
View Controller Programming Guide for iPhone OS

User Experience: Windows & Views



2008-06-23



Apple Inc.
© 2008 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

Apple, the Apple logo, and iPod are trademarks of Apple Inc., registered in the United States and other countries.

iPhone is a trademark of Apple Inc.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND

YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction [Introduction](#) 7

[Who Should Read This Document](#) 7
[Organization of This Document](#) 7
[See Also](#) 8

[What Are View Controllers?](#) 9

[User Interface Design Patterns](#) 11

[View Controller Classes](#) 15

[Using View Controllers](#) 19

[Using Tab Bar Controllers](#) 21

[Creating a View Controller Subclass for a Tab Bar Controller](#) 23
[Creating a Tab Bar Controller](#) 25

[Using Navigation Controllers](#) 27

[Creating a View Controller Subclass for a Navigation Controller](#) 31
[Creating a Navigation Controller](#) 32
[Creating Custom Buttons and Views for the Navigation Item](#) 33
[Using Edit and Done Buttons](#) 35

[Using Modal View Controllers](#) 37

[Combining Tab Bar and Navigation Controllers](#) 41

[Using Table Views for Hierarchical Data](#) 43

[Autorotating Views](#) 45

[Document Revision History](#) 49

C O N T E N T S

Figures, Tables, and Listings

What Are View Controllers? 9

Figure 1 Layout of views 9

User Interface Design Patterns 11

Figure 1 Radio interface 12
Figure 2 Navigation interface 13
Figure 3 Modal interface 13
Figure 4 Radio and navigation interfaces 14

View Controller Classes 15

Figure 1 View controller classes 15
Figure 2 View controller object diagram 16

Using View Controllers 19

Listing 1 Creating a view 19

Using Tab Bar Controllers 21

Figure 1 Tab bar controller components 22
Figure 2 Tab bar item components 24
Figure 3 Badge values 24
Figure 4 Customizing the tab bar 25
Table 1 Images displayed by the tab bar 24
Listing 1 Initializing a view controller managed by a tab bar controller 23

Using Navigation Controllers 27

Figure 1 Navigation item components 27
Figure 2 Single view controller on the stack 28
Figure 3 Multiple view controllers on the stack 28
Figure 4 Navigation controller components 30
Figure 5 Custom navigation item views 33
Figure 6 Using an Edit/Done button 36
Listing 1 Initializing a view controller managed by a navigation controller 32

Listing 2 Setting navigation item properties 34

Using Modal View Controllers 37

Figure 1 Modal view controllers 38

Figure 2 Navigation controllers with modal view controllers 39

Combining Tab Bar and Navigation Controllers 41

Figure 1 Tab bar controller containing a navigation controller 41

Autorotating Views 45

Figure 1 Autorotating view controllers 45

Listing 1 Subclasses enabling autorotation 45

Introduction

Important: This is a preliminary document for an API or technology in development. Although this document has been reviewed for technical accuracy, it is not final. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future seeds of the API or technology. For information about updates to this and other developer documentation, view the New & Updated sidebars in subsequent documentation seeds.

View controllers are a set of classes in UIKit that implement common user interface design patterns for presenting numerous application views to the user on iPhone OS. By using view controllers, you eliminate redundant code in your applications and provide a consistent and familiar user interface. View controllers provide support for tab bars, navigation bars, modal views, and table views. View controllers also simplify autorotating your user interface when the orientation changes.

Who Should Read This Document

You should read this document if you are implementing an iPhone OS application. Most productivity applications that contain a lot of user data or records benefit from using view controllers to organize and present content on small hand-held devices. Even a single view graphics application benefits from using view controllers for autorotation.

Organization of This Document

This article provides an introduction to view controllers and explains why you should use them:

- [“What Are View Controllers?”](#) (page 9) describes the benefits of using view controllers and how they fit into the Model-View-Controller design pattern.

These articles describe the user interface and object-oriented designs implemented by view controllers:

- [“User Interface Design Patterns”](#) (page 11) describes the common user interface design patterns used in iPhone OS. You use view controllers to implement these types of user interfaces.

- [“View Controller Classes”](#) (page 15) provides an overview of the different view controller classes and how they relate to other classes you use to implement your user interface.

These articles cover common programming tasks—they contain step-by-step instructions and sample code:

- [“Using View Controllers”](#) (page 19) describes the root view controller class that you subclass to manage your application views.
- [“Using Tab Bar Controllers”](#) (page 21) describes how to use a tab bar controller to create a radio-style interface.
- [“Using Navigation Controllers”](#) (page 27) describes how to use a navigation controller to create a navigation-style interface including how to set up navigation items for your application views.
- [“Using Modal View Controllers”](#) (page 37) describes how to use modal view controllers to present a view on top of another view.
- [“Combining Tab Bar and Navigation Controllers”](#) (page 41) describes how to combine different types of controllers for more complex user interfaces—for example, how to wrap a view controller in a navigation controller and add it to a tab bar controller.
- [“Using Table Views for Hierarchical Data”](#) (page 43) describes how to use a table view controller to represent hierarchical data. Explains how to create a drill-down user interface using table views and explains how to allow editing of table content.
- [“Autorotating Views”](#) (page 45) describes how to autorotate your user interface when the orientation changes.

See Also

If you are new to development for iPhone OS, read these documents first:

- *iPhone OS Programming Guide* provides basic information for creating iPhone OS applications including an introduction to the application architecture and other frameworks available to developers.
- *iPhone Human Interface Guidelines* provides user interface guidelines for designing iPhone OS applications.

If you are using Interface Builder to create view controllers, read “iPhone OS Interface Objects” in *Interface Builder User Guide*.

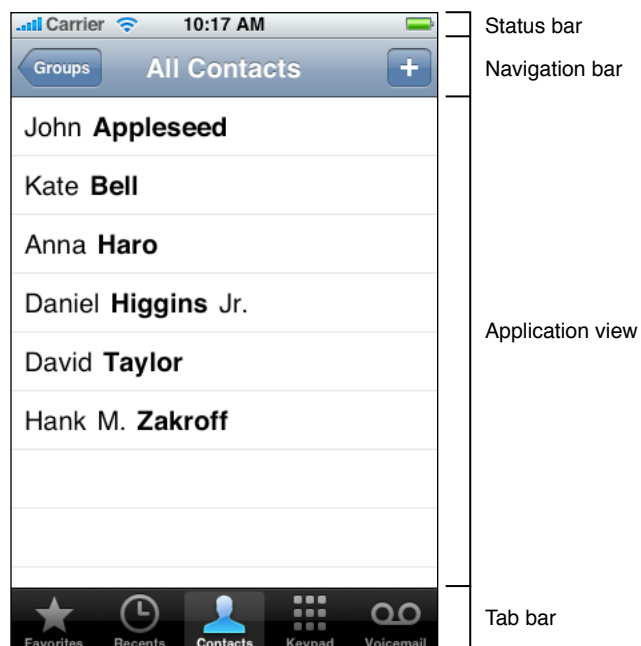
If you want to use table views for hierarchical data, read *Table View Programming Guide for iPhone OS* after this guide.

What Are View Controllers?

View controllers provide the basic user interface logic for presenting numerous application views to the user. Apple developed several common user interface design patterns to help applications present multiple functions and large data sets to users on small hand-held touchscreen devices. View controllers provide the structure and functionality for these user interface design patterns.

A view controller manages your application view that appears between the upper and lower bars shown in [Figure 1](#) (page 9). The application view appears below the status bar and navigation bar if present. Similarly, it appears above the tab bar or toolbar if present. The application view contains a portion of data and controls you want to present to the user at a given time. The view controller simply manages the presentation of this view and the next view according to some established design pattern.

Figure 1 Layout of views



Using view controllers, you eliminate unnecessary and redundant code in your applications and provide a consistent user interface familiar to users. You do not need to implement the presentation of these application views. View controllers also promote good object-oriented design by separating user interface details from your application logic.

View controllers support the Model-View-Controller (MVC) design pattern, commonly used in object-oriented programming, where model objects represent application data and are persistent. Most of your custom data and logic are implemented by model objects, so it's your responsibility to implement them. View objects display model data to the user and typically, allow the user to edit models. Controller objects mediate between both model and view objects. In UIKit, view controllers present your application views and support a design pattern, but they do not automatically synchronize data between models and views. It's your responsibility to extend view controllers to add this functionality if it's needed. For example, view controllers are typically the delegate and data source objects for table views.

In addition to your custom view controllers, you use three specialized view controllers for managing the tab bar, navigation bar, and table views. Each of these controllers support an established user interface pattern on iPhone. You use a tab bar controller to present multiple modes to the user. You use a navigation controller to present views of hierarchical or sequential data. You use a table view controller to present lists of objects—for example, children of a parent object.

These view controllers not only manage the details of setting up user interface controls, but they also automatically resize and position the application's view above or below these controls. If you choose to use view controllers, you rarely need to access tab bar and navigation bar objects directly.

Even if your application is a graphics application, you'll want to use a view controller just to manage a single view and autorotate when the orientation changes. You can also use view controllers to present a modal view on top of another view—for example, to provide more details.

The remaining articles describe the common user interface design patterns implemented by view controllers, introduce view controller classes, and provide the details on how to implement each of the user interface design patterns discussed in [“User Interface Design Patterns”](#) (page 11).

User Interface Design Patterns

UIKit provides support for implementing common user interface design patterns. The patterns used on iPhone follow some basic guidelines. A good design divides the presentation of data into small, screen-size pieces. Each screen is beautifully designed and laid out without any clutter and with the important information placed conveniently at the top. Drilling down a hierarchy of objects or inspecting objects provides more details at each level.

The bars containing buttons, at the top and bottom of the screen, allow the user to move easily from one screen to the next. For example, the user taps buttons on the bottom bar to toggle the mode, taps back buttons on the top bar to traverse up a hierarchy of objects, taps an Edit button on the top bar to edit objects, and taps other buttons in the view below to drill down or inspect objects. Applications that display large data sets—most productivity applications—usually use this drill-down model.

