model. For example, if you are using a navigation interface to facilitate the editing of content, you might use the `navigationController:willShowViewController:animated:` method to save any changes before the corresponding view controller is dismissed. However, you would not use these methods to make changes to your views or view hierarchy. Those types of changes should be handled by your custom view controllers.

Customizing the Navigation Bar Appearance

A navigation bar is a container view that manages the controls commonly found in a navigation interface. Although it is essentially just a view object, navigation bars take 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 custom view controllers) to help in this process.

Because managing the navigation bar is the responsibility of the navigation controller, direct modification of the navigation bar itself is considered off limits for the most part. Be that as it may, there are still many ways in which you can customize the navigation bar to suit your needs. The following sections explain the structure of navigation bars and how you customize them for your application.

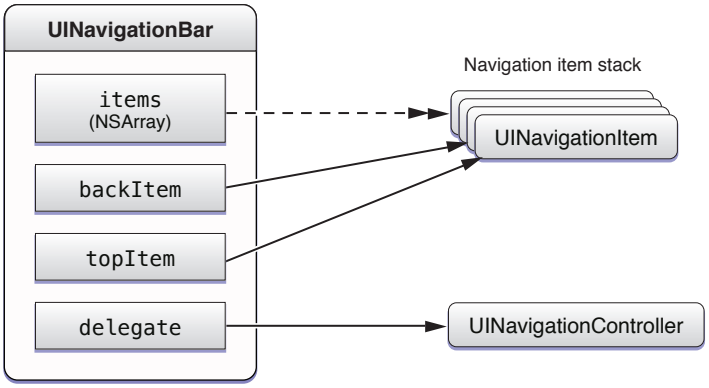
Note: Although the following sections focus on the use of navigation bars with a navigation controller, you can also create navigation bars as standalone views and use them however you wish. For more information about using the methods and properties of the `UINavigationBar` class, see *UINavigationBar 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. Unlike a navigation controller, a navigation bar is an actual view object that can be embedded inside other views.

Figure 3-7 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 3-7 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 results in the throwing of an exception.

When used in a navigation interface, the contents of the navigation bar's stack always parallel the contents of the parent navigation controller's stack. In other words, for each view controller in the navigation stack, there is a corresponding navigation item in the same position on the navigation item stack of the navigation bar. The reason for this one-to-one correspondence is that each view controller actually provides its own navigation item.

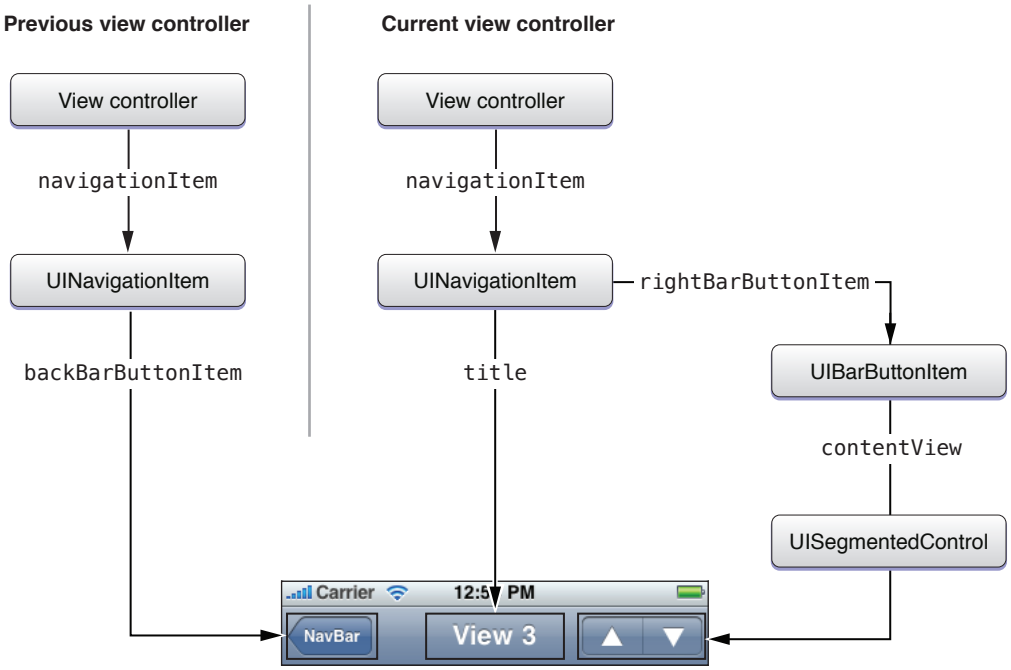
A navigation bar has three primary positions for placing items: left, right, and center. Table 3-2 lists the properties of the `UINavigationItem` class that are used to configure each of these positions. When configuring a navigation item for use with a navigation controller, be aware that custom controls 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 3-2 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.
Center	`titleView`	In a navigation interface, the navigation controller displays a custom view with the title of your 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 as well by wrapping the view in a `UIBarButtonItem` object.

Figure 3-8 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 3-8 Navigation bar structure



Showing and Hiding the Navigation Bar

When 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 application 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 it is permissible to modify, including the following:

- `barStyle` property
- `translucent` property

- `tintColor` property

Figure 3-9 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 3-9 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 78).

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. Of course, 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.

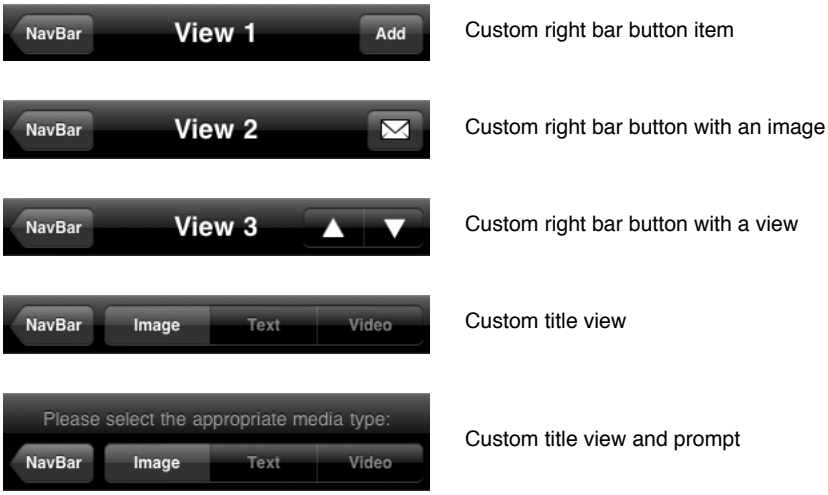
For the topmost view controller, the item that is displayed on the right side of the navigation bar is determined using the following rules:

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

Figure 3-10 shows some different navigation bar configurations, including several that use custom views and prompts. The navigation bars in this figure are from the sample project *NavBar*.

Figure 3-10 Custom buttons in the navigation bar



Listing 3-2 shows the code from the NavBar application that would be required to create the third navigation bar in Figure 3-10, 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 3-2 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 retain];    // keep track of this
 for later

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

self.navigationItem.rightBarButtonItem = segmentBarItem;
[segmentBarItem release];
```

Configuring your view controller's navigation item programmatically is the most common approach for most applications. 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 "Enabling Edit Mode for a View" (page 57).

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.

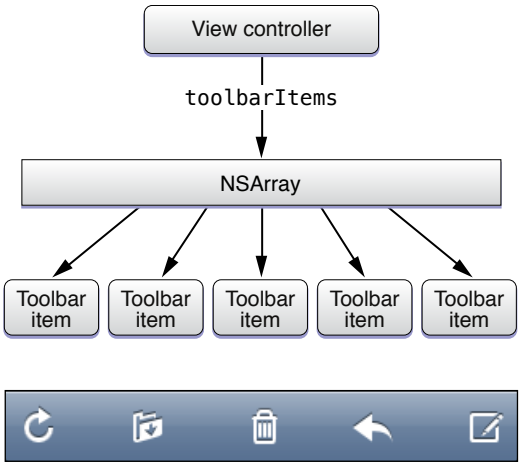
To configure a toolbar for your navigation interface, you must do the following:

- 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 custom view controllers, as described in "Specifying the Toolbar Items" (page 82).

  If you do not want to show a toolbar for a particular view controller, you can hide the toolbar as described in "Showing and Hiding the Toolbar" (page 83).

Figure 3-11 shows an example of how the objects you associate with your custom 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 application.

Figure 3-11 Toolbar items in a navigation interface



Specifying the Toolbar Items

When configuring bar button items, always 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 3-12 shows a sample toolbar that places a segmented control in the center of the toolbar and Listing 3-3 shows the code needed to configure such a toolbar. You would implement the method in your view controller and call it at initialization time.

Figure 3-12 A segmented control centered in a toolbar

Listing 3-3 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];
   [sortToggle release];

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

   [sortToggleButtonItem release];
   [flexibleSpaceButtonItem release];
}
```

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 application 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 controllers to organize your application into one or more distinct modes of operation. A tab bar controller manages a self-contained view hierarchy (known as a tab bar interface) whose contents are composed partly of views managed by the tab bar controller and partly of views managed by custom view controllers you provide.

This chapter provides an overview of how you configure and use tab bar controllers in your application. 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 119).

The 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 application 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 application'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 custom view controllers that provide the content view for each mode. Each custom view controller is designated as the **root 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 root view controller initially.

Figure 4-1 shows the tab bar interface implemented by the Clock application. 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 root view controller of the selected tab.

Figure 4-1 The views of a tab bar interface

Although a tab bar view is normally a customizable object, when it 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 94).

