# View Controller Programming Guide for iOS

Developer

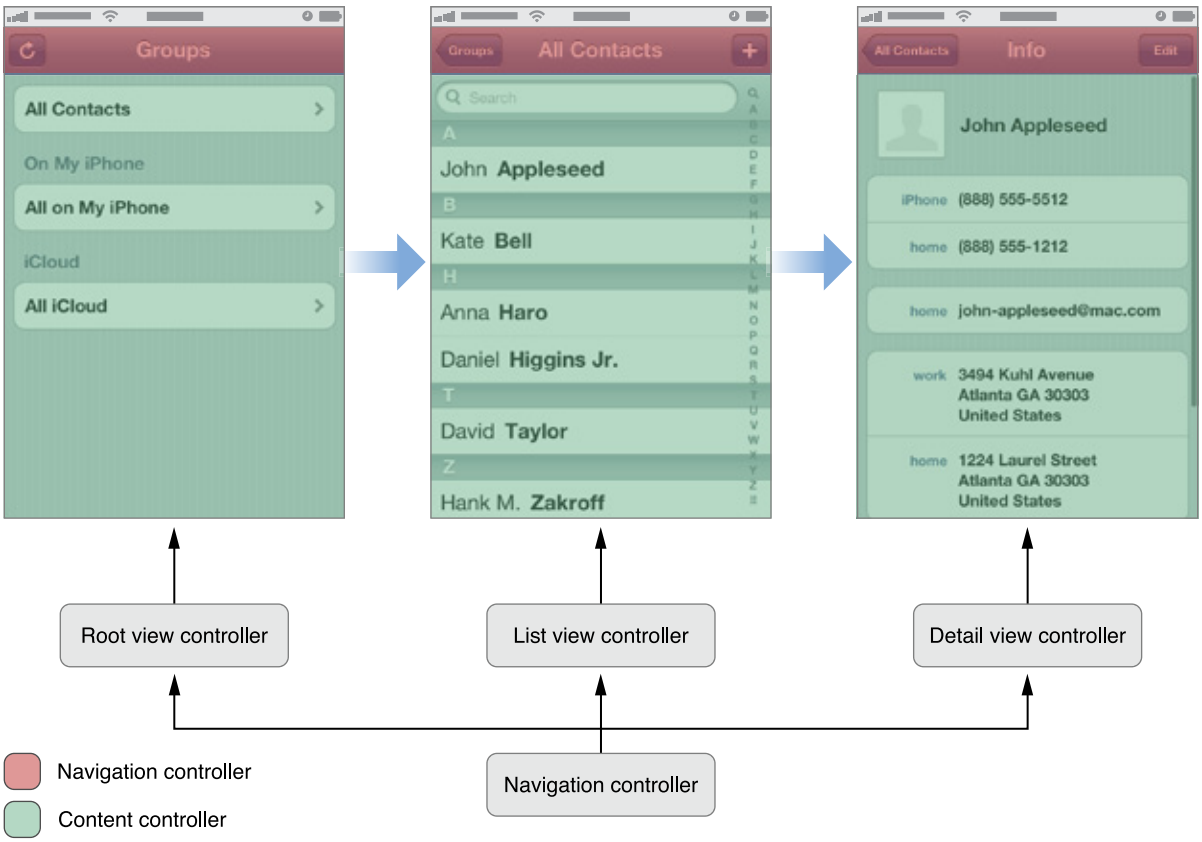
# Contents

# Figures, Tables, and Listings

# About View Controllers

View controllers are a vital link between an app's data and its visual appearance. Whenever an iOS app displays a user interface, the displayed content is managed by a view controller or a group of view controllers coordinating with each other. Therefore, view controllers provide the skeletal framework on which you build your apps.

iOS provides many built-in view controller classes to support standard user interface pieces, such as navigation and tab bars. As part of developing an app, you also implement one or more custom controllers to display the content specific to your app.



# At a Glance

View controllers are traditional controller objects in the Model-View-Controller (MVC) design pattern, but they also do much more. View controllers provide many behaviors common to all iOS apps. Often, these behaviors are built into the base class. For some behaviors, the base class provides part of the solution and your view controller subclass implements custom code to provide the rest. For example, when the user rotates the device, the standard implementation attempts to rotate the user interface; however, your subclass decides whether the user interface should be rotated, and, if so, how the configuration of its views should change in the new orientation. Thus, the combination of a structured base class and specific subclassing hooks make it easy for you to customize your app's behavior while conforming to the platform design guidelines.

## A View Controller Manages a Set of Views

A view controller manages a discrete portion of your app's user interface. Upon request, it provides a view that can be displayed or interacted with. Often, this view is the root view for a more complex hierarchy of views—buttons, switches, and other user interface elements with existing implementations in iOS. The view controller acts as the central coordinating agent for this view hierarchy, handling exchanges between the views and any relevant controller or data objects.

**Relevant chapter:** "View Controller Basics" (page 14)

## You Manage Your Content Using Content View Controllers

To present content that is specific to your app, you implement your own content view controllers. You create new view controller classes by subclassing either the `UIViewController` class or the `UITableViewController` class, implementing the methods necessary to present and control your content.

**Relevant chapter:** "Creating Custom Content View Controllers" (page 43)

## Container View Controllers Manage Other View Controllers

Container view controllers display content owned by other view controllers. These other view controllers are explicitly associated with the container, forming a parent-child relationship. The combination of container and content view controllers creates a hierarchy of view controller objects with a single root view controller.