This way of navigating in an application is only one of a number of common user interface design patterns for this small and unique touchscreen interface. The user interface pattern you choose determines what classes and methods you need to learn about. Applications with a number of separate application views to present usually fit into one of the following categories:

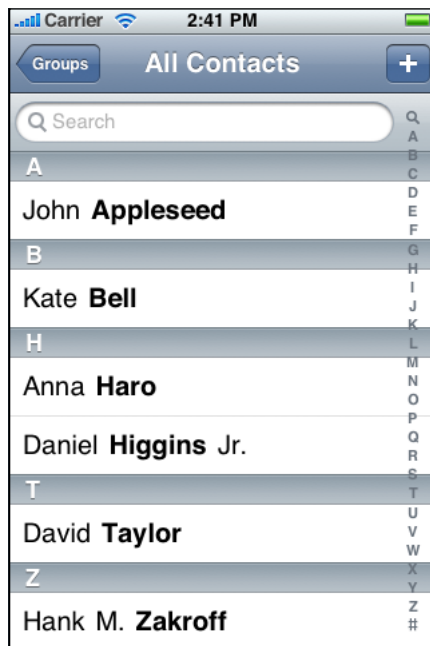
Command interfaces. Command interfaces have a set of functions the user executes by tapping buttons on a bottom bar called a toolbar. When a button on the toolbar is tapped, it is highlighted while the command executes. (This type of interface doesn't use view controllers. Read *UIToolbar Class Reference* for how to implement command interfaces.)

Radio interfaces. Radio interfaces have different modes that the user toggles between by tapping buttons on a bottom bar called a tab bar, where one item is selected at all times. For example, the Clock application has a simple radio interface. There are four buttons on the tab bar: World Clock, Alarm, Stopwatch, and Timer. Tapping a button simply changes the selected item on the tab bar and the view above the tab bar. If you tap World Clock, a list of clocks appear, as shown in Figure 1. If you tap Stopwatch, stopwatch controls appear.

Figure 1 Radio interface

Navigation interfaces. Navigation interfaces provide controls for navigating a hierarchy of objects, progressing from abstract to more detailed objects as you drill down the hierarchy. There's a navigation bar at the top of the screen and details are displayed in a view below. You can see this navigational hierarchy in the Settings, SMS, Notes, and Phone applications. The navigation bar displays the title of the selected object, and the object's children are displayed in tables. Tapping a child object in the table drills down the hierarchy and displays details about the child. If the child has no children, its properties are displayed. The user taps an Edit button in the navigation bar to modify the object's properties.

Figure 2 shows an example of what users see after tapping the Contacts button in the Phone application. The contacts mode in the Phone application lists groups. Tapping a group lists the people in that group. Tapping a person displays its properties. Tapping an Edit button allows you to change a person's properties. The navigation bar typically displays a back button on the left and the title of the current selection in the middle. The Edit button that appears on the right is optional.

Figure 2 Navigation interface

Modal interfaces. Modal interfaces present a view on top of another by animating it from below and covering everything but the status bar, as shown in Figure 3. Typically, tapping a Done button on the modal view—or in this case, a Save button—dismisses it and returns to the previous view. You can have multiple modal views creating a pile of views on top of each other. The user dismisses them individually or can pop back to a previous state of an underlying navigation bar. For example, a modal view appears when you click the new message button in the SMS application. Some applications present multiple modal views as you edit content.

Figure 3 Modal interface

Radio and navigation interfaces. Many applications combine radio, navigation, and modal interfaces to provide a multidimensional path for navigating screens of information. For example, in the Phone applications, you use the tab bar to switch modes, and in each mode except the keypad mode—the

mode when the Keypad button is tapped—there's a navigation bar, as shown in Figure 4. Only the contacts mode uses hierarchical navigation—you can navigate from groups to people—and the others use the navigation bar for buttons to filter and edit the list. If you tap the plus button on the Favorites or Contacts navigation bar, a modal view appears.

Figure 4 Radio and navigation interfaces



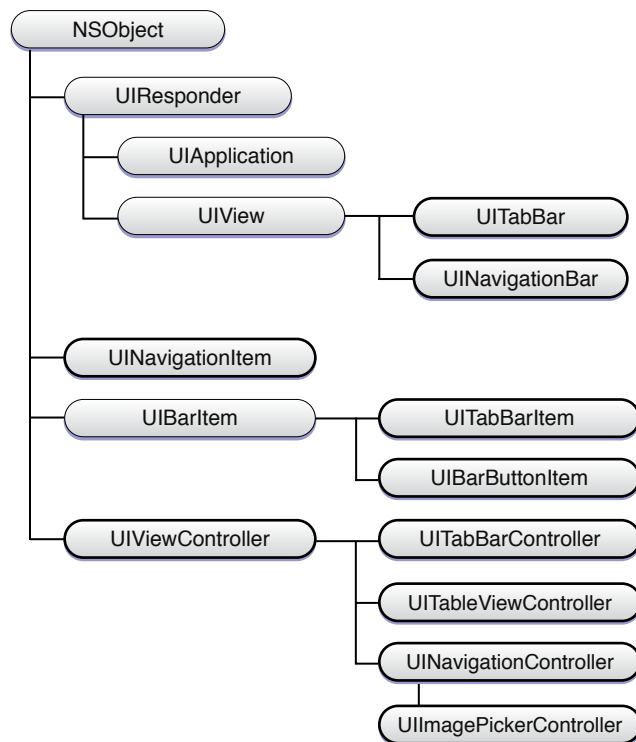
Table interfaces. In Figure 4 notice, too, the use of a table view with selectable rows representing children, for navigating hierarchical data.

UIKit supports all of these common user interface patterns, especially the radio and navigation style interfaces that require more data management and coordination with screen controls. By using view controllers, you don't have to write the code for managing the display of these application views.

View Controller Classes

The `UIViewController` class is a superclass for all controllers that manage a view. Figure 1 shows the class hierarchy of view controllers and their related classes. The `UITabBarController`, `UINavigationController`, and `UITableViewController` classes, described in this guide, inherit from the `UIViewController` class. You subclass `UIViewController` to create objects that manage your models and views.

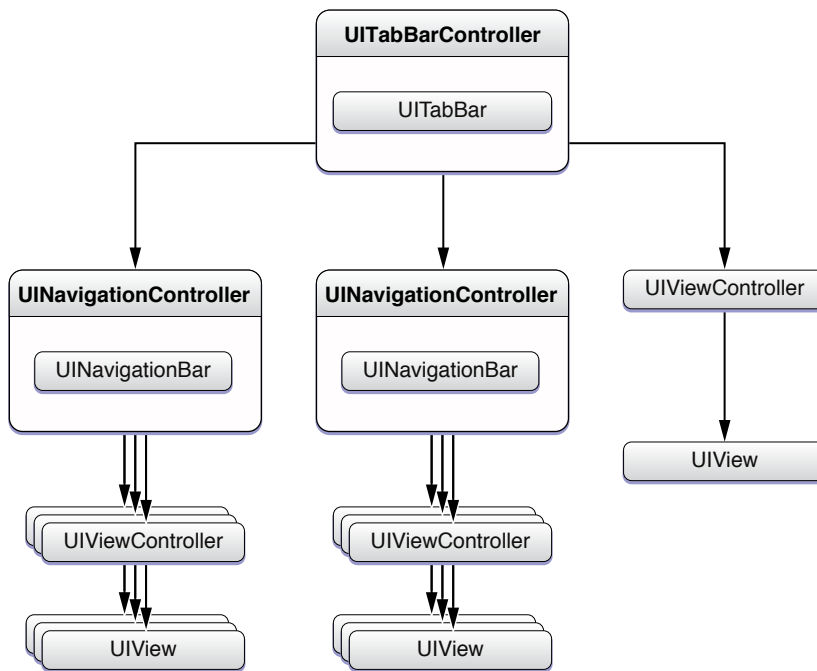
Figure 1 View controller classes



Instances of view controller subclasses you create can be used alone—a single view controller per application—or with other view controllers—specifically, `UITabBarController` and `UINavigationController` objects which, manage the complexity of navigation bars and tab bars.

Figure 2 illustrates the object graph of a `UITabBarController` root object that combines radio and navigation interfaces; it uses a tab bar and multiple navigation bars. Notice that `UITabBarController` and `UINavigationController` objects encapsulate `UITabBar` and `UINavigationController` objects—normally, you do not need to access these objects directly when using controllers. The following articles explain these objects and their relationships in detail. Which classes you learn more about depends on the style of your user interface.

Figure 2 View controller object diagram



Here are the user interfaces described in this guide, with the classes you use to implement them:

- If you want to use a view controller only to separate your controller logic from your models and views, and to autorotate your user interface, then you can subclass `UIViewController`, as described in [“Using View Controllers”](#) (page 19). To learn how to autorotate your interface when the orientation changes, read [“Autorotating Views”](#) (page 45).
- To implement radio interfaces, use the `UITabBarController` class, `UITabBarControllerDelegate` protocol, and `UIViewController` class, as described in [“Using Tab Bar Controllers”](#) (page 21).
- To implement navigation interfaces, use the `UINavigationController` and `UIViewController` classes, as described in [“Using Navigation Controllers”](#) (page 27).
- To present modal views, use the `UIViewController` class as described in [“Using Modal View Controllers”](#) (page 37).
- To combine radio and navigation interfaces, use the `UITabBarController` and `UINavigationController` classes, as described in [“Combining Tab Bar and Navigation Controllers”](#) (page 41).
- To use table views for presenting hierarchical data, use the `UITableView` class in combination with `UINavigationController`, as described in [“Using Table Views for Hierarchical Data”](#) (page 43).

The rest of this guide describes these relationships in more detail and provides sample code for implementing the different user interface design patterns described in [“User Interface Design Patterns”](#) (page 11).

Using View Controllers

View controllers help provide structure to your application code and separate the details of your model objects from your view objects. View controllers can also provide basic functionality such as autorotation and the ability to present one modal application view on top of another. You do not need to use a tab bar or navigation bar to use view controllers. You can use a view controller simply to manage a single application view that automatically rotates.

Whether you're using a tab bar or navigation controller, or just a single view controller, you still subclass `UIViewController` to create a view controller to manage your application view. If you use a single view controller in your application, you add its view to a window instead of adding the view controller to a tab bar or navigation controller.