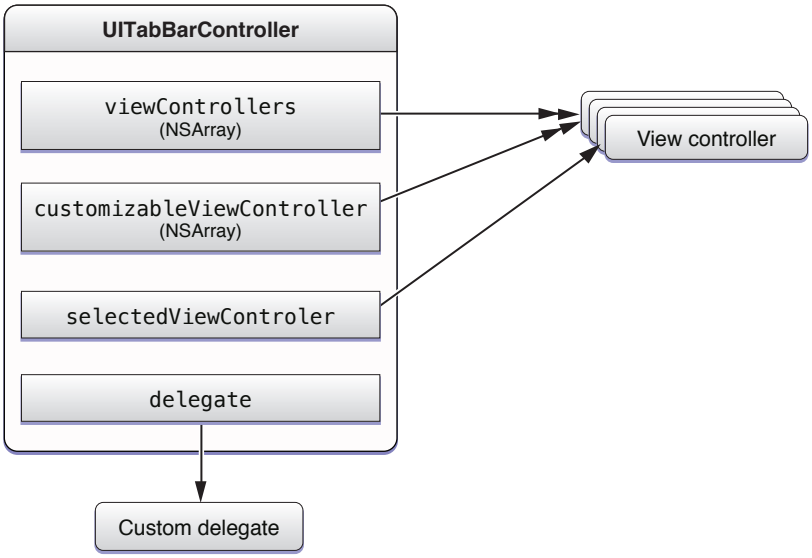
The Objects of a Tab Bar Interface

A standard tab bar interface consists of the following objects:

- A `UITabBarController` object
- One custom view controller object for each tab
- An optional delegate object

Figure 4-2 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 root view controller for a corresponding tab in the tab bar.

Figure 4-2 A tab bar controller and its associated view controllers

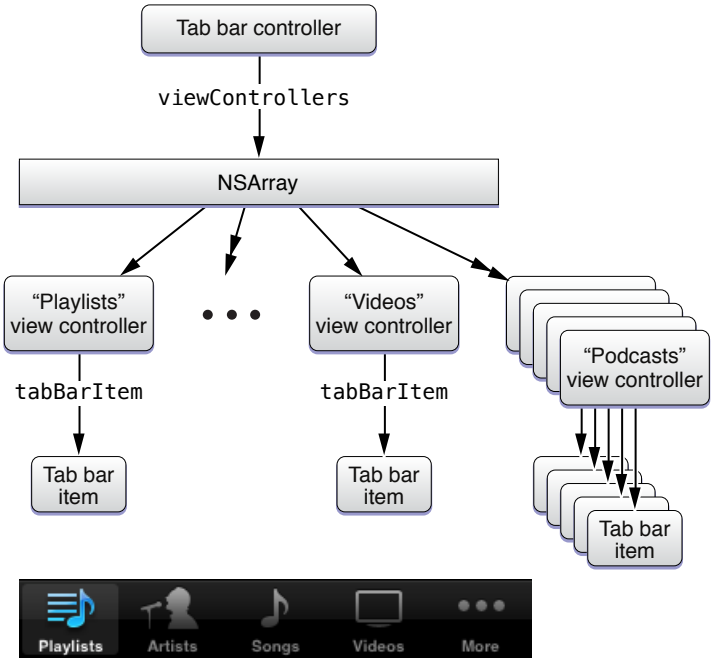


The custom view controllers you provide are the most important elements of the tab bar interface. Each view controller defines the content that is displayed when the corresponding tab is selected. You can use custom view controllers that display a single view or you can use a navigation controller to allow for more complex navigation within the tab. You would not, however, install another tab bar controller in the tab.

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. For the most part, 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`.

Although the tab bar view is a key part of your tab bar interface, you do not modify that view directly. The tab bar controller object assembles the contents of the tab bar from the `UITabBarItem` objects provided by your custom view controllers. Figure 4-3 shows the relationship between the tab bar controller, view controllers, and the tab bar item objects in the iPod application. 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 4-3 Tab bar items of the iPod application



Because tab bar items are used to configure the tab bar, you must configure the tab bar item of each root view controller prior to 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 Nib File" (page 90). If you are creating your tab bar interface programmatically, you must create a new `UITabBarItem` object for each of your view controllers as described in "Creating a Tab Bar Interface Programmatically" (page 93).

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 94).

Creating a Tab Bar Interface

Before creating a tab bar interface, you need to decide how you intend to use it. Because it imposes an overarching organization on your data, there are only a handful of appropriate ways to use a tab bar interface:

- Install it directly in your application's main window.
- Install it as one of the two root views in a split view interface. (iPad only)
- Present it modally to display some data that requires its own mode-based organization.
- Display it from a popover. (iPad only)

Installing a tab bar interface in your application'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 application's data, with each tab leading the user to a distinct part of the application. Because it provides access to your entire application, it must be the root part of your window.

Note: Although a navigation controller can be embedded inside a tab, the reverse is not true. Presenting a tab bar interface from within a navigation interface is potentially confusing for users. A navigation interface uses one or more custom view controllers to present an interface focused on one goal, which is usually the management of a specific type of data. By contrast, the tabs of a tab bar interface can reflect completely different purposes in an application and need not be related in any way. In addition, pushing a tab bar controller on a navigation stack would cause the tabs to be displayed for that screen only and not for any others.

Of course, it is still 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. Using a tab bar in this way should be avoided though if a simpler design approach is available.

Defining the Custom View Controllers for a Tab Bar Interface

Because each mode of a tab bar interface is separate from all other modes, the root view controller in each tab essentially 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 might install a navigation controller to manage the navigation through that data. If the data being presented is simpler, you could install a custom view controller with a single view.

Figure 4-4 shows several screens from the Clock application. 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 4-4 Tabs of the Clock application

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

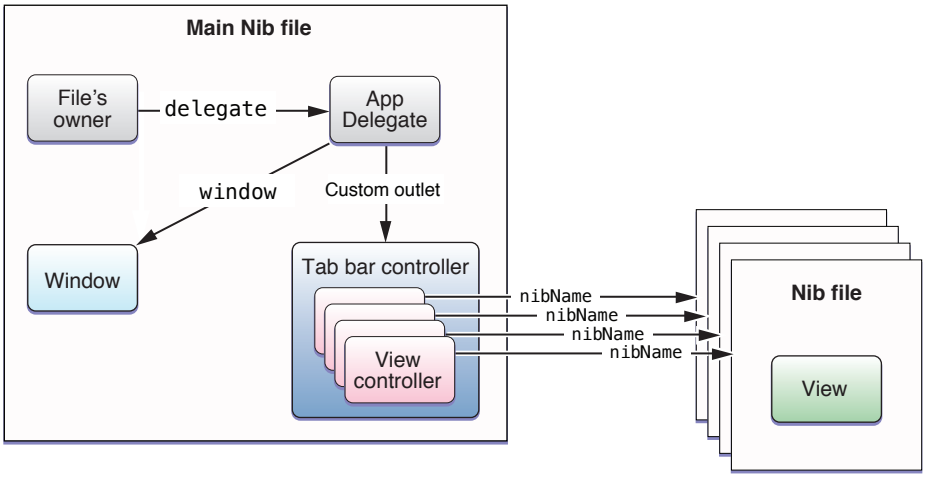
Creating a Tab Bar Interface Using a Nib File

The semantics for loading a tab bar controller from a nib file are slightly different from those for custom view controllers. With a custom view controller, you use a nib file to store the views associated with the view controller, but you typically create the view controller separately—either programmatically or by loading it from a different nib file. However, a tab bar controller always creates its view programmatically and so there is no view to put in a separate nib file. One consequence of this behavior is that a tab bar controller never manages a nib file—in other words, you never assign a tab bar controller to the File's Owner placeholder. Instead, the only time you mix tab bar controllers and nib files is when the tab bar controller itself is stored in the nib file. For this reason, a tab bar controller is almost always a passenger in a nib file managed by some other object.

It makes the most sense to include tab bar controllers in your application's main nib file. You do this when the tab bar controller itself provides the main view for your application's window. Although you could also load a modally presented tab bar controller from your main nib file (or any other nib file), doing so is not optimal. In fact, it is usually easier to create the tab bar controller programmatically at the point of use.

Figure 4-5 shows the typical configuration of a tab bar controller and its related objects in an application's main nib file. Each custom view controller represents the root view controller associated with a single tab. Because they are all custom view controllers (and not navigation controllers), each view controller has a reference to a separate nib file containing its view. Although you could include the custom views in the main nib file, doing so is not recommended. Storing the views in separate nib files gives the system the option of purging them from memory as needed.

Figure 4-5 Nib file containing a tab bar interface



If you are creating your Xcode project from scratch, you can use the Tab Bar Application template to create your project. The main nib file that comes with this project includes a tab bar controller object that is minimally configured for the project.

The following steps show you how to add a tab bar controller to your application's main nib file from scratch. If you are starting with the Tab Bar Application template, you can simply skip the first two steps.

1. Drag a tab bar controller object from the library to your Interface Builder document window.

   When you add a tab bar controller to a nib file, Interface Builder also adds a tab bar view, two root view controllers, and two tab bar items (one for each view controller). You can access some of these objects by selecting them in the tab bar controller edit surface. You can also select any of the objects from the document window when in outline or browser modes.

   If you created your project using the Tab Bar Application template, this step is already done for you.

2. Save a reference to the tab bar controller using an outlet.

   In order to access the tab bar controller at runtime, you either need to use an outlet or you must explicitly retrieve the nib file's top-level objects when you load the nib file. Using an outlet is generally much simpler. To add an outlet, add a variable to the declaration for your application delegate class that is similar to the following:

   ```
   @interface MyAppDelegate : NSObject <UIApplicationDelegate> {
      IBOutlet UITabBarController*  myTabBarController;
   }
   @end
   ```

   After adding the outlet definition, create a connection from that outlet to the tab bar controller object.