Each type of container defines its own interface to manage its children. The container's methods sometimes define specific navigational relationships between the children. A container can also set specific restrictions on the types of view controllers that can be its children. It may also expect the view controllers that are its children to provide additional content used to configure the container.

iOS provides many built-in container view controller types you can use to organize your user interface.

**Relevant chapter:** "View Controller Basics" (page 14)

## Presenting a View Controller Temporarily Brings Its View Onscreen

Sometimes a view controller wants to display additional information to the user. Or perhaps it wants the user to provide additional information or perform a task. Screen space is limited on iOS devices; the device might not have enough room to display all the user interface elements at the same time. Instead, an iOS app temporarily displays another view for the user to interact with. The view is displayed only long enough for the user to finish the requested action.

To simplify the effort required to implement such interfaces, iOS allows a view controller to *present* another view controller's contents. When presented, the new view controller's views are displayed on a portion of the screen—often the entire screen. Later, when the user completes the task, the presented view controller tells the view controller that presented it that the task is complete. The presenting view controller then dismisses the view controller it presented, restoring the screen to its original state.

Presentation behavior must be included in a view controller's design in order for it to be presented by another view controller.

**Relevant chapter:** "Presenting View Controllers from Other View Controllers" (page 88)

## Storyboards Link User Interface Elements into an App Interface

A user interface design can be very complex. Each view controller references multiple views, gesture recognizers, and other user interface objects. In return, these objects maintain references to the view controller or execute specific pieces of code in response to actions the user takes. And view controllers rarely act in isolation. The collaboration between multiple view controllers also defines other relationships in your app. In short, creating a user interface means instantiating and configuring many objects and establishing the relationships between them, which can be time consuming and error prone.

Instead, use Interface Builder to create **storyboards**. A storyboard holds preconfigured instances of view controllers and their associated objects. Each object's attributes can be configured in Interface Builder, as can relationships between them.

At runtime, your app loads storyboards and uses them to drive your app's interface. When objects are loaded from the storyboard, they are restored to the state you configured in the storyboard. UIKit also provides methods you can override to customize behaviors that cannot be configured directly in Interface Builder.

By using storyboards, you can easily see how the objects in your app's user interface fit together. You also write less code to create and configure your app's user-interface objects.

**Relevant chapter:** "View Controller Basics" (page 14), "Using View Controllers in Your App" (page 35)

## How to Use This Document

Start by reading "View Controller Basics" (page 14), which explains how view controllers work to create your app's interface. Next, read "Using View Controllers in Your App" (page 35) to understand how to use view controllers, both those built into iOS and those you create yourself.

When you are ready to implement your app's custom controllers, read "Creating Custom Content View Controllers" (page 43) for an overview of the tasks your view controller performs, and then read the remaining chapters in this document to learn how to implement those behaviors.

## Prerequisites

Before reading this document, you should be familiar with the content in *Start Developing iOS Apps Today* and *Your Second iOS App: Storyboards*. The storyboard tutorial demonstrates many of the techniques described in this book, including the following Cocoa concepts:

- Defining new Objective-C classes
- The role of delegate objects in managing app behaviors
- The Model-View-Controller paradigm

## See Also

For more information about the standard container view controllers provided by UIKit, see *View Controller Catalog for iOS*.

For guidance on how to manipulate views in your view controller, see *View Programming Guide for iOS*.

For guidance on how to handle events in your view controller, see *Event Handling Guide for iOS*.

For more information about the overall structure of an iOS app, see *iOS App Programming Guide*.

For guidance on how to configure storyboards in your project, see *Xcode Overview*

# View Controller Basics

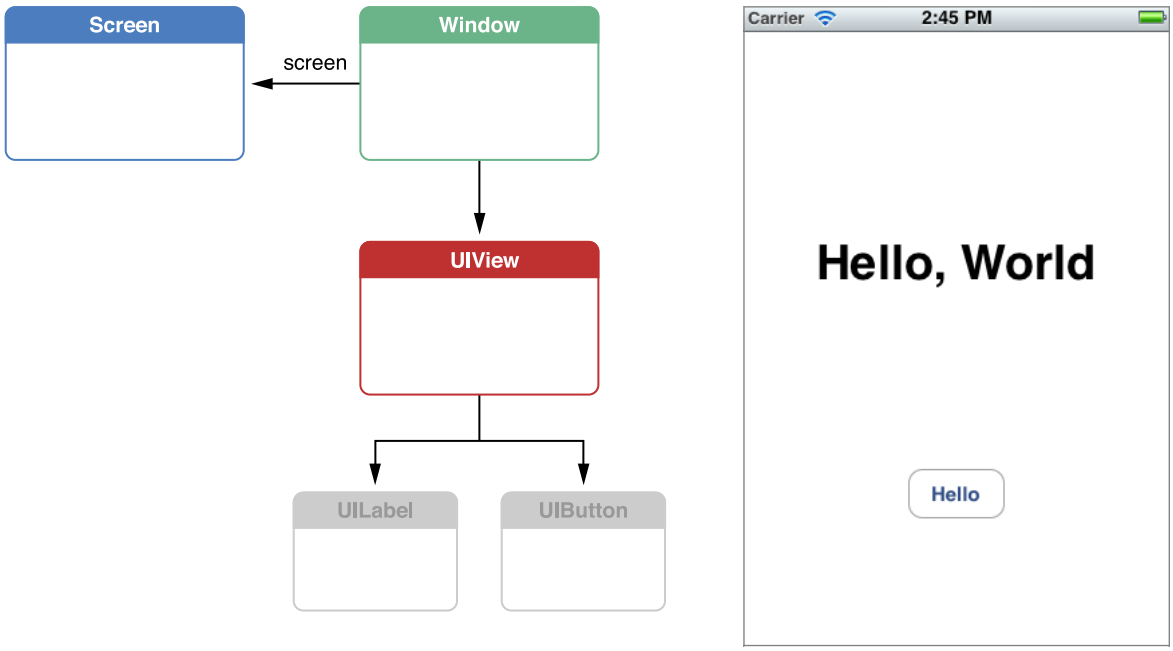
Apps running on iOS–based devices have a limited amount of screen space for displaying content and therefore must be creative in how they present information to the user. Apps that have lots of information to display must therefore only show a portion to start, and then show and hide additional content as the user interacts with the app. View controller objects provide the infrastructure for managing content and for coordinating the showing and hiding of it. By having different view controller classes control separate portions of your user interface, you break up the implementation of your user interface into smaller and more manageable units.