Furthermore, you can set up your view controller objects programmatically or using Interface Builder. If you choose to use Interface Builder, you still create your view controller subclass and possibly write some glue code. This section discusses the methods you use to create view controllers programmatically and methods you need to implement in your subclasses, whether you use Interface Builder or not, to handle memory warnings. Read "iPhone OS Interface Objects" in *Interface Builder User Guide* for the steps to create view, tab bar, and navigation controllers using Interface Builder.

If you create a tab bar or navigation controller using Interface Builder, you may still create view controllers programmatically. For example, you typically add tab bar and navigation controller objects to your main nib file. Next you configure the tab bar and navigation items. When you add items to a tab bar or navigation controller in Interface Builder, a corresponding view controller object also appears in the nib file. There are several ways to configure the view controller objects in a nib file.

To add view controllers to a tab bar or navigation controller programmatically, create outlets for your tab bar and navigation controller objects so you can access them from your code and override the `applicationDidFinishLaunching:` application delegate method. Then follow the instructions in ["Using Tab Bar Controllers"](#) (page 21) and ["Using Navigation Controllers"](#) (page 27) to configure and add view controllers to these objects. You add the top level controller's view to the window to display the bars and your application views.

If you choose to create the views for your view controllers programmatically, then you need to implement the `loadView` method in your view controller subclass to set the `view` property. For example, Listing 1 shows a `loadView` method that simply creates a view with a background color.

Listing 1 Creating a view

```
// The simplest UIViewController subclass just overrides -loadView.
- (void)loadView
{
```

```

    UIView *view = [[UIView alloc] initWithFrame:[UIScreen
 mainScreen].applicationFrame];
    [view
 setAutoresizingMask:UIViewAutoresizingFlexibleHeight|UIViewAutoresizingFlexibleWidth];
    [view setBackgroundColor:_color];
    self.view = view;
    [view release];
}

```

You don't invoke the `loadView` method directly—it is invoked only when the view is needed for display. By implementing the `loadView` method, you hook into the default memory management behavior. If memory is low, a view controller may receive the `didReceiveMemoryWarning` message. The default implementation checks to see if the view is in use. If its view is not in the view hierarchy and the view controller implements the `loadView` method, its view is released. Later when the view is needed, the `loadView` method is invoked again to create the view.

If you use Interface Builder, it is recommended that you set up the views for your view controllers in separate nib files—that is, you create a nib file for each application view. In this case, you set the File's Owner to your view controller subclass and connect the File's Owner view outlet to the root view in the nib file. You can then create the view controller programmatically using the `initWithNibName:bundle:` method or by setting the view controller's nib file name in Interface Builder. When you set the nib file name, an instance of your view controller class is created and initialized with the contents of the specified nib file.

If you set the nib file name of a view controller and you need to perform additional setup after the nib file is loaded, override the `viewDidLoad` method, not the `loadView` method. The default behavior of the `loadView` method already releases the view when memory is low and reloads the nib file when the view is needed again. Similarly, do not override the `initWithNibName:bundle:` method to set up your view because the view is probably not created yet. However, you can add other non-view initialization code to the `initWithNibName:bundle:` method—for example, code that configures its tab bar or navigation bar items.

In many cases this default behavior of responding to low memory warnings is sufficient. If not, you should override the `didReceiveMemoryWarning` method in your view controller subclass to release additional unused objects—typically, expensive objects such as images that can be recreated later. Always incorporate the superclass implementation when overriding this method to benefit from the default behavior. Note that unless you explicitly retain view controllers, they are already released when popped from a navigation controller or when removed from a tab bar controller.

Finally, if you combine your view controller and its view in the same nib file, you need to handle all memory warnings yourself. Override the `didReceiveMemoryWarning` method to remove any unnecessary objects that you can easily recreate when needed later.

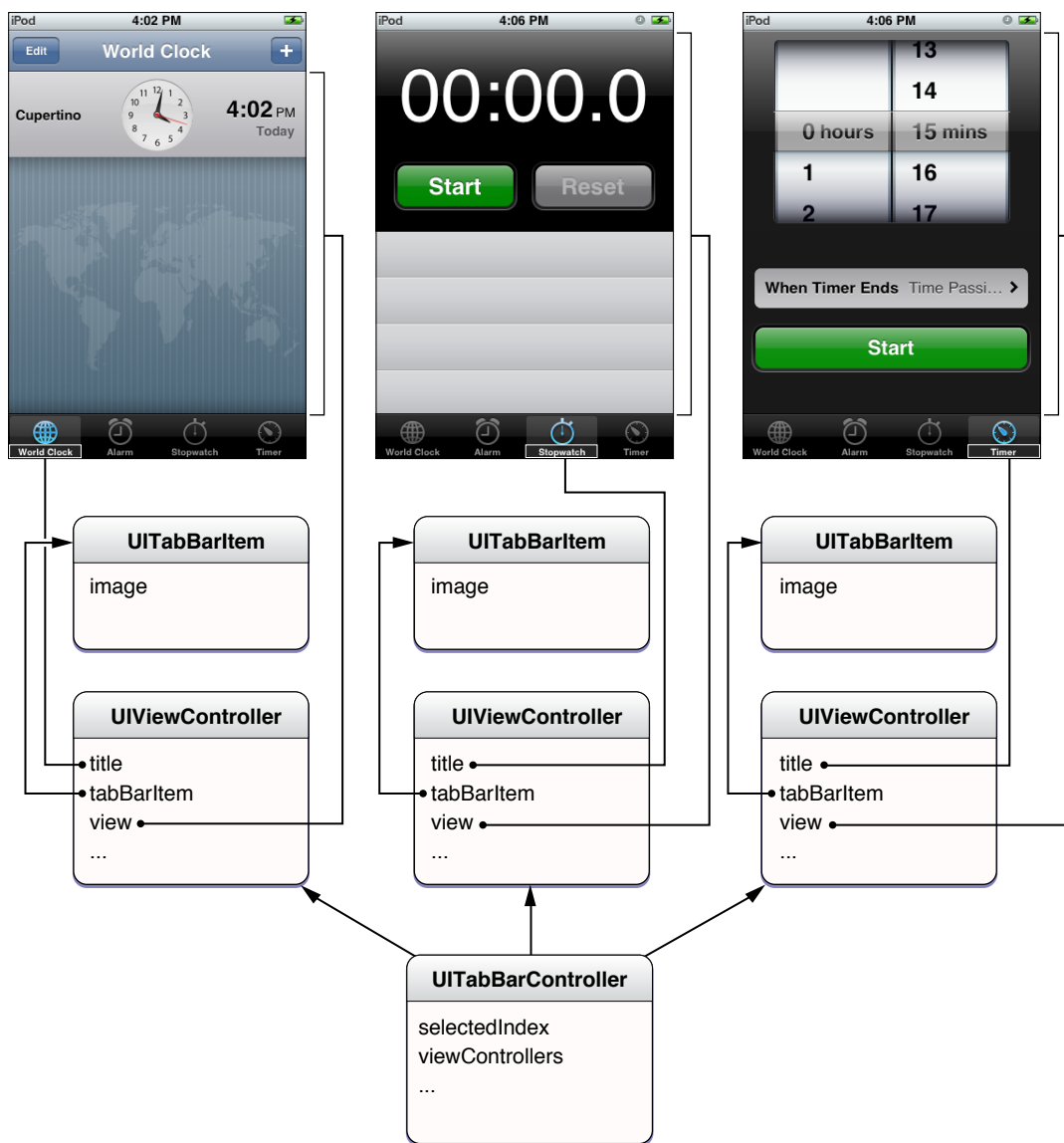
Also, the frame of your application view is automatically resized and repositioned by the view controller. Therefore, you should set the autoresize masks of any subviews—set the springs and struts in Interface Builder—so they are laid out correctly. This is also important to do if your interface rotates when the orientation changes. Read "Autore sizing Views" in *iPhone OS Programming Guide* to learn how to set the autoresize mask programmatically.

The remaining sections in this document describe how to combine view controllers to follow established design patterns. If you do not follow these guidelines, the results may be unpredictable. Specifically, if you embed a view controller into another view controller without using a navigation or tab bar controller as the root controller, view appear and disappear messages to the enclosing view controller are not sent. In this case, it is your responsibility to forward these messages to the embedded view controller.

Using Tab Bar Controllers

Use **tab bar controllers**, instances of `UITabBarController`, to implement a simple radio interface that changes the view when buttons are tapped on a tab bar. Tab bars are for toggling between application views with parallel or sibling functionality. This section describes only tab bars, not toolbars used to create command interfaces. Read *UIToolbar Class Reference* for how to use toolbars.

Figure 1 illustrates how tab bar controllers work. A `UITabBarController` object manages a tab bar by setting up its items and displaying the appropriate view when the selection changes. There's a `UIViewController` object per mode that encapsulates tab bar item properties and the view to display when that item is selected. A `UITabBarController` object simply contains an array of `UIViewController` objects whose views are swapped when the selection on the tab bar changes.

Figure 1 Tab bar controller components

Each screenshot in Figure 1 shows the state of the display when that view controller is selected. For example, when World Clock is tapped, the associated view controller's `tabBarItem` and `view` objects are displayed. The view for the World Clock controller happens to include a navigation bar at the top with custom buttons.

The first step is to create a subclass of `UIViewController`, typically one for each mode. Then you create instances of the custom view controllers and add them to the tab bar controller in the order in which you want them to appear on the tab bar.

Creating a View Controller Subclass for a Tab Bar Controller

To add a custom view to a tab bar, you must first create your custom view controller. You do this by subclassing `UIViewController`, since all controller objects inherit from this superclass. Into these subclasses go all your custom views and logic. For a view controller to be managed by a tab bar controller, certain properties need to be set, too.

In the steps that follow, you learn how to create the view and set the properties.

1. Create the view.

As described in [“Using View Controllers”](#) (page 19), you can create a view programmatically by setting the `view` property directly or by implementing the `loadView` method. You can also load a view from a nib file using the `initWithNibName:bundle:` method. Read [“Using View Controllers”](#) (page 19) for the methods you should implement in your view controller subclass whether you decide to use Interface Builder or not.

The view is automatically resized and displayed when its view controller is the selected view controller. It is placed between the navigation bar (if present) and the tab bar; otherwise, it is placed between the status bar and tab bar. For this reason, you should set the `autoresizeMask` of any subviews and not change the view’s frame later.