   If you created your project using the Tab Bar Application template, this step is already done for you.

3. Add or delete view controllers to reflect the desired number of tabs for your interface.

   The number of view controllers embedded inside your tab bar controller object determines the number of tabs displayed by your tab bar interface. Your final tab bar controller should have at least two view controllers; otherwise, the need for a tab bar controller becomes questionable. From the Interface Builder

library, you can drag a View Controller object (`UIViewController`), Navigation Controller object (`UINavigationController`), or Table View Controller object (`UITableViewController`) and associate it with a tab.

To add a view controller, do one of the following:

- Drag the appropriate object from the library to the tab bar in the edit surface.
- Drag the object from the library to the tab bar controller in your Interface Builder document window. The window must be in outline mode.

When adding navigation or table view controllers to your interface, you should either drag the appropriate object from the library or select the tab bar controller and configure your view controller types using the Attributes inspector. Both of these options add the correct type of view controller object to your nib file. You should never add navigation or table view controllers by dragging a generic View Controller object to the nib file and change its class name to the desired class type.

To delete a view controller, select the view controller object in the edit surface or document window and press the Delete key.

4. Arrange the view controllers in the order you want them to appear onscreen.

   You can rearrange view controllers (and the corresponding tabs) by dragging the tabs displayed on the tab bar controller edit surface or by dragging the view controllers in the Interface Builder document window (outline mode only). Although the edit surface shows all of the tabs, only five are displayed at runtime. If your tab bar controller contains six or more view controllers, only the first four are displayed in the tab bar initially. The last spot on the tab bar is reserved for the More view controller, which presents the remaining view controllers.

5. Configure the root view controller for each tab.

   For each root view controller, you should configure the following attributes:

   - Set the class of any custom view controller objects using the Identity inspector. If the root view controller is a generic View Controller object or a Table View Controller object, you can change the class name to your custom subclass. If it is a Navigation Controller object, do not change the class name.
   - Provide a view for the view controller. The preferred way to do this is to configure the NIB Name attribute of the view controller with the name of the nib file containing the view. Although you can include the view in your main nib file if you want, doing so is not recommended.
   - As appropriate, configure any style or appearance information for the view controller.

   If you are using a navigation controller for one of the root view controllers, configure it as described in "Loading Your Navigation Interface from a Nib File" (page 68). For additional information and examples of how to embed a navigation controller in a tab bar controller, see "Adding a Navigation Controller to a Tab Bar Interface" (page 119).

6. Configure the tab bar item for each view controller.

You can select the tab bar item from the tab bar controller edit surface or from the Interface Builder document window when it is in outline or browser mode. Using Interface Builder, you can specify the title, image, and badge of a tab bar item. Alternatively, you can set the tab bar item to one of the standard system tabs by assigning a value to the Identifier property in the Attributes inspector.

7. In the `applicationDidFinishLaunching:` method of your application delegate, add the tab bar controller's view to your main window.

   The tab bar controller does not install itself automatically in your application's window. You must do this programmatically with code similar to the following:

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
    [window addSubview:myTabBarController.view];
}
```

8. Save your nib file.

You should not need to configure the tab bar view that is added to your nib file with the tab bar controller. This bar does not contain any style options and all other options are managed for you by the tab bar controller object.

For additional information about using Interface Builder to configure nib files, see *Interface Builder User Guide*.

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 application'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 custom root view controller object for each tab.
3. Add the root view controllers to an array and assign that array to your tab bar controller's `viewControllers` property.
4. Add the tab bar controller's view to your application's main window.

Listing 4-1 shows the basic code needed to create and install a tab bar controller interface in the main window of your application. 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 would need to replace the custom view controller names `MyViewController` and `MyOtherViewController` with classes from your own application.

Listing 4-1 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;

    // Add the tab bar controller's current view as a subview of the window
    [window addSubview:tabBarController.view];
}
```

Creating a Tab Bar Item Programmatically

For each root 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 ideal 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 could just as easily create and initialize the view controller object, create the tab bar item, and then make the association.

Listing 4-2 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's bundle) and a custom title string.

Listing 4-2 Creating the view controller's tab bar item

```
- (id)init {
    if (self = [super initWithNibName:@"MyViewController" bundle:nil]) {
        self.title = @"My View Controller";

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

If you are loading your tab bar interface from a nib file, you can also create your tab bar items using Interface Builder. For more information about including tab bar controllers (and tab bar items) in a nib file, see "Creating a Tab Bar Interface Using a Nib File" (page 90).

Managing Tabs at Runtime

After creating your tab bar interface, there are several ways to modify it and respond to changes in your application. 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. The following sections show you how to take advantage of these features in your application.

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 4-3 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 root 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 4-3 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 application 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 4-4 shows an example of what such a method would look like. The `hasValidLogin` method is a custom method that you would implement to validate the provided information.

Listing 4-4 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.

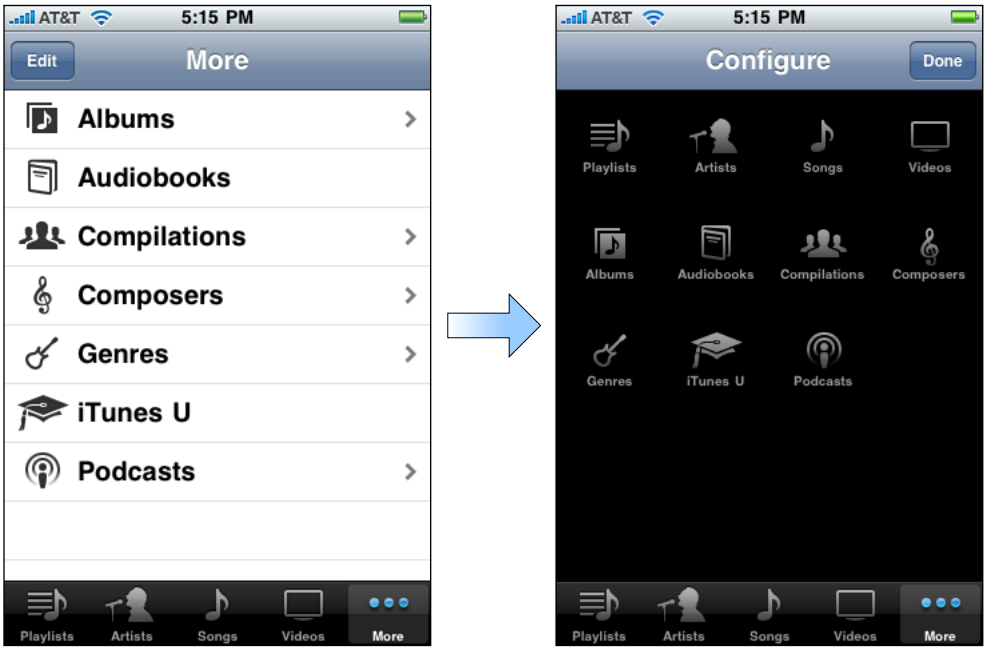
Both types of changes are reported to the tab bar controller's delegate, which is an object that conforms to the `UITabBarControllerDelegate` protocol. You might provide a delegate to keep track of user changes and update your application's state information accordingly. However, you should not use these 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 custom 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 applications 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 4-6 shows the More selection screen displayed by the iPod application. 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 4-6 Configuring the tab bar of the iPod application



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.

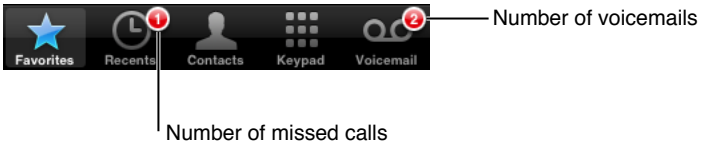
Important: Adding or removing view controllers in your tab bar interface also resets the array of customizable view controllers to the default value, allowing all view controllers to be customized again. Therefore, if you make modifications to the `viewControllers` property (either directly or by calling 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. However, if you want to call attention to a specific tab, perhaps because there is new content on that tab for the user to look at, you can do so with 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 4-7 shows badges for tabs in the Phone application.

Figure 4-7 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. For example, a view controller that displays the number of new items in its badge might use code similar to the following to create the badge value. (Note that the `numberOfNewItems` property in the example is a fictitious property implemented by the view controller to track the number of new items. To implement the example in your own code, you would replace references to that property with an appropriate value from your own view controller.)

```
if (self.numberOfNewItems == 0)
   self.tabBarItem.badgeValue = nil;
else
   self.tabBarItem.badgeValue = [NSString stringWithFormat:@"%d",
self.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 (such as the fictitious `numberOfNewItems` property in the preceding example), you can use 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 root 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 custom 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 "Adopting a Full-Screen Layout for Custom Views" (page 56).

iPad-Specific Controllers

Because the iPad has a different form factor than iPhone and iPod touch devices, it also has a couple of special controller objects for presenting content in that form factor. Developers of iPad applications are encouraged to use these controllers whenever possible. If you are developing a universal application, though, be sure not to create and use these controllers when your application is running on an iPhone or iPod touch.