Before you can use view controllers in your app, you need a basic understanding of the major classes used to display content in an iOS app, including windows and views. A key part of any view controller's implementation is to manage the views used to display its content. However, managing views is not the only job view controllers perform. Most view controllers also communicate and coordinate with other view controllers when transitions occur. Because of the many connections view controllers manage, both looking inward to views and associated objects and looking outward to other controllers, understanding the connections between objects can sometimes be difficult. Instead, use Interface Builder to create storyboards. Storyboards make it easier to visualize the relationships in your app and greatly simplify the effort needed to initialize objects at runtime.

# Screens, Windows, and Views Create Visual Interfaces

Figure 1-1 shows a simple interface. On the left, you can see the objects that make up this interface and understand how they are connected to each other.

**Figure 1-1** A window with its target screen and content views

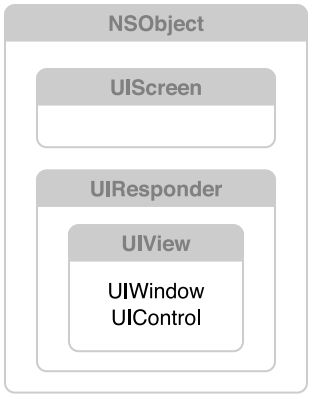



There are three major objects at work here:

- A `UIScreen` object that identifies a physical screen connected to the device.
- A `UIWindow` object that provides drawing support for the screen.
- A set of `UIView` objects to perform the drawing. These objects are attached to the window and draw their contents when the window asks them to.

Figure 1-2 shows how these classes (and related important classes) are defined in UIKit.

**Figure 1-2** Classes in the view system




Although you don't need to understand everything about views to understand view controllers, it can be helpful to consider the most salient features of views:

- A view represents a user interface element. Each view covers a specific area. Within that area, it displays contents or responds to user events.
- Views can be nested in a view hierarchy. Subviews are positioned and drawn relative to their superview. Thus, when the superview moves, its subviews move with it. This hierarchy makes it easy to assemble a group of related views by placing them in a common superview.
- Views can animate their property values. When a change to a property value is animated, the value gradually changes over a defined period of time until it reaches the new value. Changes to multiple properties across multiple views can be coordinated in a single animation.

  Animation is critically important to iOS app development. Because most apps display only a portion of their contents at one time, an animation allows the user to see when a transition occurred and where the new content came from. An instantaneous transition might confuse the user.
- Views rarely understand the role they play in your app. For example, Figure 1-1 shows a button (titled Hello), which is a special kind of view, known as a *control* . Controls know how to respond to user interaction in their area, but they don't know what they control. Instead, when a user interacts with a control, it sends messages to other objects in your app. This flexibility allows a single class (`UIButton`) to provide the implementation for multiple buttons, each configured to trigger a different action.