2. Set the tab bar item properties.

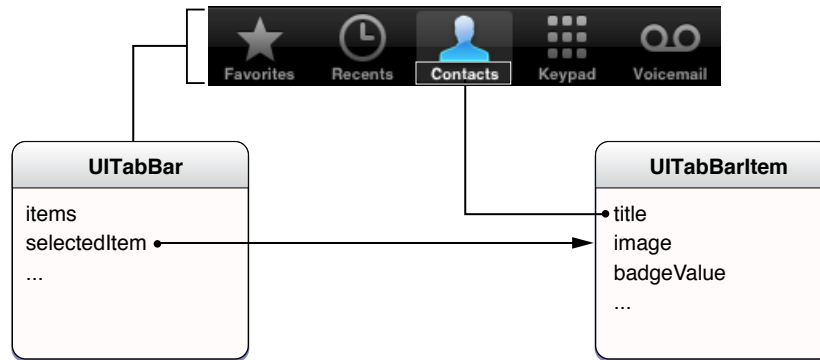
At a minimum, view controllers need to set a title and image to be managed by a tab bar controller. You can configure the tab bar item using Interface Builder or by overriding the `initWithNibName:bundle:` designated initializer in your view controller subclass, as shown in Listing 1.

Listing 1 Initializing a view controller managed by a tab bar controller

```
- (id)initWithNibName:(NSString *)nibName bundle:(NSBundle *)nibBundle
{
    self = [super initWithNibName:nibName bundle:nibBundle];
    if (self) {
        self.title = @"Green";
        self.tabBarItem.image = [UIImage imageNamed:@"Green.png"];
    }
    return self;
}
```




You change the appearance of the item by setting the properties of the `tabBarItem` property, as shown in Listing 1. Figure 2 shows the relationship between a tab bar item and its tab bar object.

Figure 2 Tab bar item components



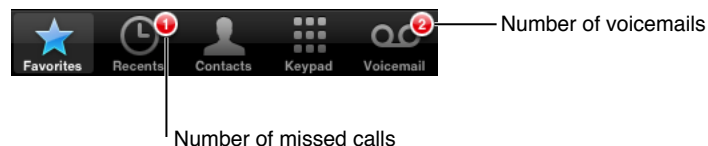
The unselected and selected images displayed by the tab bar are derived from the images that you set. The alpha values in the source image are used to create the other images—opaque values are ignored. Table 1 shows the results for a given source image. The title is displayed below the derived images.

Table 1 Images displayed by the tab bar

Source image	
Unselected image	
Selected image	

The `badgeValue` property contains optional text displayed in the upper-right corner of the tab bar item, as shown in Figure 3.

Figure 3 Badge values



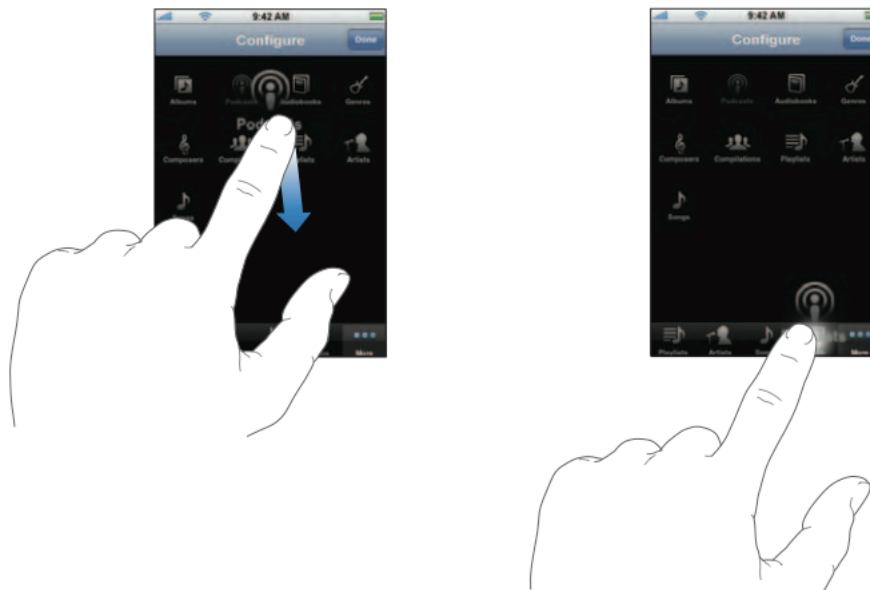
3. Set the appearance of the more list.

If you add more items than can be displayed on a tab bar controller, a More item automatically appears as the last item on the tab bar, shown in the screenshots in Figure 4. When the user taps the More item, a more view appears, containing a list of the tab bar items that don't fit on the tab bar.

4. Allow the tab bar to be customized.

By default, the user is allowed to customize the tab bar by clicking an Edit button on the navigation bar when the more view is displayed—for example, add, remove, and rearrange items, as shown in Figure 4. If you do not want the user to customize the tab bar, set the `customizableViewControllers` tab bar controller property to `nil`. You can also set the `customizableViewControllers` property to a subset of your view controllers if you want to allow some customization of items but not others.

Figure 4 Customizing the tab bar



Creating a Tab Bar Controller

Next, create a tab bar controller, add view controllers to it, and then add its view to a window. You can create all or some of the user interface objects programmatically by overriding the `applicationDidFinishLaunching:` application delegate method. Follow these steps to do this:

1. Create a tab bar controller.

Simply use `alloc` and `initWithNibName:bundle:` to create the tab bar controller. A `UITabBar` object is automatically positioned at the bottom of the screen.

```
tabBarController = [[UITabBarController alloc] initWithNibName:nil bundle:nil];
```

2. Create and add view controllers to the tab bar controller.

```
// Create three different custom view controllers
CustomViewController1 *viewController1 = [[[CustomViewController1 alloc]
initWithNibName:nil bundle:nil] autorelease];
CustomViewController2 *viewController2 = [[[CustomViewController2 alloc]
initWithNibName:nil bundle:nil] autorelease];
CustomViewController3 *viewController3 = [[[CustomViewController3 alloc]
initWithNibName:nil bundle:nil] autorelease];
tabBarController.viewControllers =
    [NSArray arrayWithObjects:viewController1, viewController2, viewController3,
    nil];
```

3. Create a window and add the tab bar controller's view to the window.

```
// Create a window
window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen].bounds];
[window addSubview:tabBarController.view];
```

A tab bar controller's view contains a `UITabBar` object and its selected view controller's view in its view hierarchy. You do not need to access the `UITabBar` object directly.

4. Display the window with the tab bar

The tab bar and your views are not visible until the window appears on the screen.

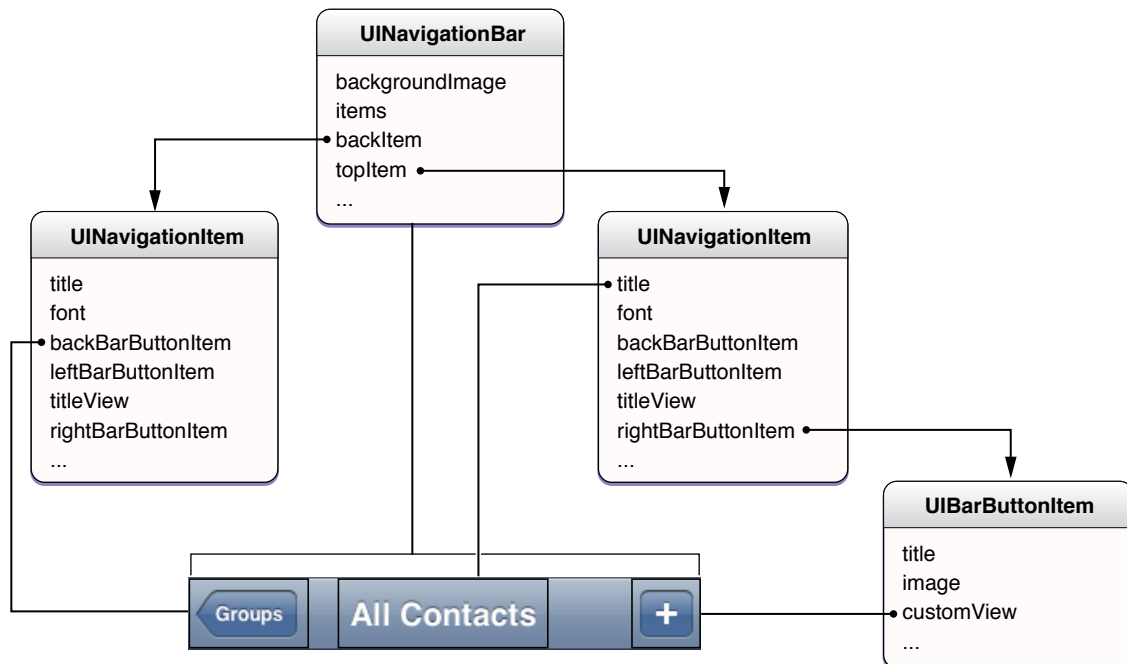
```
[window makeKeyAndVisible];
```

Using Navigation Controllers

Use navigation controllers, instances of `UINavigationController`, to implement a navigation interface where the user navigates a hierarchy of objects. This class hides the details of managing a navigation bar and its navigation items.

A navigation bar manages navigation items that are popped and pushed on a stack that it manages. Each navigation item encapsulates the state of the navigation bar when that item is the top item as illustrated in Figure 1. For instance, the navigation item encapsulates the title of the item displayed in the center of the navigation bar and whether a back button, labeled with the title of the previous item, is automatically displayed. In fact, a navigation item allows custom buttons to be placed anywhere on the navigation bar—you don't have to use a navigation bar for strictly hierarchical navigation. This design provides maximum flexibility for programmers. However, because most applications do want to navigate hierarchically and update the view below the navigation bar depending on the top item, more infrastructure is needed.

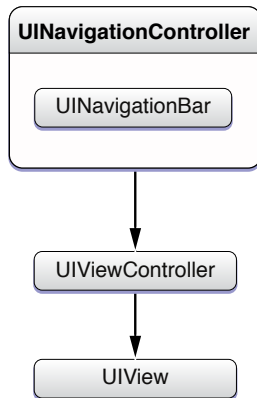
Figure 1 Navigation item components



Navigation controllers are designed to make the task of updating the navigation bar and swapping the view below easier. A `UINavigationController` object manages a stack of view controllers much as a navigation bar manages a stack of navigation items. A navigation controller is required to have at least one view controller on its stack, as illustrated by the object diagram in Figure 2. A

`UIViewController` object encapsulates properties for setting up its navigation item and the view to display when that view controller is the top view controller. You can access the navigation item object (not shown in the diagram) directly to set its properties.