Popovers

Although not actually 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 application's window. Popovers provide a lightweight way to present or gather information from the user and are commonly used in the following situations:

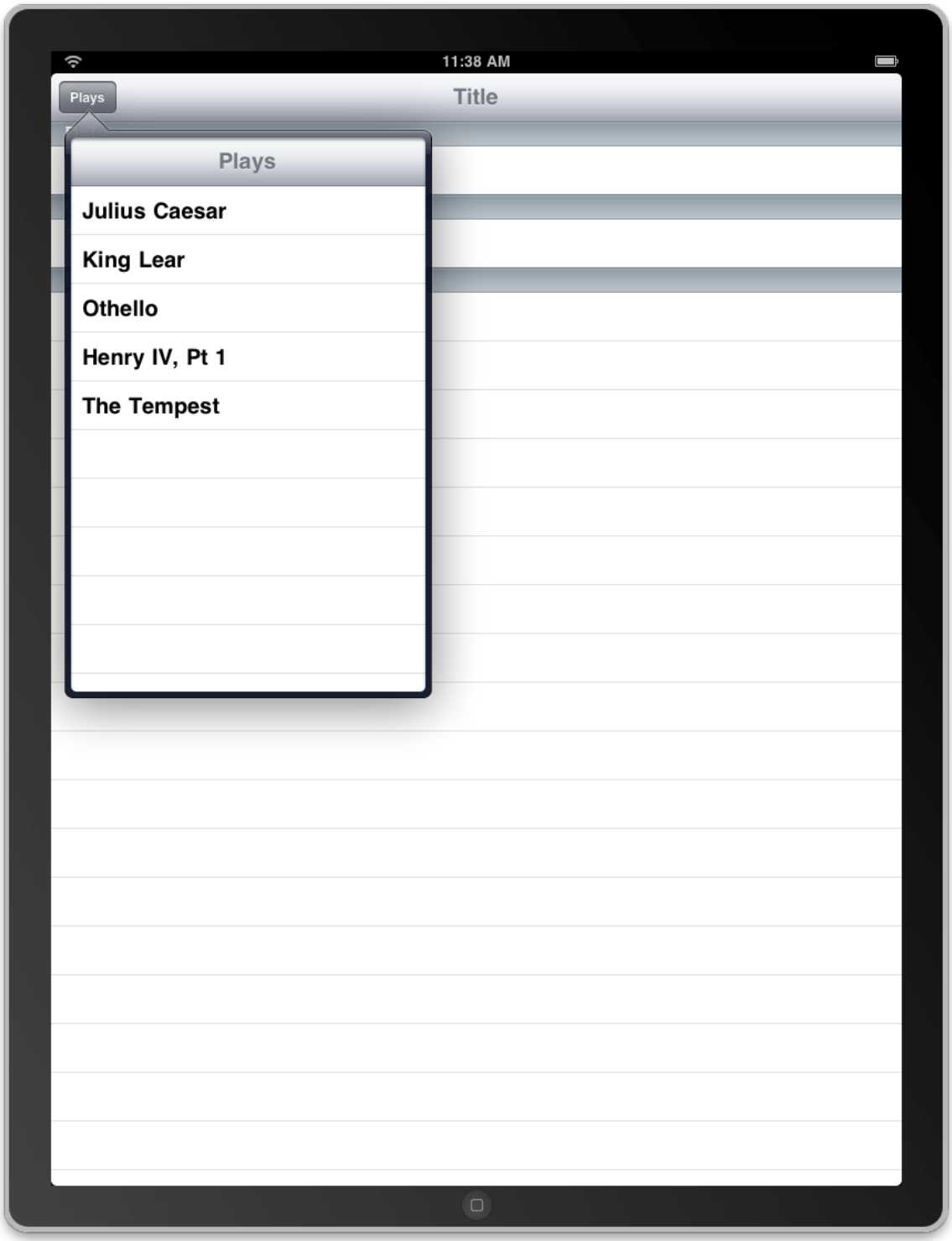
- To display information about an object on the screen.
- To manage frequently accessed tools or configuration options
- To present a list of actions to perform on objects inside one of your views
- To present one pane from a split view controller when the device is in a portrait orientation

The use of a popover for the preceding actions is less intrusive and cumbersome than a modal view. In iPad applications, 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 application. 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 application, 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 application's main view to display information about that play. (For more information about creating a split view interface, see "Split View Controller" (page 103).)

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, including custom view controllers, table view controllers, navigation controllers, and even tab bar 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 "container view controller" (page 129).
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 application'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

```
- (IBAction)toolbarItemTapped:(id)sender
{
   MyCustomViewController* content = [[MyCustomViewController alloc] init];
```

```
    UIPopoverController* aPopover = [[UIPopoverController alloc]
         initWithContentViewController:content];
    aPopover.delegate = self;
    [content release];

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

    [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 application. Rather than returning `NO` from this method, though, it is better to avoid designs that require keeping the popover alive. 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 application's state.

Tips for Managing Popovers in Your Application

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

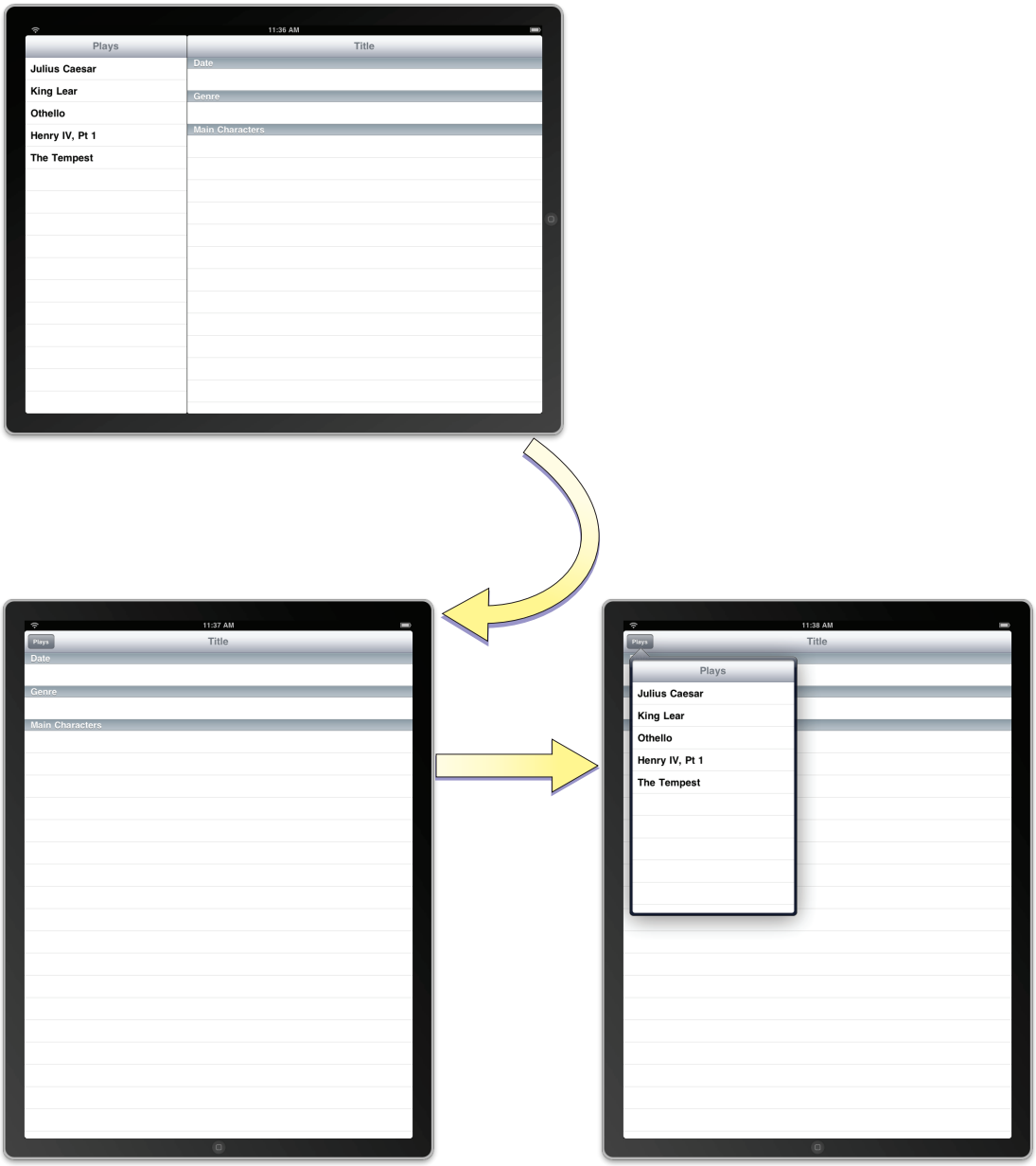
- Dismissing a popover programmatically requires a pointer to the popover controller. The only way to get such a pointer is to store it yourself, typically in the content view controller. This ensures that the content view controller is able to dismiss the popover in response to appropriate user actions.

- Popover controllers can be reused, so cache popover controllers 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.

Split View Controller

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. 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 shows only the second, larger pane and provides a toolbar button for displaying the first pane using a popover, as shown in Figure 5-2

Figure 5-2 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.

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 custom 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, as described in "More view controller" (page 129).

Adding a Split View Controller in Interface Builder

If you do not want to start with the Split View-based Application template project, you can still add a split view controller to your user interface. The library in Interface Builder includes a split view controller object that you can add to your existing nib files. When adding a split view controller, you typically add it to your application's main nib file. This is because the split view is usually inserted as the top-level view of your application's window and therefore needs to be loaded at launch time.

To add a split view controller to your application's main nib file:

1. Open your application's main nib file.
2. Drag a split view controller object to the nib file window.

   The split view controller object includes generic view controllers for the two panes.
3. Add an outlet for the split view controller in your application delegate object and connect that outlet to the split view controller object.
4. In the `application:didFinishLaunchingWithOptions:` method of your application delegate, install the split view controller's view as the main view of the window:

   ```
   [window addSubview:mySplitViewController.view];
   ```
5. For each of the split view controller's contained view controllers:
   - Use the Identity inspector to set the class name of the view controller.
   - In the Attributes inspector, set the name of the nib file containing the view controller's view.

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. Setting the class and nib names is all you have to do in your application's main nib file. The rest of the configuration is dependent on the type of view controller. For example, for 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 5-2 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 custom 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 5-2 Creating a split view controller programmatically

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
   MyFirstViewController* firstVC = [[[MyFirstViewController alloc]
                     initWithNibName:@"FirstNib" bundle:nil] autorelease];
   MySecondViewController* secondVC = [[[MySecondViewController alloc]
                     initWithNibName:@"SecondNib" bundle:nil] autorelease];

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

   [window addSubview:splitVC.view];
   [window makeKeyAndVisible];

   return YES;
}
```

Supporting Orientation Changes in a Split View

A split view controller relies on its two view controllers to determine whether interface orientation changes should be made. If one or both of the view controllers do not support the new orientation, no change is made. This is true even in portrait mode, where the first view controller is not displayed. Therefore, you must override the `shouldAutorotateToInterfaceOrientation:` method for both view controllers and return `YES` for all supported orientations.