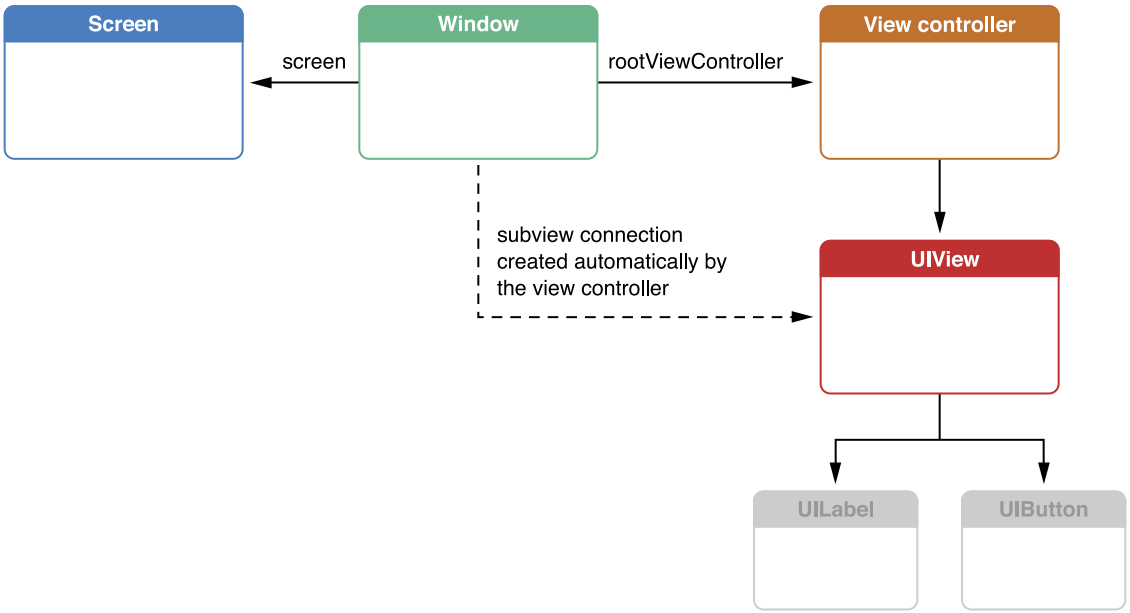
A complex app needs many views, often assembling them into view hierarchies. It needs to animate subsets of these views onto or off the screen to provide the illusion of a single larger interface. And finally, to keep view classes reusable, the view classes need to be ignorant of the specific role they perform in the app. So the app logic—the brains—needs to be placed somewhere else. Your view controllers are the brains that tie your app's views together.

# View Controllers Manage Views

Each view controller organizes and controls a view; this view is often the root view of a view hierarchy. View controllers are controller objects in the MVC pattern, but a view controller also has specific tasks iOS expects it to perform. These tasks are defined by the `UIViewController` class that all view controllers inherit from. All view controllers perform view and resource management tasks; other responsibilities depend on how the view controller is used.

Figure 1-3 shows the interface from Figure 1-1, but updated here to use a view controller. You never directly assign the views to the window. Instead, you assign a view controller to the window, and the view controller automatically adds its view to the window.

**Figure 1-3** A view controller attached to a window automatically adds its view as a subview of the window



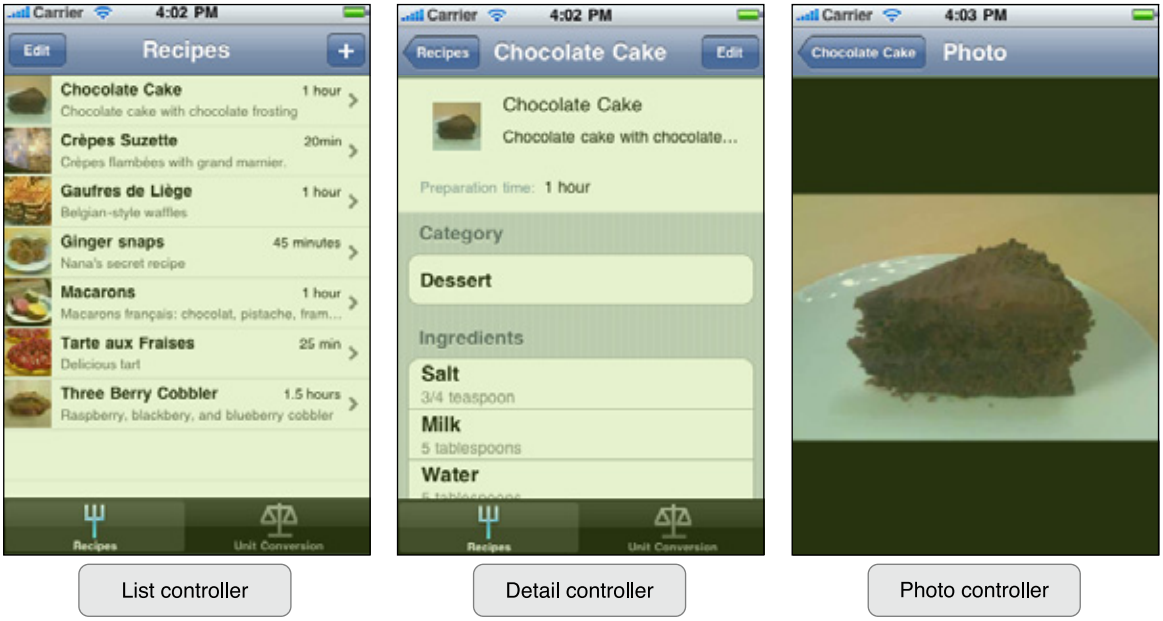

A view controller is careful to load its view only when the view is needed. It can also release the view under certain conditions. For these reasons, view controllers play a key part in managing resources in your app.

A view controller is the natural place to coordinate actions of its connected views. For example, when a button is pressed, it sends a message to the view controller. Although the view itself may be ignorant of the task it performs, the view controller is expected to understand what the button press means and how it should respond. The controller might update data objects, animate or change property values stored in its views, or even bring another view controller's contents to the screen.

Usually, each view controller instantiated by your app sees only a subset of your app's data. It knows how to display that particular set of data, without needing to know about other kinds of data. Thus, an app's data model, user interface design, and the view controllers you create are all influenced by each other.

Figure 1-4 shows an example of an app that manages recipes. This app displays three related but distinct views. The first view lists the recipes that the app manages. Tapping a recipe shows the second view, which describes the recipe. Tapping the recipe's picture in the detail view shows the third view, a larger version of the photo. Each view is managed by a distinct view controller object whose job is to present the appropriate view, populate the subviews with data, and respond to user interactions within the view hierarchy.