Figure 2 Single view controller on the stack



Typically, a navigation controller has multiple view controllers on its stack, as illustrated in the object diagram in Figure 3. When the top view controller changes, the navigation bar is updated and the top view controller's view is displayed below it. You can even push multiple view controllers on the stack before displaying a window to set the user interface in a desired state. For example, when the user taps the Contacts item on the tab bar in the Phone application, the navigation controller already has two view controllers on its stack: a "Groups" and an "All Contacts" view controller, where "All Contacts" is the top view controller.

Figure 3 Multiple view controllers on the stack

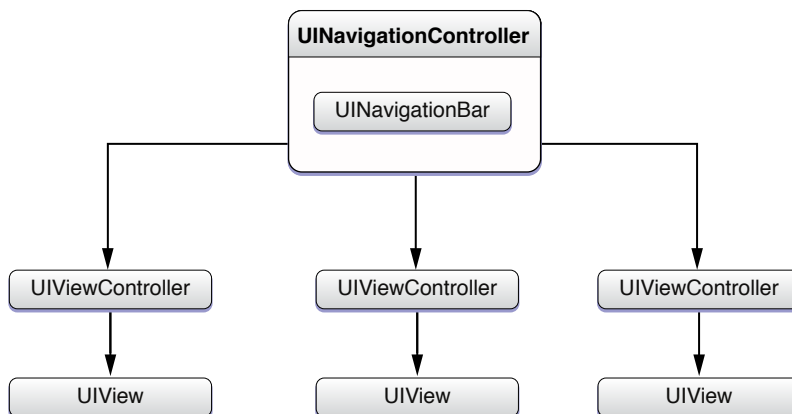
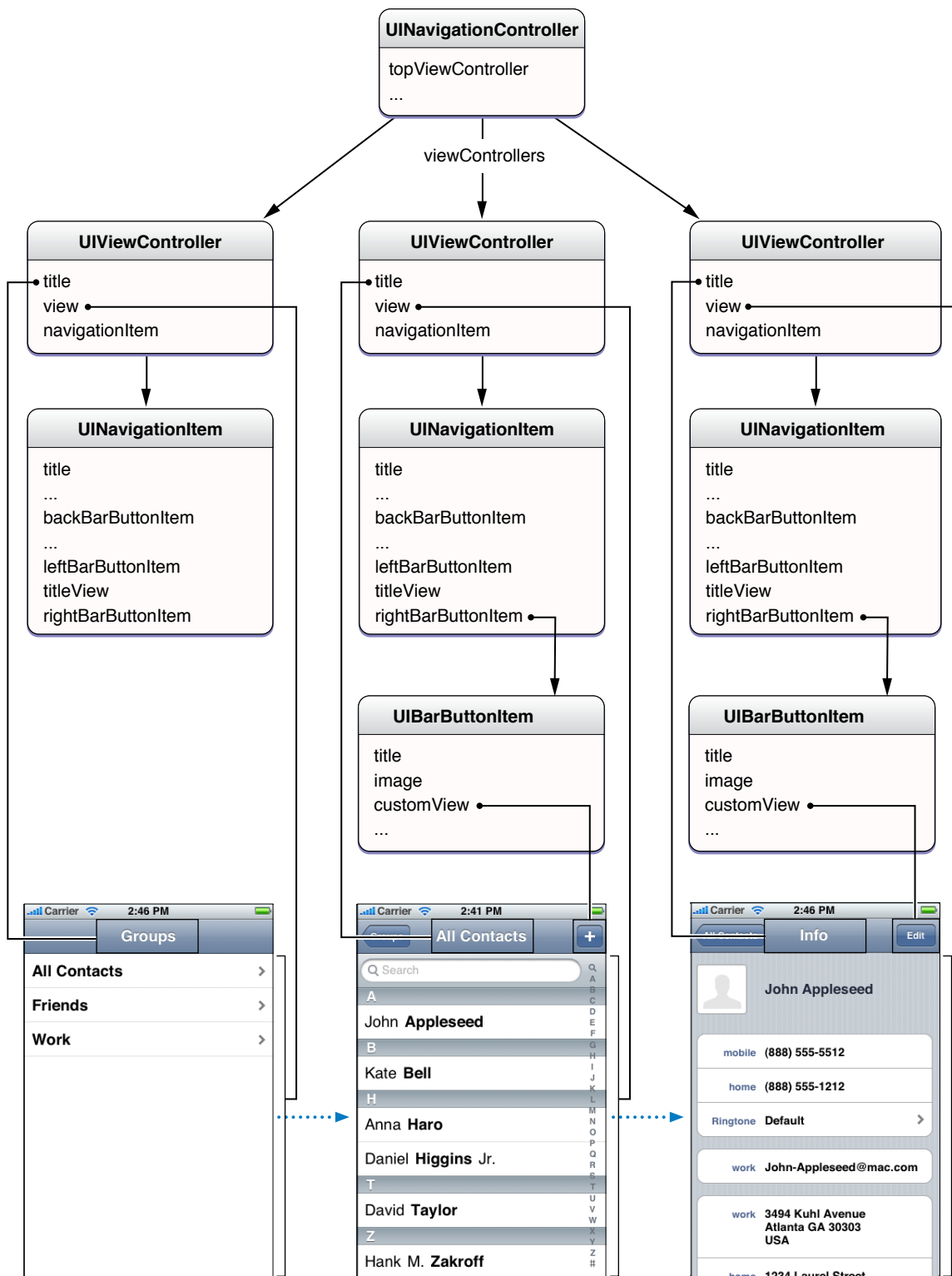


Figure 4 shows the screenshots in sequence when each view controller is the top view controller. For example, the first screen shows when the Groups view controller is the top view controller. The Groups `title` property is displayed in the middle of the navigation bar, and the contents of Groups is listed in a table view below. The second screen shows what results when the user taps All Contacts

from the list, and the third screen shows what results when the user selects a person from the list of people in the All Contacts group. Each time the user drills down the hierarchy, a new view controller is pushed onto the stack and the display is updated accordingly. If the user taps a back button—such as the All Contacts button in the third screen—the top view controller is popped from the stack.

Figure 4 Navigation controller components

Even though the screen changes as the user navigates, the underlying object graph is still present, as shown in the object diagram in the upper portion of [Figure 4](#) (page 30). The navigation controller manages a collection of view controller objects. When the view controller on the right is the top view controller, the other view controllers are next on the stack. Each view controller contains a navigation item, a `UINavigationControllerItem` object, that encapsulates presentation information when that view controller is the top view controller. Each view controller is also responsible for creating its own custom view displayed below the navigation bar.

How a navigation bar is updated is actually more complex because navigation items support custom buttons and views in all locations on the navigation bar. For example, the plus button on the navigation bar in the second screen shot is a custom button. You can use the `UIBarButtonItem` class to create custom bar button items specifically for a navigation bar. Read [“Creating Custom Buttons and Views for the Navigation Item”](#) (page 33) to learn how to fully customize a navigation bar.

The first step is to subclass `UIViewController`, typically, for each item that is pushed onto the stack. Next you create a navigation controller and add your view controllers to it. Optionally, you can create custom buttons and views for the navigation item that are displayed instead of the default views when the view controller is the top view controller. Many Apple applications take advantage of this flexible design by customizing the navigation bar.

Creating a View Controller Subclass for a Navigation Controller

To use navigation controllers, you must first create your custom view controllers—one for each application view you plan to display. You do this by subclassing `UIViewController` because all controller objects inherit from this superclass. Into these subclasses go all your custom views and logic. For a view controller to be managed by a navigation controller, certain properties need to be set, too.

1. Create the view.

As described in [“Using View Controllers”](#) (page 19), you can create a view programmatically by setting the `view` property directly or by implementing the `loadView` method. You can also load a view from a nib file using the `initWithNibName:bundle:` method. Read [“Using View Controllers”](#) (page 19) for the methods you should implement in your view controller subclass whether you decide to use Interface Builder or not.

The view you create is automatically resized and displayed when the view controller becomes the top view controller. If you implement the `loadView` method or specify a nib file name, the view is not created until the controller is the top view controller. The view is placed between the navigation bar and the bottom bar (if present); otherwise, it is placed just below the navigation bar. For this reason, you should set the autoresize mask of any subviews and not change the view’s frame later.

2. Set the view controller and navigation item properties.

You just need to set the `title` property of a view controller for it to be managed by a navigation controller. Listing 1 shows how to set the properties programmatically by overriding the `initWithNibName:bundle:` designated initializer.

Listing 1 Initializing a view controller managed by a navigation controller

```

- (id)initWithNibName:(NSString *)nibName bundle:(NSBundle *)nibBundle
{
    self = [super initWithNibName:nibName bundle:nibBundle];
    if (self) {
        self.title = @"Yellow";
        // Insert other initialization code here
    }
    return self;
}

```

By default, the navigation controller uses the view controller’s title as the back button when the view controller is the next view controller on the stack. Set the properties of the enclosing `UINavigationController` object to specify other values. Use the `navigationItem` property to get the `UINavigationController` object. [Figure 1](#) (page 27) illustrates the relationship between the navigation item and its navigation bar object, as well as the views its properties affect. Read [“Creating Custom Buttons and Views for the Navigation Item”](#) (page 33) to learn how to add custom views to the navigation item and specify alternate titles and buttons instead of the default back button.

Creating a Navigation Controller

Create a navigation controller, add view controllers to it, and then add the navigation controller’s view to the view hierarchy—for example, by adding its view to a window. You can create a navigation controller using Interface Builder but typically you push and pop view controllers programmatically based on some user action—for example, when the user selects a cell in a table.

To create the navigation controller and its view controllers programmatically, implement the `applicationDidFinishLaunching:` application delegate method as follows.