When an orientation change occurs, the split view controller automatically handles most of the rotation behaviors. Specifically, the split view controller automatically hides the first view controller in its `viewControllers` array when rotating to a portrait orientation and shows it when rotating to a landscape orientation.

If you want to display the first view controller when in portrait orientations, you do so using a delegate object. When rotating to a portrait orientation, the split-view controller provides its delegate with a button that, when tapped, shows the first pane in a popover. All your application has to do is add this button to your application's toolbar in the delegate's `splitViewController:willHideViewController:withBarButtonItem:forPopoverController:` method and remove the button in the `splitViewController:willShowViewController:invalidatingBarButtonItem:` method. Listing 5-3 shows the implementations of these methods provided by the Split-View based application template. These methods are implemented by the detail view controller, which manages the contents of the split-view controller's second pane.

Listing 5-3 Adding and removing the toolbar button in response to split view orientation changes

```
// Called when rotating to a portrait orientation.
- (void)splitViewController: (UISplitViewController*)svc
willHideViewController:(UIViewController *)aViewController
```

```
withBarButtonItem:(UIBarButtonItem*)barButtonItem forPopoverController:
(UIPopoverController*)pc
{
    barButtonItem.title = @"Root List";
    NSMutableArray *items = [[toolbar items] mutableCopy];
    [items insertObject:barButtonItem atIndex:0];
    [toolbar setItems:items animated:YES];
    [items release];
    self.popoverController = pc;
}

// Called when the view is shown again in the split view, invalidating the button
 and popover controller.
- (void)splitViewController: (UISplitViewController*)svc
willShowViewController:(UIViewController *)aViewController
invalidatingBarButtonItem:(UIBarButtonItem *)barButtonItem {

    NSMutableArray *items = [[toolbar items] mutableCopy];
    [items removeObjectAtIndex:0];
    [toolbar setItems:items animated:YES];
    [items release];
    self.popoverController = nil;
}
```

Modal View Controllers

Modal view controllers provide interesting ways to manage the flow of your application. Most commonly, applications use modal view controllers as a temporary interruption in order to obtain key information from the user. However, you can also use modally presented view controllers to implement alternate interfaces for your application at specific times.

This chapter describes the uses for modal views in your application and shows you when and how to present them.

About Modal View Controllers

Modal view controllers are a tool that you have at your disposal for interrupting the current workflow and displaying a new set of views. A modal view controller is not a specific subclass of `UIViewController`, like `UITabBarController` or `UINavigationController` are. Instead, any view controller can be presented modally by your application. However, like tab bar and navigation controllers, you present view controllers modally when you want to convey a specific meaning about the relationship between the previous view hierarchy and the newly presented view hierarchy.

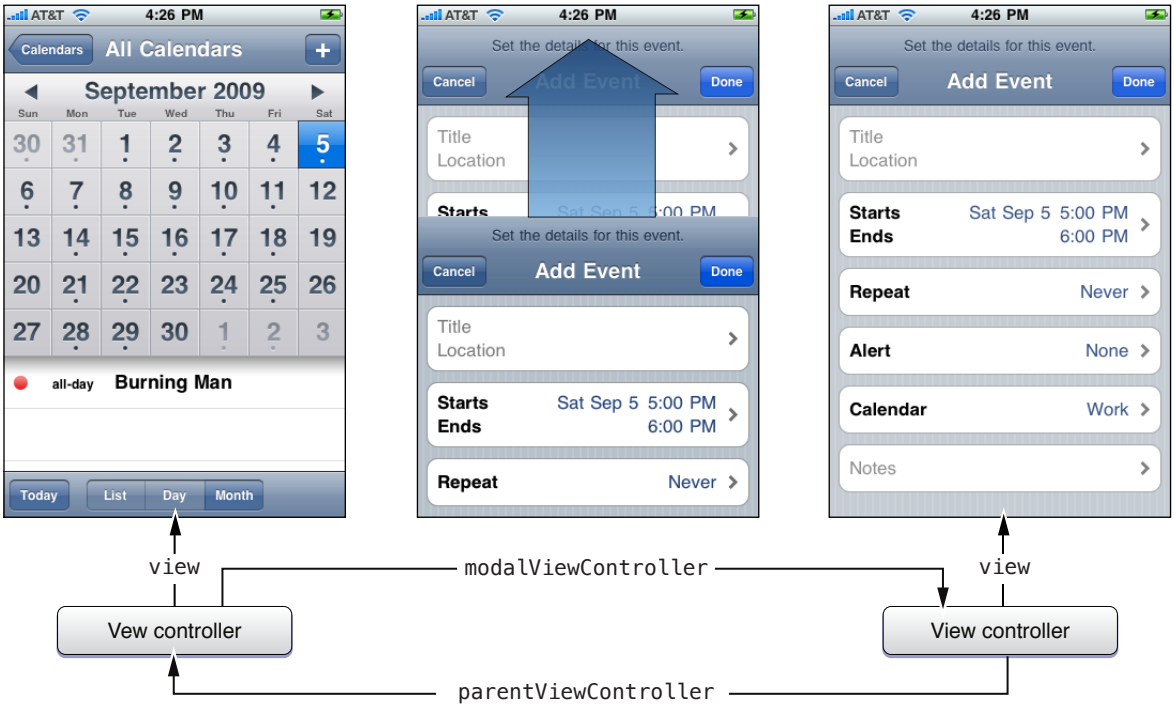
There are several reasons to use modal view controllers in your application:

- Use them to gather information from the user immediately.
- Use them to present some content temporarily.
- Use them to change work modes temporarily.
- Use them to implement alternate interfaces for different device orientations.
- Use them to present a new view hierarchy with a specific type of animated transition (or no transition).

Most of these reasons involve interrupting your application's workflow temporarily in order to gather or display some information. Once you have the information you need (or have presented the user with the appropriate information), you dismiss the modal view controller to return the application to its previous state. Even the last option (implementing an alternate interface) should be treated as a temporary interruption.

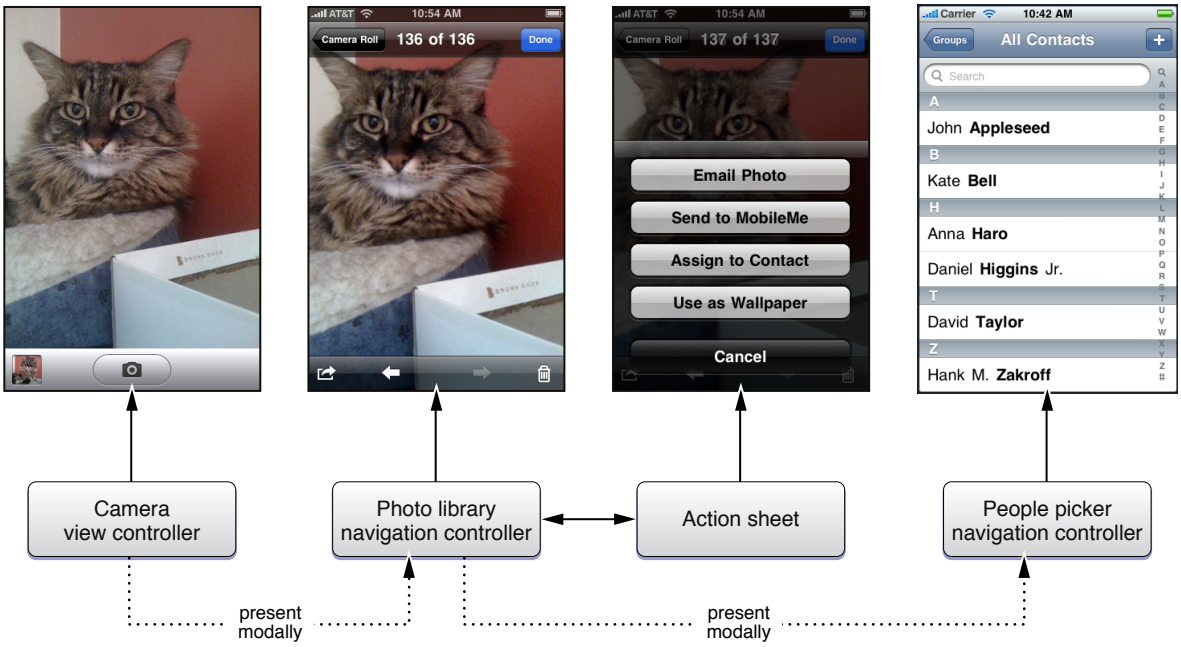
When you present a modal view controller, the system creates a parent-child relationship between the view controller that did the presenting and the view controller that was presented. Specifically, the view controller that did the presenting updates its `modalViewController` property to point to its presented (child) view controller. Similarly, the presented view controller updates its `parentViewController` property to point back to the view controller that presented it. Figure 6-1 shows the relationship between the view controller managing the main screen in the Calendar application and the modal view controller used to create new events.