**Figure 1-4** Distinct views managed by separate view controllers



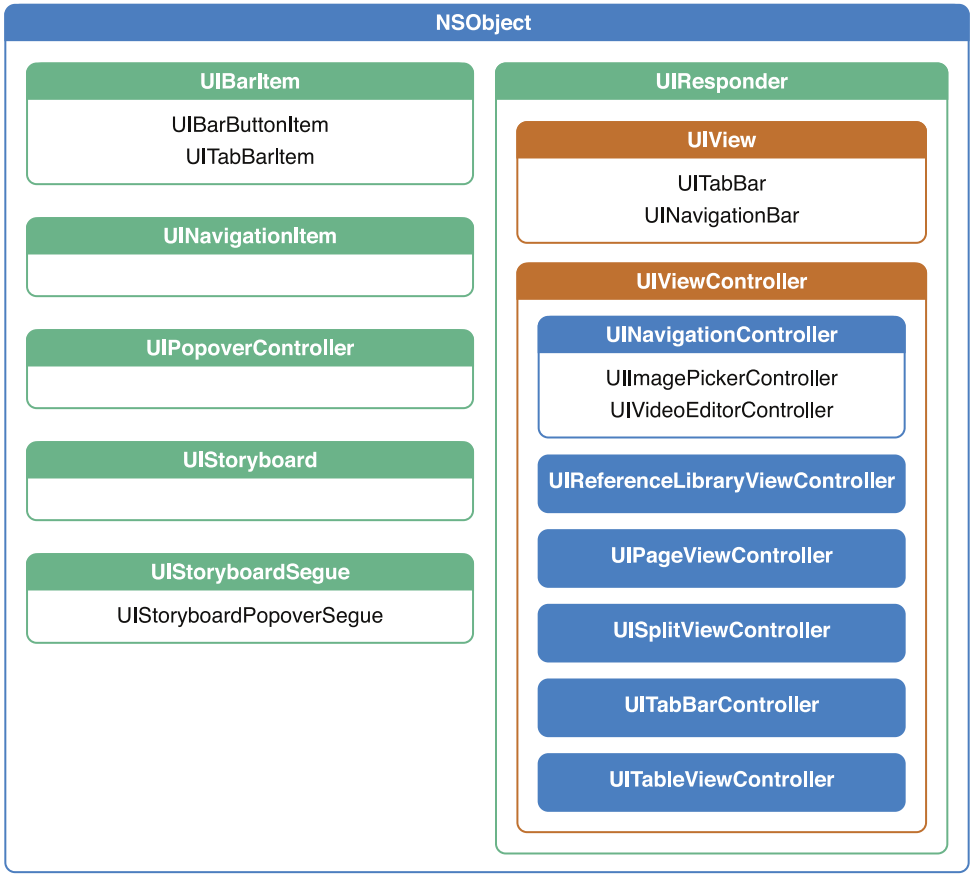

This example demonstrates a few factors common to view controllers:

- Every view is controlled by only one view controller. When a view is assigned to the view controller's `view` property, the view controller owns it. If the view is a subview, it might be controlled by the same view controller or a different view controller. You'll learn more about how to use multiple view controllers to organize a single view hierarchy when you learn about container view controllers.
- Each view controller interacts with a subset of your app's data. For example, the Photo controller needs to know only the photo to be displayed.
- Because each view controller provides only a subset of the user experience, the view controllers must communicate with each other to make this experience seamless. They may also communicate with other controllers, such as data controllers or document objects.

# A Taxonomy of View Controllers

Figure 1-5 shows the view controller classes available in the UIKit framework along with other classes important to view controllers. For example, the `UITabBarController` object manages a `UITabBar` object, which actually displays the tabs associated with the tab bar interface. Other frameworks define additional view controller classes not shown in this figure.

**Figure 1-5** View controller classes in UIKit




View controllers, both those provided by iOS and those you define, can be divided into two general categories—content view controllers and container view controllers—which reflect the role the view controller plays in an app.

## Content View Controllers Display Content

A **content view controller** presents content on the screen using a view or a group of views organized into a view hierarchy. The controllers described up to this point have been content view controllers. A content view controller usually knows about the subset of the app's data that is relevant to the role the controller plays in the app.

Here are common examples where your app uses content view controllers:

- To show data to the user
- To collect data from the user
- To perform a specific task
- To navigate between a set of available commands or options, such as on the launch screen for a game

Content view controllers are the primary coordinating objects for your app because they know the specific details of the data and tasks your app offers the user.

Each content view controller object you create is responsible for managing all the views in a single view hierarchy. 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 content view controllers to manage the same view hierarchy. Similarly, you should not use a single content view controller object to manage multiple screens' worth of content.