1. Create a navigation controller with a root controller.

Use the `initWithRootViewController:` designated initializer to create a navigation controller with a single view controller.

```

GroupsController *groupsController = [[GroupsController alloc] initWithNibName:nil
bundle:nil] autorelease];
UINavigationController *navigationController =
    [[UINavigationController alloc] initWithRootViewController:groupsController];

```

2. Optionally, push other view controllers onto the navigation controller.

```

ContactsController *contactsController = [[ContactsController alloc]
initWithNibName:nil bundle:nil] autorelease];
[navigationController pushViewController:contactsController animated:NO];

```

You do not need to animate the navigation bar when setting up the navigation controller the first time in the `applicationDidFinishLaunching:` application delegate method, but you typically animate it later when a user action pushes or pops an item from the navigation bar. You animate the changes by passing `YES` as the `animated:` argument in the `pushViewController:animated:` or `popViewControllerAnimated:` methods.

3. Add the navigation controller's view to a window.

```
// Create a window
window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen].bounds];
[window addSubview:[navigationController view]];
```

A navigation controller's view contains a `UINavigationController` object and the top view controller's view in its view hierarchy.

4. Display the window with the navigation bar.

The navigation bar and your views are not visible until the window appears on the screen.

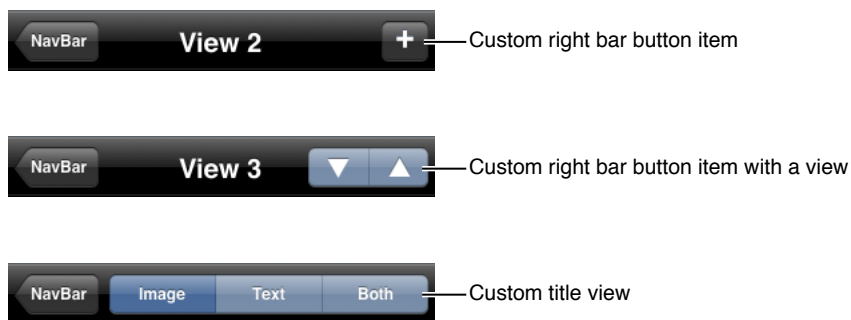
```
[window makeKeyAndVisible];
```

It's the responsibility of your application to provide buttons in the view to navigate down a hierarchy. Read [“Using Table Views for Hierarchical Data”](#) (page 43) to learn how to use a table view to represent children of a parent object using a table view.

Creating Custom Buttons and Views for the Navigation Item

In the simplest case, a back button is displayed on the left side, the top view controller's title is displayed in the middle, and nothing is displayed on the right side of the navigation bar. However, you can specify any custom button or view for all locations on the navigation bar. Many iPhone applications use custom views—for example, iPod uses custom views when you listen to a song. Figure 5 shows different ways to customize a navigation bar.

Figure 5 Custom navigation item views



You specify custom views by setting the properties of the enclosing `UINavigationController` object shown in [Figure 4](#) (page 30). You access the navigation item using the `navigationItem` property.

The rules for displaying the left side of the navigation bar are as follows:

- The next view controller (if it exists) is represented as a back button on the left side of the navigation bar unless another view is specified. The action of the back button pops the current view controller to make the next view controller the top view controller.

The next view controller's `title` property is used as the default button title. Set the `backBarButtonItem` property of the next view controller's `navigationItem` property to specify an alternate title or button.

- If a custom left view is specified, it is displayed instead of a back button.

Set the `leftBarButtonItem` property of the `navigationItem` property to specify a custom left view. Use the `UIBarButtonItem` class to create a bar button item for a navigation item. Use the `initWithBarButtonSystemItem:target:action:` method to programmatically create common system buttons.

The rules for displaying the middle of the navigation bar are as follows:

- The view controller's title is displayed in the middle of the navigation bar unless it has a custom title view.

Set the view controller title using the `title` property or if different, set the navigation item's `title` property.

- If a custom title view is specified, it is displayed in the middle of the navigation bar instead of the title.

Set the `titleView` property of the `navigationItem` property to specify a custom title view. This can be any view that fits on a navigation bar—for example, a label, button, or control.

The rules for displaying the right side of the navigation bar are as follows:

- If a custom right view is specified, it is displayed on the right side of the navigation bar; otherwise, nothing is displayed on the right side.

Set the `rightBarButtonItem` property of the `navigationItem` property to specify a custom right view. Use the `UIBarButtonItem` class to programmatically create a bar button item for a navigation item. Use the `initWithBarButtonSystemItem:target:action:` method of `UIBarButtonItem` to create common system buttons.

For example, the `initWithNibName:bundle:color:` method, in Listing 2, is the designated initializer for a view controller subclass. The `initWithNibName:bundle:color:` method initializes the properties of the inherited `navigationItem` property. It first sets the prompt (a single line of text above the title) and then sets the right custom view to an instance of the `UIBarButtonItem` class. It's the responsibility of the application to provide the target and action for any buttons added to a navigation item. In Listing 2, the application implements the `showAnotherClicked:` method.

Listing 2 Setting navigation item properties

```
- (id)initWithNibName:(NSString *)nibName bundle:(NSBundle *)nibBundle
color:(UIColor *)color
{
    self = [super initWithNibName:nil bundle:nil];
    if (self) {
        _color = [color copy];

        self.navigationItem.prompt = @"Just a color...";

        UIBarButtonItem *anotherButton = [[UIBarButtonItem alloc]
initWithTitle:@"Show" style:UIBarButtonItemStylePlain
        target:self action:@selector(showAnotherClicked)];
        self.navigationItem.rightBarButtonItem = anotherButton;
    }
}
```

```

    }
    return self;
}

```

Note: The custom views you create for a navigation item should be normal views, controls, or buttons that fit on a navigation bar. For example, do not add action sheets, alert views, toolbars, tab bars, or search bars to a navigation bar.

Using Edit and Done Buttons

Navigation controllers also support editable views that toggle between an editable and noneditable state—for example, tap the Edit button on the right side of the navigation bar to edit the contents of the view and the Done button to save the changes. The button and the view are the same objects in each state. The button and view simply redisplay when the edit/done state changes.

Follow these steps to use an Edit/Done button:

1. Set the custom left or right bar button item of the navigation item to the button returned by the `editButtonItem` method and then set the initial state of your view as follows:

```

myViewController.navigationItem.rightBarButtonItem = [myViewController
editButtonItem];
myViewController.editing = NO; // Initially displays an Edit button and
noneditable view

```

The Edit/Done button is a single button object that displays "Edit" if `editing` is `NO` or "Done" if `editing` is `YES`. The default button action sends `setEditing:animated:` to the view controller.

2. Override the `setEditing:animated:` method in the view controller subclass to switch the state of the view controller's view depending on the state.

If the `setEditing:` parameter is `YES`, the view should display editable controls; otherwise, it should display noneditable controls. Subclass implementations of this method should first invoke the superclass implementation as follows:

```

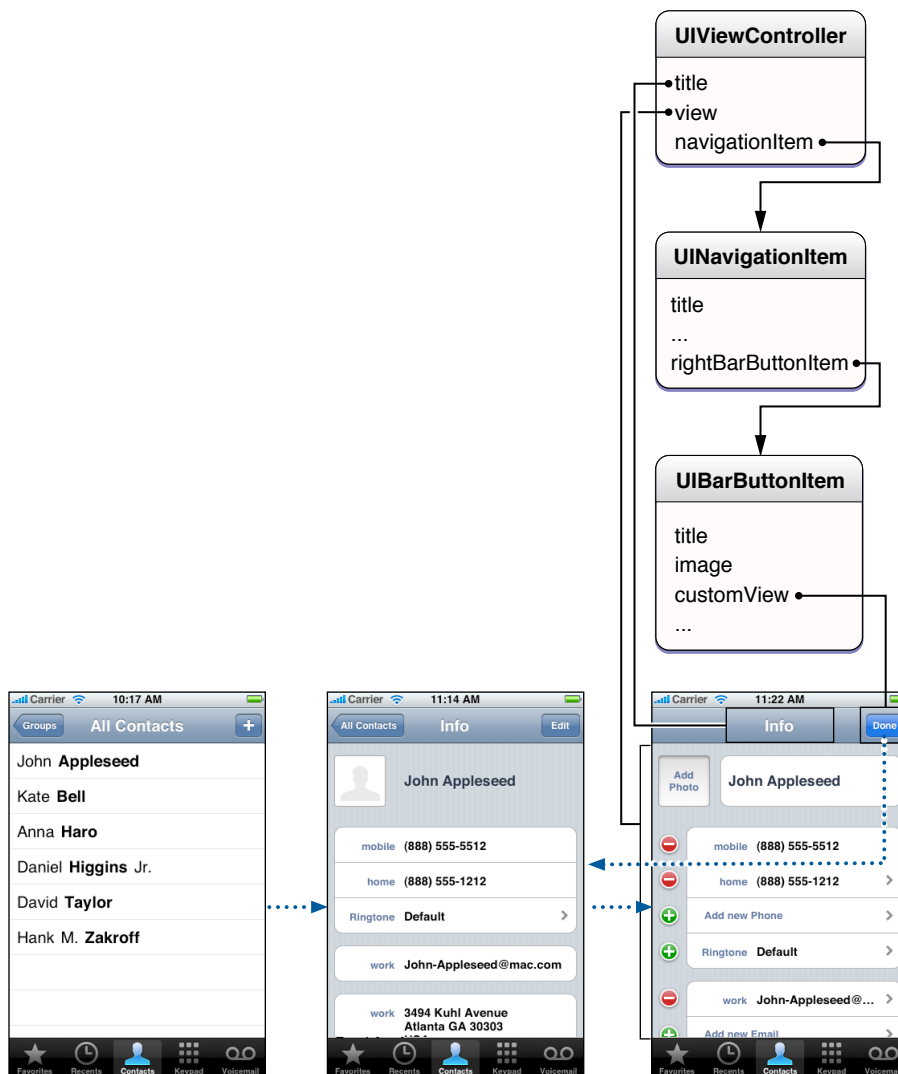
- (void)setEditing:(BOOL)flag animated:(BOOL)animated
{
    [super setEditing:flag animated:animated];
    if (flag == YES){
        // change view to an editable view
    }
    else {
        // save the changes if needed and change view to noneditable
    }
}

```

Note that subclasses of `UITableViewController` already support editing. The `UITableViewController` class overrides the `setEditing:animated:` method to toggle the editing state of the table view. Read [“Using Table Views for Hierarchical Data”](#) (page 43) for how to subclass `UITableViewController`.