Figure 6-1 Modal views in the calendar application.



Any view controller object can present any other single view controller modally. This is true even for view controllers that were themselves presented modally. In other words, you can chain modal view controllers together as needed, presenting new modal view controllers on top of other modal view controllers as needed. Figure 6-2 shows a visual representation of the chaining process and the actions that initiate it. In this case, when the user taps the icon in the camera view, the application presents a modal view controller with the user's photos. Tapping the action button in the photo library's toolbar prompts the user for an appropriate action and then presents another modal view controller (the people picker) in response to that action. Selecting a contact (or canceling the people picker) dismisses that interface and takes the user back to the photo library. Tapping the Done button then dismisses the photo library and takes the user back to the camera interface.

Figure 6-2 Creating a chain of modal view controllers

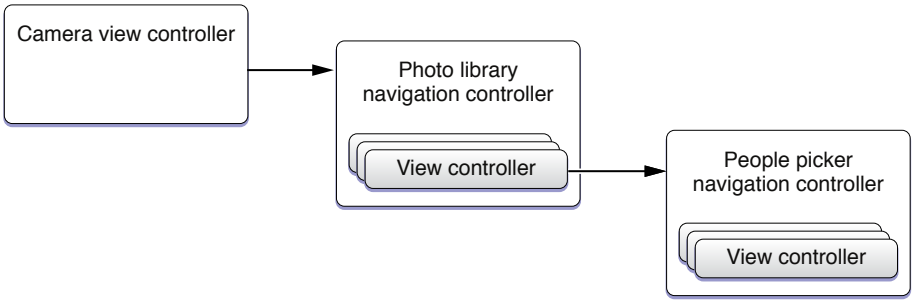


Each view controller in a chain of modally presented view controllers has pointers to the other objects surrounding it in the chain. In other words, a modal view controller that presents another modal view controller has valid objects in both its `parentViewController` and `modalViewController` properties. You can use these relationships to trace through the chain of view controllers as needed. For example, if the user cancels the current operation, you could remove all objects in the chain by dismissing the first modally presented view controller. In other words, dismissing a modal view controller dismisses not only that view controller but any view controllers it presented modally.

In Figure 6-2 (page 111), a point worth noting is that the modally presented view controllers are both navigation controllers. You can present `UINavigationController` objects modally in the same way that you would present a custom view controller. (In rare cases, you could even present a tab bar controller.)

When presenting a navigation controller modally, you always present the `UINavigationController` object itself, rather than presenting any of the view controllers on its navigation stack. However, individual view controllers on the navigation stack may themselves present other view controllers modally, including other navigation controllers. Figure 6-3 shows more detail of the objects that would be involved in the preceding example. As you can see, the people picker is not presented by the photo library navigation controller but by one of the custom view controllers on its navigation stack.

Figure 6-3 Presenting navigation controllers modally

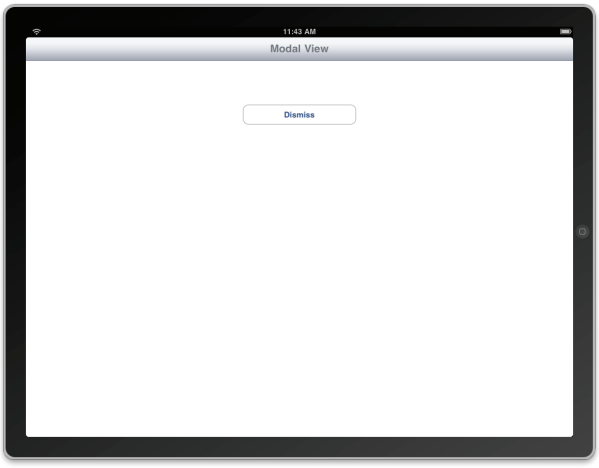


Configuring the Presentation Style for Modal Views

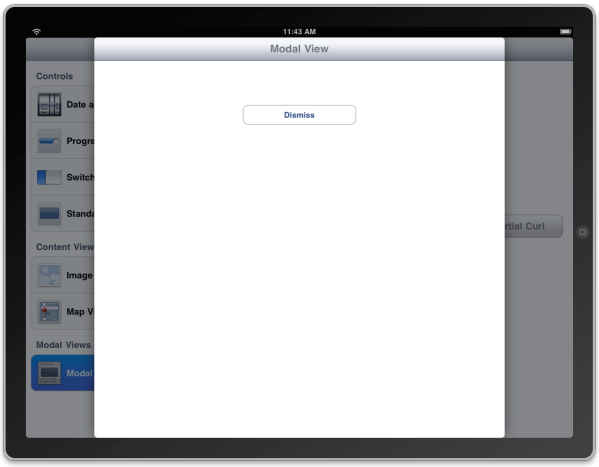
For iPad applications, you can present content modally using several different styles. In iPhone applications, modally presented views always cover the visible portion of the window, but when running on an iPad, view controllers use the value in their `modalPresentationStyle` property to determine their appearance when presented modally. Different options for this property allow you to present the view controller so that it fills all or only part of the screen.

Figure 6-4 shows the core presentation styles that are available. (The `UIModalPresentationCurrentContext` style lets a view controller adopt the presentation style of its parent.) In each modal view, the dimmed areas show the underlying content but do not allow taps in that content. Therefore, unlike a popover, your modal views must still have controls that allow the user to dismiss the modal view.

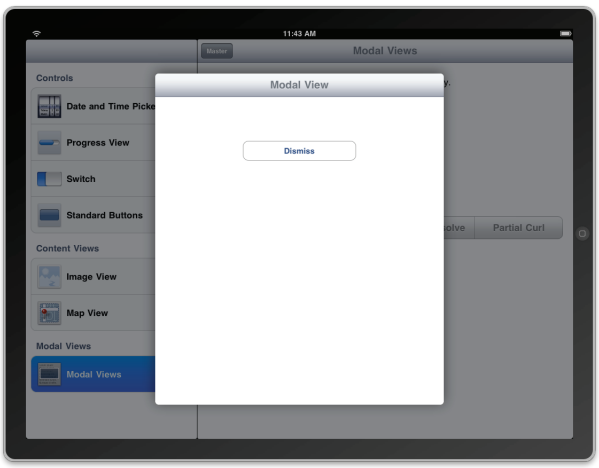
Figure 6-4 Modal presentation styles



`UIModalPresentationFullScreen`



`UIModalPresentationPageSheet`



`UIModalPresentationFormSheet`

For guidance on when to use the different presentation styles, see “Popover (iPad Only)” in *iOS Human Interface Guidelines*.

Presenting a View Controller Modally

To present a view controller modally, you must do the following:

1. Create the view controller you want to present.
2. Set the `modalTransitionStyle` property of the view controller to the desired value.
3. Assign a delegate object where appropriate. (The delegate is used primarily by system view controllers to notify your code when the view controller is ready to be dismissed. For more information, see "Presenting Standard System Modal View Controllers" (page 117).)
4. Call the `presentModalViewController:animated:` method of the current view controller, passing in the view controller you want to present modally.

The `presentModalViewController:animated:` method presents the view for the specified view controller object and configures the parent-child relationships between the modal view controller and the current view controller. Unless you are restoring your application to some previous state, you usually want to animate the appearance of a modal view controller. The transition style you should use depends on how you plan to use the presented view controller. Table 6-1 lists the transition styles you can assign to the `modalTransitionStyle` property of the presented view controller and how you might use each one.

Table 6-1 Transition styles for modal view controllers

Transition style	Usage
`UIModalTransitionStyleCoverVertical`	Use this style when you want to interrupt the current workflow to gather information from the user. You can also use it to present content that the user might or might not modify. For this style of transition, custom view controllers should provide buttons to dismiss the view controller explicitly. Typically, this would be a Done button and an optional Cancel button. If you do not explicitly set a transition style, this style is used by default.
`UIModalTransitionStyleFlipHorizontal`	Use this style to change the work mode of your application temporarily. The most common usage for this style is to display settings that might change frequently, such as in the Stocks and Weather applications. These settings could be meant for the entire application or they could be specific to the current screen. For this style of transition, you usually provide some sort of button to return the user to the normal running mode of your application.
`UIModalTransitionStyleCrossDissolve`	Use this style to present an alternate interface when the device changes orientations. In such a case, your application would be responsible for presenting and dismissing the alternate interface in response to orientation change notifications. Media-based applications can also use this style to fade in screens displaying media content. For an example of how to implement an alternate interface in response to device orientation changes, see "Creating an Alternate Landscape Interface" (page 50).

Listing 6-1 shows an example from a custom recipes application of how to present a modal controller. When the user adds a new recipe, the application prompts the user for basic information about the recipe by presenting a navigation controller modally. A navigation controller was chosen so that there would be a standard place to put a Cancel and Done button. Using a navigation controller also makes it easier to expand the new recipe interface in the future. All you would have to do is push new view controllers on the navigation stack.

Listing 6-1 Presenting a view controller modally

```
- (void)add:(id)sender {
    // Create the root view controller for the navigation controller
    // The new view controller configures a Cancel and Done button for the
    // navigation bar.
    RecipeAddViewController *addController = [[RecipeAddViewController alloc]
                        initWithNibName:@"RecipeAddView" bundle:nil];

    // Configure the RecipeAddViewController. In this case, it reports any
    // changes to a custom delegate object.
    addController.delegate = self;

    // Create the navigation controller and present it modally.
   UINavigationController *navigationController = [[UINavigationController alloc]
                             initWithRootViewController:addController];
    [self presentModalViewController:navigationController animated:YES];

    // The navigation controller is now owned by the current view controller
    // and the root view controller is owned by the navigation controller,
    // so both objects should be released to prevent over-retention.
    [navigationController release];
    [addController release];
}
```

When the user taps either the Done or Cancel button from the new recipe entry interface, the application dismisses the view controller and returns the user to the main view. Although you could call the `dismissModalViewControllerAnimated:` method directly from the action method associated with either button, a more robust approach involves using a delegate object, as described in "Dismissing a Modal View Controller" (page 115).