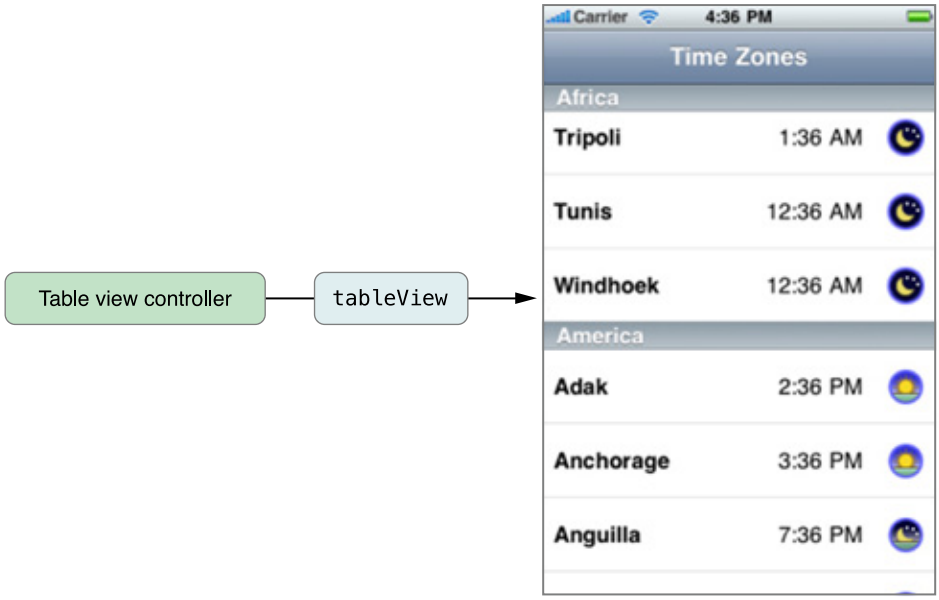
For information about defining your content view controller and implementing the required behaviors, see "Creating Custom Content View Controllers" (page 43).

## About Table View Controllers

Many apps display tabular data. For this reason, iOS provides a built-in subclass of the `UIViewController` class designed specifically for managing tabular data. This class, `UITableViewController`, manages a table view and adds support for many standard table-related behaviors such as selection management, row editing, and table configuration. This additional support is there to minimize the amount of code you must write to create and initialize a table-based interface. You can also subclass `UITableViewController` to add other custom behaviors.

Figure 1-6 shows an example using a table view controller. Because it is a subclass of the `UIViewController` class, 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 1-6** Managing tabular data




For more information about table views, see *Table View Programming Guide for iOS*.

## Container View Controllers Arrange Content of Other View Controllers

A **container view controller** contains content owned by other view controllers. These other view controllers are explicitly assigned to the container view controller as its children. A container controller can be both a parent to other controllers and a child of another container. Ultimately, this combination of controllers establishes a view controller hierarchy.

Each type of container view controller establishes a user interface that its children operate in. The visual presentation of this user interface and the design it imposes on its children can vary widely between different types of containers. For example, here are some ways that different container view controllers may distinguish themselves:

- A container provides its own API to manage its children.
- A container decides whether the children have a relationship between them and what that relationship is.

- A container manages a view hierarchy just as other view controllers do. A container can also add the views of any of its children into its view hierarchy. The container decides when such a view is added and how it should be sized to fit the container's view hierarchy, but otherwise the child view controller remains responsible for the view and its subviews.
- A container might impose specific design considerations on its children. For example, a container might limit its children to certain view controller classes, or it might expect those controllers to provide additional content needed to configure the container's views.

The built-in container classes are each organized around an important user interface principle. You use the user interfaces managed by these containers to organize complex apps.