Figure 6 illustrates the series of screens that are displayed when the user edits a person's contact information. In the first screenshot, when the user selects a person, the person view controller is pushed onto the stack and an Edit button appears on the right side of the navigation bar. When the user taps the Edit button, the view below changes, allowing the user to edit the data. When the user taps the Done button, the changes are saved, the view changes to noneditable controls, and the Edit button appears again.

Figure 6 Using an Edit/Done button



Read [“Using Table Views for Hierarchical Data”](#) (page 43) for how a table view controller behaves when editing is enabled.

Using Modal View Controllers

Modal view controllers provide another way to present multiple screens of information to the user. A modal view controller is simply a view controller with a view that is presented on top of another view and requires some action by the user to be dismissed. When the view is presented, it is animated from below and placed on top of all other views, hiding everything but the status bar. Every view controller can have a modal view controller creating a stack of views, as shown in [Figure 3](#) (page 13). You can even stack a navigation controller on top of a regular view controller. It is the responsibility of the application to provide buttons on the modal view to dismiss it.

Figure 1 illustrates a navigation controller with multiple modal view controllers on its top view controller. Upon display, navigation views animate from left to right and modal views animate from bottom to top. Therefore it's helpful to visualize the navigation views moving horizontally from left to right and modal views stacked vertically, as illustrated in Figure 1. In this example, the user might be drilling down to edit a particular field or make a selection—each modal view presents more detailed information. Dismissing a modal view causes all the modal view controllers on top of it to be dismissed as well.

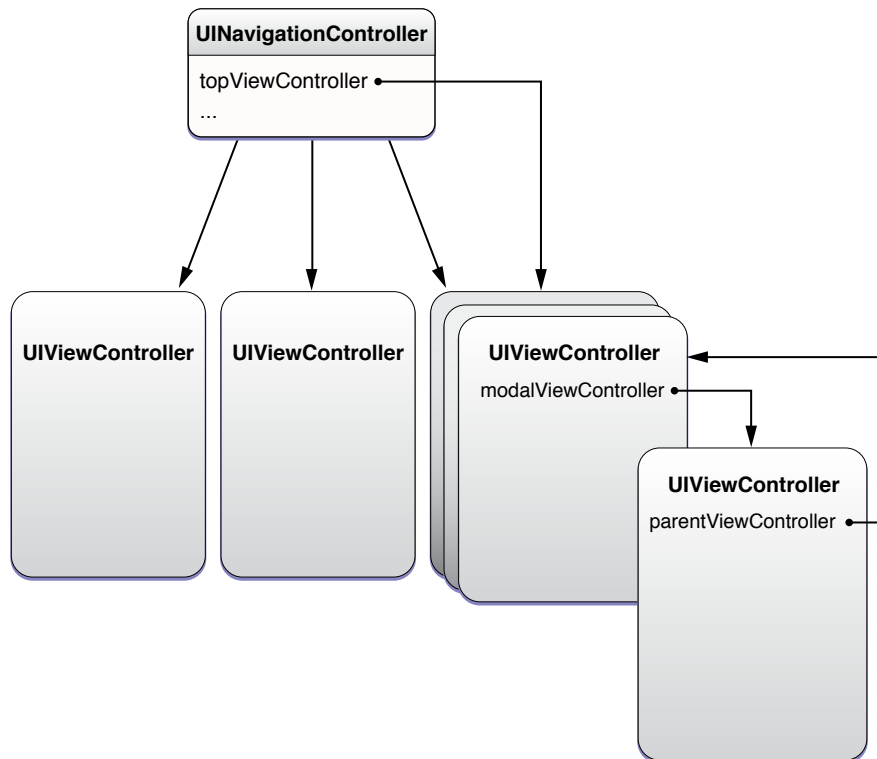
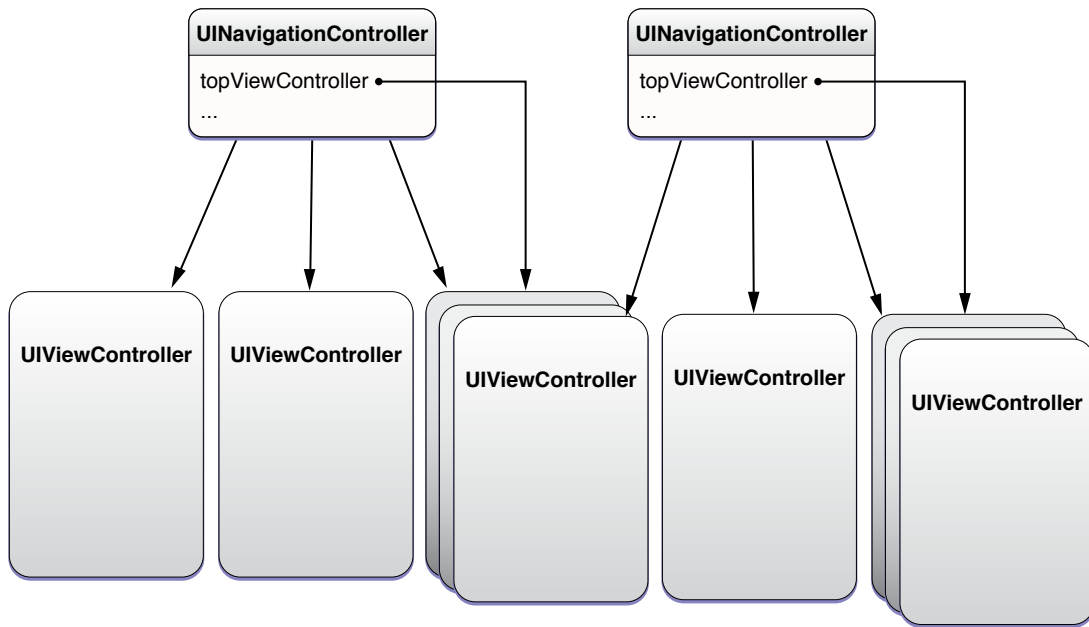
Figure 1 Modal view controllers

Figure 2 illustrates how you might combine multiple navigation controllers with modal view controllers to create levels of navigation in your user interface. In this example, a second navigation controller has a modal view controller, shown in the center of the figure. If you combine these types of controllers, you need to provide a custom button on the navigation bar to dismiss the navigation controller.

Figure 2 Navigation controllers with modal view controllers

The root navigation controller on the right in Figure 2 could also be added to an underlying tab bar controller as described in [“Combining Tab Bar and Navigation Controllers”](#) (page 41).

In addition, you can use modal views without tab bars and navigation bars. You can attach a view controller’s view directly to a window and then add a modal view controller to the view controller. The view for each modal view controller needs to provide a mechanism—typically, a button—to dismiss the modal view.

Use the `presentModalViewController:animated:` method to present a modal view controller on top of another view controller. Each view controller has a `modalViewController` and a `parentViewController` property so that you can traverse the stack of modal view controllers. Use the `dismissModalViewControllerAnimated:` method to remove a modal view controller. This method removes its view from the view hierarchy revealing the view in back. For example, a button that dismisses a modal view controller should invoke the `dismissModalViewControllerAnimated:` method. If you send the `dismissModalViewControllerAnimated:` message to a view controller that is not on the top view controller, it dismisses itself and all view controllers on top of it. Typically, you do this to pop back to an underlying navigation controller.

Follow these steps to present one navigation controller on top of another navigation controller.

1. Create a regular view controller.

```
MyViewController *modalViewController = [[[MyModalViewController alloc]
initWithNibName:nil bundle:nil] autorelease];
```

2. Create a navigation controller containing the view controller.

```
UINavigationController *secondNavigationController =
    [[UINavigationController alloc]
    initWithRootViewController:modalViewController];
```

3. Present the navigation controller as a modal view controller on top of an existing navigation controller.

```
[[self firstNavigationController]  
presentModalViewController:secondNavigationController animated:YES];
```

The second navigation controller's view—the view of its top view controller—animates from the bottom to the top of the screen hiding everything but the status bar. The navigation bar belonging to the existing navigation controller is replaced by a new navigation bar belonging to the second navigation controller.

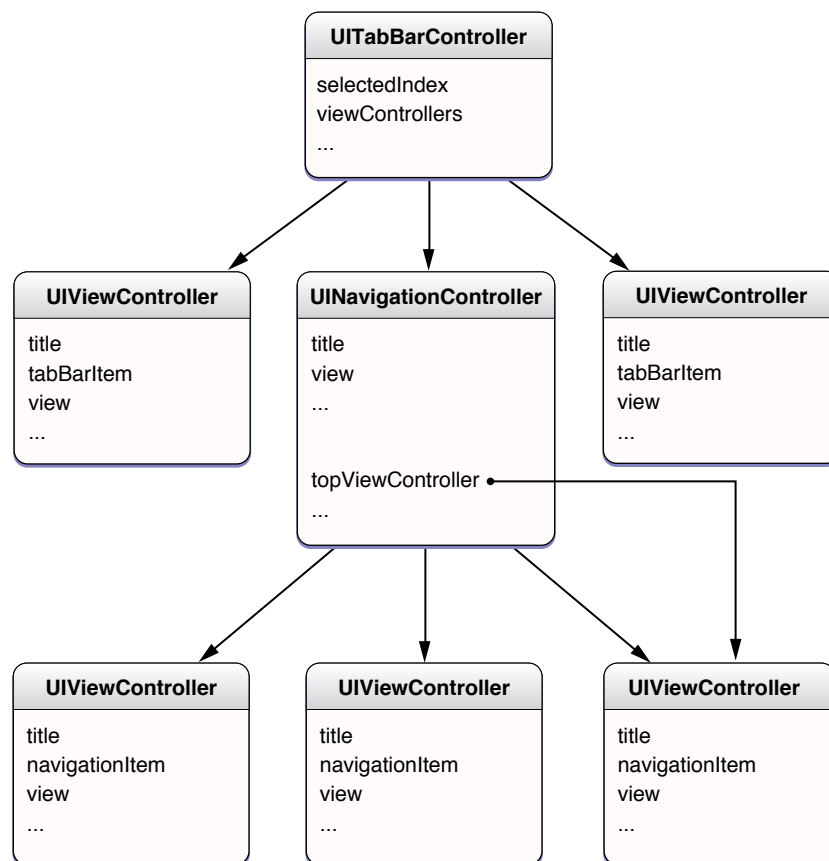
4. Provide a dismiss button.