Dismissing a Modal View Controller

When it comes time to dismiss a modal view controller, the preferred approach is to let the parent view controller do the dismissing. In other words, the same view controller that presented the modal view controller should also take responsibility for dismissing it whenever possible. Although there are several techniques for notifying a parent view controller that it should dismiss its modally presented child, the preferred technique is delegation.

In a delegate-based model, the view controller being presented modally must define a protocol for its delegate to implement. The protocol defines methods that are called by the modal view controller in response to specific actions, such as taps in a Done button. The delegate is then responsible for implementing these methods and providing an appropriate response. In the case of a parent view controller acting as a delegate for its modal child, the response would include dismissing the child view controller when appropriate.

This use of delegation to manage interactions with a modal view controller has some key advantages over other techniques:

- The delegate object has the opportunity to validate or incorporate changes from the modal view controller before that view controller is dismissed.
- The use of a delegate promotes better encapsulation because the modal view controller does not have to know anything about the parent object that presented it. This enables you to reuse that modal view controller in other parts of your application.

To illustrate the implementation of a delegate protocol, consider the recipe view controller example that was used in "Presenting a View Controller Modally" (page 114). In that example, a recipes application presented a modal view controller in response to the user wanting to add a new recipe. Prior to presenting the modal view controller, the current view controller made itself the delegate of the `RecipeAddViewController` object. Listing 6-2 shows the definition of the delegate protocol for `RecipeAddViewController` objects.

Listing 6-2 Delegate protocol for dismissing a modal view controller

```
@protocol RecipeAddDelegate <NSObject>
// recipe == nil on cancel
- (void)recipeAddViewController:(RecipeAddViewController *)recipeAddViewController
                    didAddRecipe:(MyRecipe *)recipe;
@end
```

When the user taps the Cancel or Done button in the new recipe interface, the `RecipeAddViewController` object calls the preceding method on its delegate object. The delegate is then responsible for deciding what course of action to take.

Listing 6-3 shows the implementation of the delegate method that handles the addition of new recipes. This method is implemented by the view controller that presented the `RecipeAddViewController` object modally. If the user accepted the new recipe—that is, the recipe object is not `nil`—this method adds the recipe to its internal data structures and tells its table view to refresh itself. (The table view subsequently reloads the recipe data from the same `recipesController` object shown here.) As its final act, the delegate method dismisses the modal view controller.

Listing 6-3 Dismissing a modal view controller using a delegate

```
- (void)recipeAddViewController:(RecipeAddViewController *)recipeAddViewController
                    didAddRecipe:(Recipe *)recipe {
    if (recipe) {
        // Add the recipe to the recipes controller.
        int recipeCount = [recipesController countOfRecipes];
        UITableView *tableView = [self tableView];
        [recipesController insertObject:recipe inRecipesAtIndex:recipeCount];

        [tableView reloadData];
    }
    [self dismissModalViewControllerAnimated:YES];
}
```

Presenting Standard System Modal View Controllers

In iOS, there are a number of standard system view controllers that are designed to be presented modally by your application. The basic rules for presenting these view controllers are the same as for your custom view controllers. However, because your application does not have access to the view hierarchy managed by the system view controllers, you cannot simply implement actions for the controls in the views. Interactions with the system view controllers typically take place through a delegate object.

Each system view controller defines a corresponding protocol, whose methods you implement in your delegate object. Each delegate usually implements a method to accept whatever item was selected or cancel the operation. Your delegate object should always be ready to handle both cases. One of the most important things the delegate must do is dismiss the presented view controller by calling the `dismissModalViewControllerAnimated:` method of the view controller that did the presenting—in other words, the parent of the modal view controller.

Table 6-2 lists several of the standard system view controllers found in iOS. For more information about each of these classes, including the features it provides, see the corresponding class reference documentation.

Table 6-2 Standard system view controllers

Framework	View controllers

Address Book UI	`ABNewPersonViewController`
	`ABPeoplePickerNavigationController`
	`ABPersonViewController`
	`ABUnknownPersonViewController`

Event Kit UI	`EKEventViewController` `EKEventEditViewController`

Game Kit	`GKPeerPickerController`
	`GKAchievementViewController`
	`GKMatchmakerViewController`
	`GKLeaderboardViewController`

Message UI	`MFMailComposeViewController` `MFMessageComposeViewController`
Media Player	`MPMediaPickerController` `MPMoviePlayerViewController`
UIKit	`UIImagePickerController` `UIVideoEditorController`

Note: Although the `MPMoviePlayerController` class in the Media Player framework might technically be thought of as a modal controller, the semantics for using it are slightly different. Instead of presenting the view controller yourself, you initialize it and tell it to play its media file. The view controller then handles all aspects of presenting and dismissing its view. (However, the `MPMoviePlayerViewController` class can be used instead of `MPMoviePlayerController` as a standard view controller for playing movies.)

Combined View Controller Interfaces

The UIKit framework provides only a handful of standard view controllers for implementing your application interface:

- A custom view controller (or table view controller) provides a self-contained set of views to present.
- A navigation controller presents multiple view controllers hierarchically.
- A tab bar controller presents multiple view controllers as different operational modes of the application.
- A split view controller presents two view controllers side-by-side in landscape orientations. (In portrait orientations, one view controller is displayed in a popover.)

You can use these view controllers singly or in conjunction with other view controllers to create even more sophisticated interfaces. When combining view controllers, however, the order of items in the preceding list is significant. The general rule is that each view controller can incorporate the view controllers preceding it in the list. Thus, a navigation controller can incorporate custom view controllers, and a tab bar controller can incorporate both navigation controllers and custom view controllers. However, a navigation controller should not incorporate a tab bar controller as part of its navigation interface. The resulting interface would be confusing to users because the tab bar would not be consistently visible.

Because modal view controllers represent an interruption of sorts anyway, 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 new interface style supplants the style of its parent, albeit only temporarily.

The following sections show you how to combine table view, navigation and tab bar controllers in your iOS applications. For more information on using split view controllers in iPad applications, see "iPad-Specific Controllers" (page 99).

Adding a Navigation Controller to a Tab Bar Interface

An application 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. You never want to push a tab bar controller onto the navigation stack of a navigation controller. Doing so creates an unusual situation whereby the tab bar appears only while a specific view controller is at the top of the navigation stack. Tab bars are designed to be persistent, and so this transient approach can be confusing to users.

The most common way to use a tab bar controller is to embed its view in your application's main window. Therefore, the following sections show you how to configure your application's main window to include a tab bar controller and one or more navigation controllers. Examples are included for doing this both programmatically and using Interface Builder. If you need to present a tab bar controller modally, it is generally recommended that you create the relevant objects programmatically.

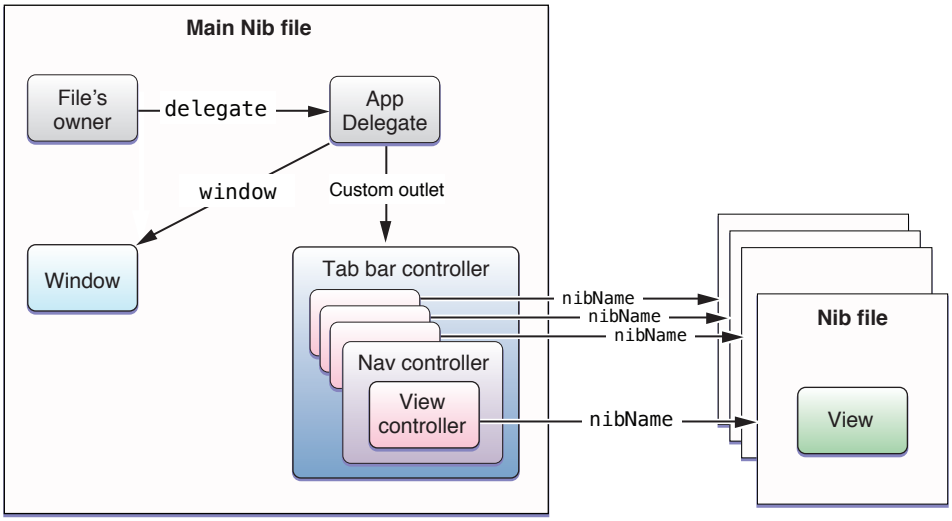
Note: When embedding navigation controllers in a tab bar interface, you should embed only generic `UINavigationController` objects and not the system view controllers that are based on the `UINavigationController` class. Although the system provides custom navigation controllers for selecting contacts, picking images, and implementing other behaviors, these view controllers were designed to be presented modally in nearly all cases. For information about how to use a specific view controller, see the reference documentation for that class.

Creating the Objects in Interface Builder

The process for combining tab bar and navigation controllers in a nib file is relatively straightforward. The only real difference is how you create the relationship between the tab bar controller and the navigation controller. When using these objects by themselves, each one takes on the role of the root view for the application's window. When combined, however, only the tab bar controller assumes this role. Instead of providing the root view for the window, the navigation controller acts as the root view for a tab in the tab bar interface.

Figure 7-1 shows the configuration of the objects that you need to create in your nib file. In this example, the first three tabs of the tab bar interface use custom view controllers but the last tab uses a navigation controller. One additional view controller is then added to act as the navigation controller's root view controller. To manage memory better, each of the custom view controllers (including the root view controller of the navigation controller) stores its corresponding view in a different nib file.