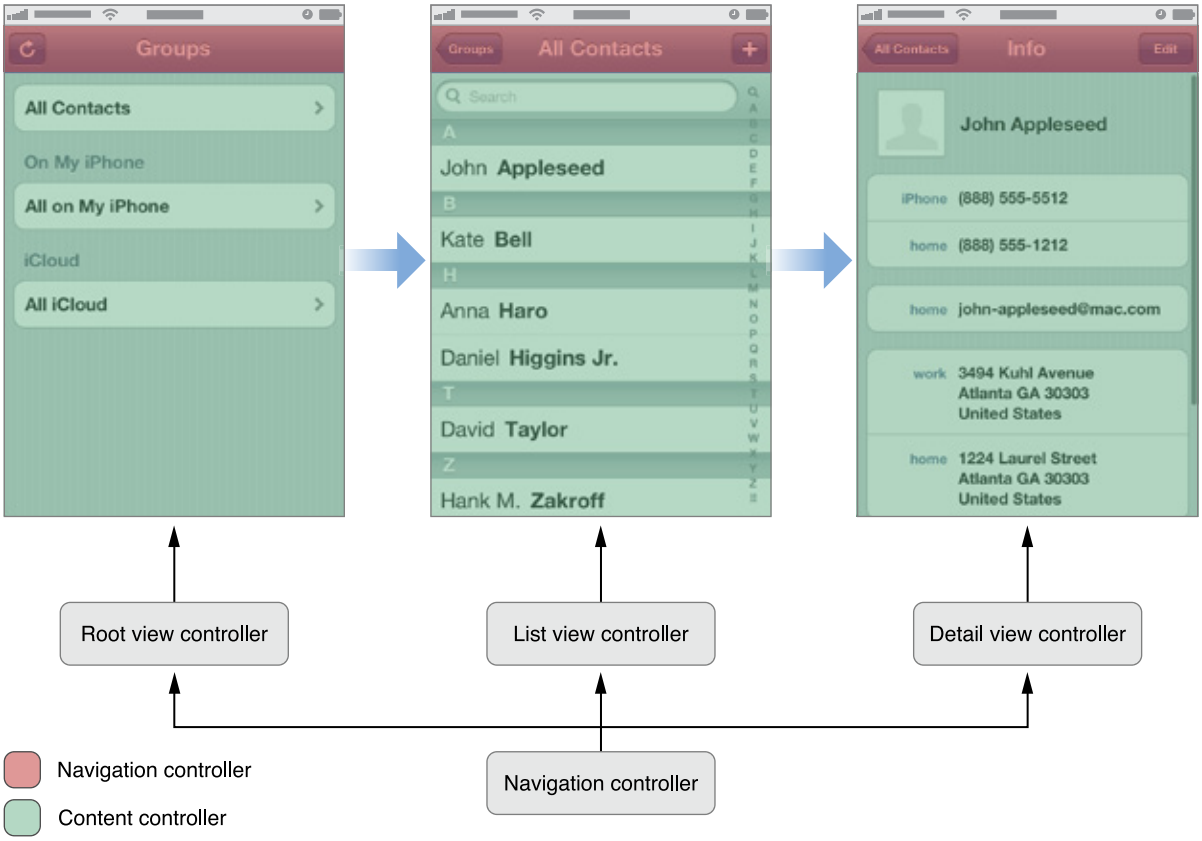
## About Navigation Controllers

A **navigation controller** presents data that is organized hierarchically and is an instance of the `UINavigationController` class. The methods of this class provide support for managing a stack-based collection of content 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.

Figure 1-7 shows screens from the Contacts app, which uses a navigation controller to present contact information to the user. The navigation bar at the top of each page is owned by the navigation controller. The rest of each screen displayed to the user is managed by a content view controller that presents the information

at that specific level of the data hierarchy. 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 1-7** Navigating hierarchical data




Although a navigation controller's primary job is to manage its child 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 button (for navigating back to previous screens), and any custom controls the current view controller needs. You do not directly modify the views owned by the view controller. Instead, you configure the controls that the navigation controller displays by setting properties on each child view controller.

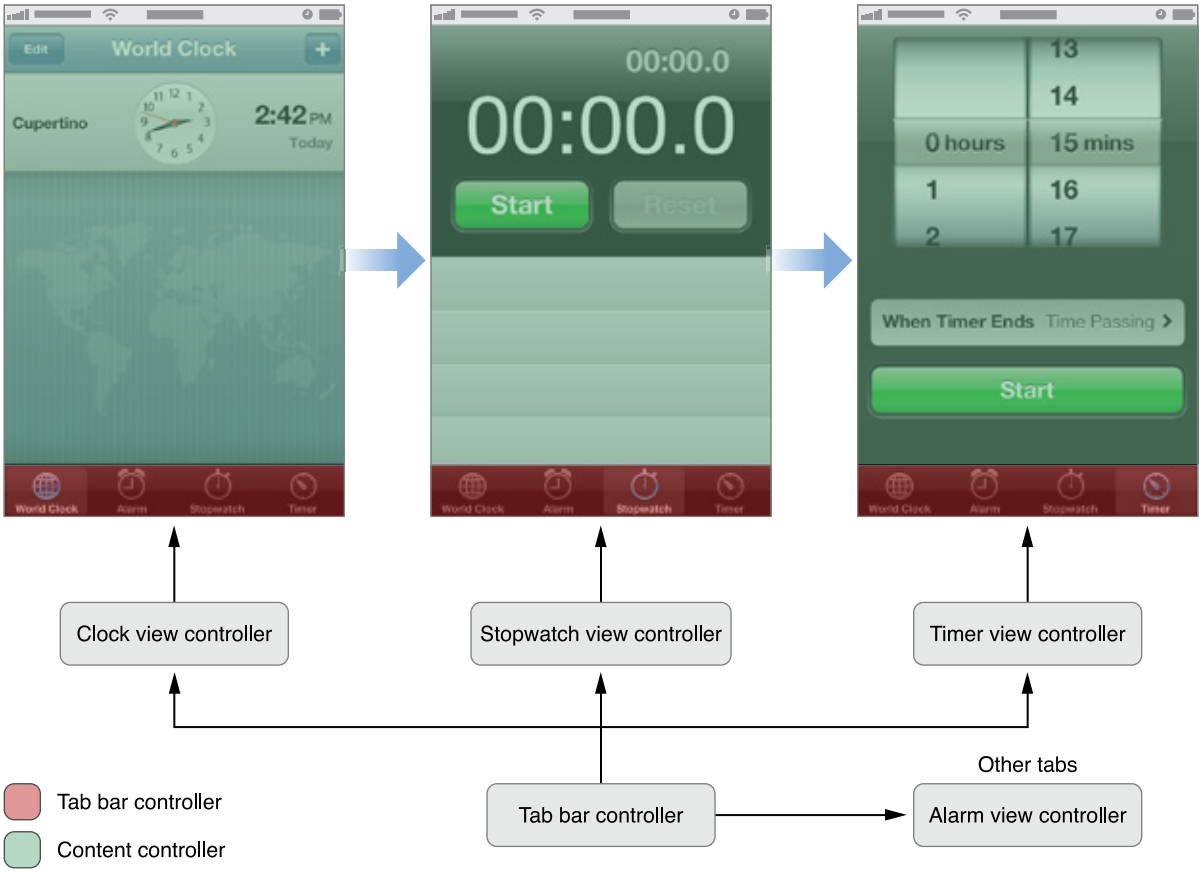
For information about how to configure and use navigation controller objects, see "Navigation Controllers".