The top view controller of the second navigation controller needs to provide a dismiss button. Typically, you do this by setting the right custom view of its navigation item to a bar button item object with the action set to the `dismissModalViewControllerAnimated:` method and the target set to the first navigation controller. When this button is tapped, the second navigation controller disappears.

Combining Tab Bar and Navigation Controllers

It's very common to combine tab bar and navigation controllers, as illustrated in [Figure 4](#) (page 14). To do this, you simply add navigation controllers to a tab bar controller (however, you should never add a tab bar controller to a navigation controller). It's OK to have multiple navigation controllers—hence, multiple navigation bars—that maintain the navigation state for different modes in your application. Figure 1 shows the resulting object graph when you combine tab bar and navigation controllers.

Figure 1 Tab bar controller containing a navigation controller



For example, the Phone application uses a tab bar controller containing multiple navigation controllers. The contacts mode uses a navigation controller as illustrated in [Figure 6](#) (page 36). The favorites, recents, and voicemail modes also use navigation controllers although some of the navigation items use custom buttons.

Follow the same steps described in [“Creating a Tab Bar Controller”](#) (page 25) and [“Creating a Navigation Controller”](#) (page 32) to add a navigation controller to a tab bar controller. You are not required to subclass `UINavigationController` to add a navigation controller to a tab bar controller. By default, the top view controller’s `title` and `tabBarItem.image` properties are used to represent the navigation controller on the tab bar. Optionally, you can also set the `title` and `tabBarItem.image` properties of the navigation controller itself.

Since a navigation controller’s view contains both the navigation bar and its top view controller’s view in its view hierarchy, the tab bar simply displays the navigation controller’s view when it is the selected view controller. When you add the tab bar controller’s view to a window, the following view hierarchy is created:

1. The window contains the tab bar controller’s view.
2. The tab bar controller’s view contains the tab bar and its selected view controller’s view.
3. If the selected view controller is a navigation controller, its view contains a navigation bar and its top view controller’s view.
4. If any of the above view controllers have a modal view controller, then its view is inserted in the hierarchy instead of the view controller’s view.

You don’t need to implement a full-blown navigation interface when using navigation controllers. It’s very common to wrap a single view controller in a navigation controller and add it to a tab bar controller. You might do this just to display a title and custom buttons above your view.

You can optionally hide the bottom bar (either a toolbar or tab bar) when your view controller is pushed by setting the `hidesBottomBarWhenPushed` view controller property to `YES`.

Using Table Views for Hierarchical Data

It's very common to use table views to display the children of parent objects when using navigation controllers, as shown in the Contacts screenshots in [Figure 4](#) (page 30). The first and second screenshots show groups and people displayed using table views.

The `UITableViewController` class makes it easy to add a table view to your user interface. It takes care of creating a table view and even handles editing. The table view controller is the delegate and data source for its table view. Follow these steps to programmatically create a table view controller:

1. Subclass `UITableViewController` and implement the data source methods.

Since the table view controller is the data source for the table view, it needs to implement some `UITableViewDataSource` protocol methods that specify the table dimensions.

Implement the `numberOfSectionsInTableView:` method to return the number of sections (1 if you have no sections), and the `tableView:numberOfRowsInSection:` method to return the number of rows per section (the total number of rows if you have no sections). For example, this implementation returns the number of objects in an array.

```
- (NSInteger)tableView:(UITableView *)table
numberOfRowsInSection:(NSInteger)section
{
    return arrayOfStrings.count;
}
```

2. Implement the `tableView:cellForRowAtIndexPath:` data source method to return a cell per row in the table.

For example, the following implementation returns a simple text cell where `arrayOfStrings` is an array of `NSString` objects:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [[UITableViewCell alloc] initWithFrame:CGRectZero
reuseIdentifier:nil];
    cell.autoresizingMask = UIViewAutoresizingFlexibleWidth;
    cell.text = [arrayOfStrings objectAtIndex:indexPath.row];

    return [cell autorelease];
}
```

3. If you are using a table view controller for hierarchical navigation, then implement the `tableView:didSelectRowAtIndexPath:` delegate method to push another view controller on the stack when a row is selected as follows:

```
- (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) {
        MyViewController *viewController = [[[MyViewController alloc] initWithNibName:nil bundle:nil];
        viewController.title = [anArrayOfStrings objectAtIndex:indexPath.row];
        [[self navigationController] pushViewController:viewController animated:YES];
    }
}
```

4. Optionally, display an accessory disclosure at the end of each row to indicate that there are child objects by implementing the `tableView:accessoryTypeForRowAtIndexPath:` delegate method as follows:

```
- (UITableViewCellAccessoryType)tableView:(UITableView *)tableView accessoryTypeForRowAtIndexPath:(NSIndexPath *)indexPath
{
    return UITableViewCellAccessoryDisclosureIndicator;
}
```

You can also set the accessory type of a cell when creating the cell in the `tableView:cellForRowAtIndexPath:` method.

5. If you allow the user to edit the contents of a table view, then add an edit button to the navigation item as follows:

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

Typically, you configure the navigation item in the view controller's `initWithNibName:bundle:` method. Then implement the `tableView:commitEditingStyle:forRowAtIndexPath:` delegate method to take some action when a cell is added, modified, or deleted.

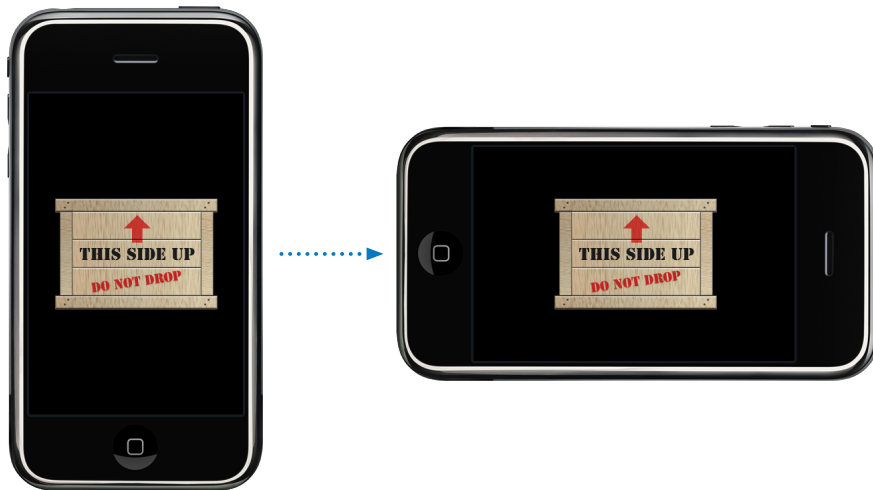
Use the `initWithStyle:` method to create an instance of your `UITableViewController` subclass which allows you to specify the type of table view you want.

Read *Table View Programming Guide for iPhone OS* for more details on implementing these and other protocol methods.

Autorotating Views

View controllers also support autorotation when the orientation changes—for example, when the user changes from portrait to landscape orientation, the toolbar, tab bar, navigation bar, and view also rotates, as shown in Figure 1. However, this feature can be used without navigation bars and tab bars. In fact, some applications might use a single view controller in their application just to take advantage of autorotation.

Figure 1 Autorotating view controllers



To autorotate when the orientation changes, subclasses of the `UIViewController` class simply override `shouldAutorotateToInterfaceOrientation:` to return `YES`, as shown in Listing 1. By default, views display only in portrait orientation, so you need to implement this method if you want to support other orientations.

Listing 1 Subclasses enabling autorotation

```
-  
(BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation  
{  
    return YES;  
}
```

By default, the navigation bar and tab bar are the header and footer views that slide out before the window rotates and that slide in after the window rotates. This occurs only if a tab bar or navigation bar is present. If a view controller is used without these controls—for example, if its view is directly attached to a window—then a view controller subclass can optionally override the `rotatingFooterView` and `rotatingHeaderView` methods to return application specific views to slide in and out.

If you set the `autoresizeMask` as described in "Autoresizing Views" in *iPhone OS Programming Guide*, the view displays correctly when it autorotates. However, if you want to customize the rotation by either changing the layout or adding additional animations, you need to override one or more of the rotation methods. These methods are sent to the view controller in this order while the interface rotates:

1. `willRotateToInterfaceOrientation:duration:`

Sent to the view controller before autorotation begins.

For example, you might implement this method to disable views, stop media playback, or temporarily turn off expensive drawing or live updates. You may also implement this method to swap the view if it should be different for landscape as compared with portrait orientation.

2. `willAnimateFirstHalfOfRotationToInterfaceOrientation:duration:`

Sent to the view controller just before the animation begins.

When this method is invoked, the `interfaceOrientation` property is still set to the old orientation. The new orientation is passed as an argument to this method. This method is invoked within an animation block that begins sliding the header and footer views out.

For example, you might implement this method to adjust the zoom level, scroller position, or other attribute of your view.

3. `willAnimateSecondHalfOfRotationFromInterfaceOrientation:duration:`

Sent to the view controller just before the second half of the animation begins.

When this method is invoked, the `interfaceOrientation` property is set to the new orientation. The old orientation is passed as an argument to this method. This method is invoked within an animation block that begins sliding the header and footer views in.

For example, you might implement this method to adjust the zoom level, scroller position, or other attribute of your view.

4. `didRotateFromInterfaceOrientation:`

Sent to the view controller after autorotation ends.

When this method is invoked, the `interfaceOrientation` property is set to the new orientation. The old orientation is passed as an argument to this method.

For example, you might implement this method to resume any activities you turned off in the `willRotateToInterfaceOrientation:duration:method`.



Warning: When a view rotates from portrait to landscape orientation, its `transform` property is set. If the `transform` property is not the identity transform, the value of the view's `frame` property is undefined. Therefore, use the view's `center` and `bounds` properties, not the `frame` property, to get its position and dimensions during and after rotation.

Document Revision History

This table describes the changes to *View Controller Programming Guide for iPhone OS*.

Date	Notes
2008-06-23	New document that explains how to use view controllers to implement radio, navigation, and modal interfaces.

REVISION HISTORY

Document Revision History