Figure 7-1 Mixing navigation and tab bar controllers in a nib file



Assuming you are starting from a generic main nib file (one that does not include a tab bar controller), you would use the following steps to create the objects from Figure 7-1 (page 120) in Interface Builder:

1. Drag a tab bar controller object from the library to your Interface Builder document window.

   When you add a tab bar controller to a nib file, Interface Builder also adds a tab bar view, two root view controllers, and two tab bar items (one for each view controller).

2. Save a reference to the tab bar controller using an outlet.

In order to access the tab bar controller at runtime, you either need to use an outlet or you must explicitly retrieve the nib file's top-level objects when you load the nib file. Using an outlet is generally much simpler. To add an outlet for both the tab bar controller and window, you need to include code similar to the following in your application delegate's header file:

```
@interface MyAppDelegate : NSObject <UIApplicationDelegate> {
   UITabBarController*  tabBarController;
   UIWindow *window;
}
@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet UITabBarController *tabBarController;
@end
```

After adding the outlet definition, create a connection from that outlet to the tab bar controller object.

3. Synthesize the properties from the preceding step by adding the following code to the implementation file of your application delegate class:

```
@synthesize window;
@synthesize tabBarController;
```

4. Add one View Controller object and one Navigation Controller object to the tab bar controller.

   The number of view controllers embedded inside your tab bar controller object determines the number of tabs displayed by your tab bar interface. Because the initial tab bar controller already has two generic view controllers, you need to add one more View Controller object (`UIViewController`) and one Navigation Controller object (`UINavigationController`).

   To add a view controller, do one of the following:

   - Drag the appropriate object from the library to the tab bar in the edit surface.
   - Drag the object from the library to the tab bar controller in your Interface Builder document window. The window must be in outline mode.

   When adding the navigation controller, you should either drag the appropriate object from the library or select the tab bar controller object and configure your view controller types using the Attributes inspector. Both of these options add the correct type of view controller object to your nib file. You should never add a navigation controller by dragging a generic View Controller object to the nib file and change its class name to the desired class type.

   To delete a view controller, select the view controller object in the edit surface or document window and press the Delete key.

5. Arrange the view controllers in the order you want them to appear in the tab bar interface.

   You can rearrange view controllers (and the corresponding tabs) by dragging the tabs displayed on the tab bar controller edit surface or by dragging the view controllers in the Interface Builder document window (outline mode only). Although the edit surface shows all of the tabs, only five are displayed at runtime. If your tab bar controller contains six or more view controllers, only the first four are displayed in the tab bar initially. The last spot on the tab bar is reserved for the More view controller, which presents the remaining view controllers.

6. Configure the view controllers.

   For each root view controller, you should configure the following attributes:

- Set the class of each custom view controller object using the Identity inspector. For generic View Controller objects, change the class name to the custom subclass you want to use to display the contents of that tab. Do not change the class of the Navigation Controller object itself but do set the class of the navigation controller's embedded custom view controller.
- Provide a view for each custom view controller. The preferred way to do this is to configure the NIB Name attribute of each custom view controller with the name of the nib file containing the view. Although you can include each view in the same nib file as the view controller, doing so is not recommended. For information on how to configure the nib file for a custom view controller, see "Storing the View in a Detached Nib File" (page 33).
- As appropriate, configure any style or appearance information for any of the view controllers.

7. Configure the tab bar item for each view controller.

   You can select the tab bar item from the tab bar controller edit surface or from the Interface Builder document window when it is in outline or browser modes. Using Interface Builder, you can specify the title, image, and badge of a tab bar item. Alternatively, you can set the tab bar item to one of the standard system tabs by assigning a value to the Identifier property in the Attributes inspector.

8. Save your nib file.

Although the preceding steps configure the tab bar interface, they do not install it in your application's main window. To do that, you need to add some code to the `applicationDidFinishLaunching:` method of your application delegate, as shown in Listing 7-1. This is where you use the outlet you created for the tab bar controller in your application delegate.

Listing 7-1 Installing the combined interface in your application's window

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
    [window addSubview:tabBarController.view];
}
```

Creating the Objects Programmatically

If you are creating a combined tab bar and navigation interface programmatically for your application'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.
6. Add the tab bar controller's view to your application's main window.

Listing 4-1 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 would provide yourself. And the `init` method of each custom view controller is also a method that you would provide to initialize the view controllers. The `tabBarController` and `window` variables are declared properties of the class that retain their values.

Listing 7-2 Creating a tab bar controller from scratch

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

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

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

   // Add the tab bar controller's current view as a subview of the window
   window = [[[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]]
autorelease];
   [window addSubview:tabBarController.view];
}
```

Note: The preceding example autoreleases objects as a convenience, rather than releasing them directly. Because the `applicationDidFinishLaunching:` method is called once in your application, autoreleasing objects at this point is unlikely to have a significant performance impact. However, if you do notice a performance impact in your own application, you could release each of the objects directly at the end of the method.

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 the View for Your View Controller" (page 33).

Displaying a Navigation Controller Modally

It is perfectly reasonable (and relatively common) to present navigation controllers modally from your application. In fact, many of the standard system view controllers (including `UIImagePickerController` and `ABPeoplePickerNavigationController`) are navigation controllers that were 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 7-3 shows an example of how you would 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 would need to define and configure with views. And the `currentViewController` object is a reference to the currently visible view controller that you would also need to provide.

Listing 7-3 Displaying a navigation controller modally

```
MyViewController1*  rootVC = [[MyViewController1 alloc] init];
MyViewController2*  nextVC = [[MyViewController2 alloc] init];

// Create the nav controller and add the view controllers.
UINavigationController*  theNavController = [[UINavigationController alloc]
                          initWithRootViewController:rootVC];
[theNavController pushViewController:nextVC animated:NO];

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

// Release the view controllers to prevent over-retention.
[rootVC release];
[nextVC release];
[theNavController release];
```

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 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.

Note: When presenting a navigation controller modally, it is often simpler to create and configure your navigation controller object programmatically. Although you could also use Interface Builder to do so, doing so is generally not recommended.

For information on how to present view controllers (including navigation controllers) modally, see "Presenting a View Controller Modally" (page 114). For additional information on how to configure a navigation controller for use in your application, see "Creating a Navigation Interface" (page 66).

Displaying a Tab Bar Controller Modally

It is possible (albeit uncommon) to present a tab bar controller modally in your application. Tab bar interfaces are normally installed in your application'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 application'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. This means that 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.

Note: When presenting a tab bar controller modally, it is often simpler to create and configure your tab bar controller object programmatically. Although you could also use Interface Builder to do so, doing so is generally not recommended.

For information on how to present view controllers (including navigation controllers) modally, see "Presenting a View Controller Modally" (page 114). For information on how to configure a tab bar controller programmatically, see "Creating a Tab Bar Interface Programmatically" (page 93).

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 application takes the user to a list of songs in that playlist.

When it comes to managing tables, one way to do so 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 7-4 shows an example of how you would navigate to the next level of data in a navigation interface. Whenever the user taps a specific row in the current table, you would use the information associated with that row to initialize a new view controller, which you would then push onto the navigation stack. The `initWithTable:andDataAtIndexPath:` method is a custom method that you would have to implement yourself; its job would be to get the data object for the row and use it to initialize the next level view controller.

Listing 7-4 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 Programming Guide for iOS*.

Date	Notes
2011-01-07	Fixed several typos.
2010-11-12	Added information about iPad-only controller objects.
2010-07-08	Changed the title from "View Controller Programming Guide for iPhone OS."
2010-05-03	Fixed some typos.
2010-02-24	Fixed several typos and updated the figure for the two-step rotation process.
2009-10-19	Rewrote the document and expanded the content to address iOS 3.0 changes.
2009-05-28	Added a note about the lack of iOS 3.0 support.
2011-03-04	Expanded rotated application details.
2008-10-15	Updated obsolete references to the iOS Programming Guide.
2008-09-09	Corrected typos.
2008-06-23	New document that explains how to use view controllers to implement radio, navigation, and modal interfaces.

Document Revision History

Glossary

container view controller A view controller that coordinates the interaction of other view controllers in order to present a specific type of user interface.

custom view controller A view controller that you define for the express purpose of displaying some content on the screen.

detached nib file A nib file that contains the view for a given view controller but not the view controller object itself. The view controller is instead loaded from a separate nib file or created programmatically.

integrated nib file A nib file that contains both a view controller and its associated view.

modal view controller A view controller presented on top of the current view controller using a special type of transition. Modal view controllers are often used to gather information from the user but may be used in other ways too.

More view controller A system-supplied view controller that manages the presentation of extra tabs in a tab bar interface. This view controller is displayed only when the number of tabs exceeds the number that can be displayed simultaneously on screen.

navigation interface The style of interface that is presented by a navigation controller's view. A navigation interface includes a navigation bar along the top of the screen to facilitate navigating between different screens.

navigation stack The list of view controllers currently being managed by a navigation controller. The view controllers on the stack represent the content currently being displayed by a navigation interface.

root view controller The first view controller presented in a navigation interface, or the first view controller associated with a tab in a tab bar interface. A root view controller acts as an anchor for the given interface. Whereas most view controllers in a navigation or tab bar interface are added and removed dynamically, the root view controller of an interface is added at initialization time and can never be removed.

tab bar interface The style of interface that is presented by a tab bar controller's view. A tab bar interface includes one or more tabs at the bottom of the screen. Tapping a tab changes the currently displayed screen contents.

view controller An object that descends from the `UIViewController` class. View controllers coordinate the interactions between a set of views and the custom data presented by those views.