## About Tab Bar Controllers

A **tab bar controller** is a container view controller that you use to divide your app into two or more distinct modes of operation. A tab bar controller is an instance of the `UITabBarController` class. The tab bar has multiple tabs, each represented by a child view controller. Selecting a tab causes the tab bar controller to display the associated view controller's view on the screen.

Figure 1-8 shows several modes of the Clock app along with the relationships between the corresponding view controllers. Each mode has a content view controller to manage the main content area. In the case of the Clock app, 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 content view controllers to present a single screen.

**Figure 1-8** Different modes of the Clock app




You use tab bar controllers when your app either presents different types of data or presents the same data in different ways.

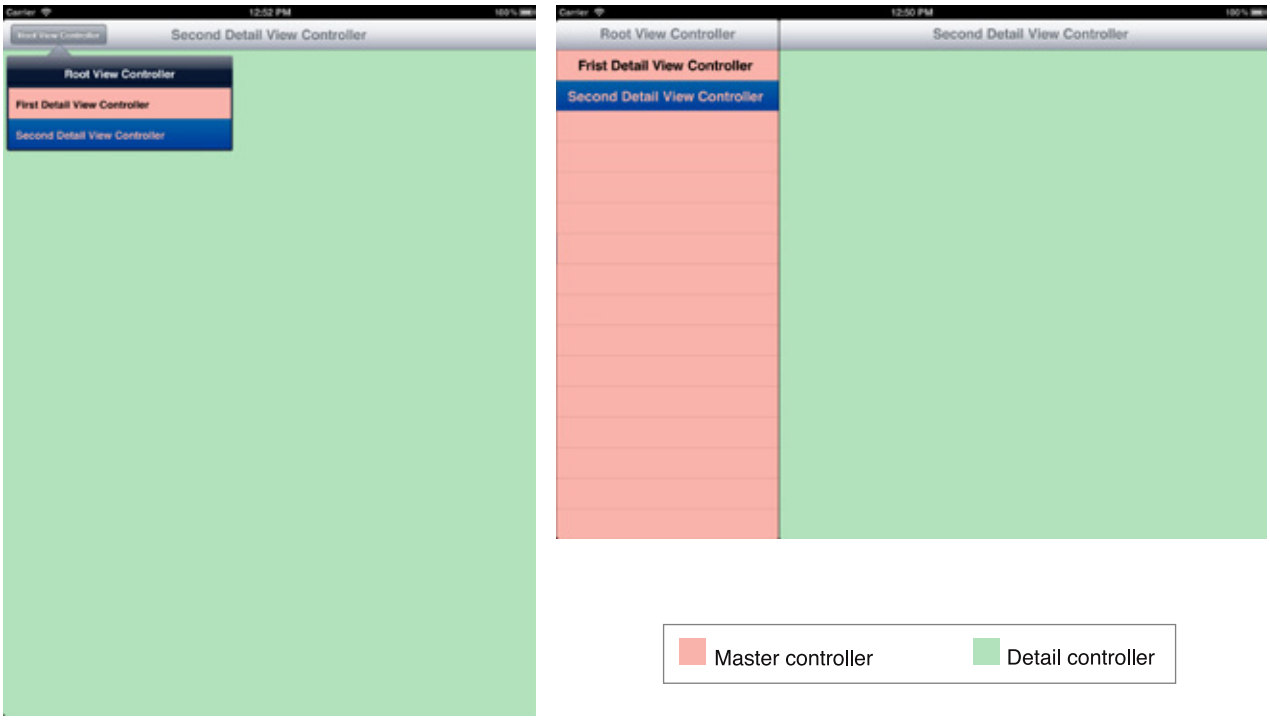
For information about how to configure and use a tab bar controller, see “Tab Bar Controllers”.

## About Split View Controllers

A **split view controller** divides the screen into multiple parts, each of which can be updated separately. The appearance of a split view controller may vary depending on its orientation. A split view controller is an instance of the `UISplitViewController` class. The contents of a split view interface are derived from two child view controllers.

Figure 1-9 shows a split view interface from the *MultipleDetailViews* sample app. In portrait mode, only the detail view is displayed. The list view is made available using a popover. However, when displayed in landscape mode, the split view controller displays the contents of both children side by side.

**Figure 1-9** A master-detail interface in portrait and landscape modes




Split view controllers are supported on iPad only and are designed to help you take advantage of the larger screen of that device. They are the preferred way to implement master-detail interfaces in iPad apps.

For information about how to configure and use a split view controller, see “Popovers”.

## About Popover Controllers

Look again at Figure 1-9. When the split view controller is displayed in portrait mode, the master views is displayed in a special control, known as a *popover*. In an iPad app, you can use popover controllers (`UIPopoverController`) to implement popovers in your own app.

A popover controller is not actually a container; it does not inherent from `UIViewController` at all. But, in practice, a popover controller is similar to a container, so you apply the same programming principles when you use them.

For information about how to configure and use a popover controller, see “Popovers”.

### About Page View Controllers

A **page view controller** is a container view controller used to implement a page layout. That layout allows users to flip between discrete pages of content as if it were a book. A page view controller is an instance of the `UIPageViewController` class. Each content page is provided by a content view controller. The page view controller manages the transitions between pages. When new pages are required, the page view controller calls an associated data source to retrieve a view controller for the next page.

For information about how to configure and use a page view controller, see "Page View Controllers".

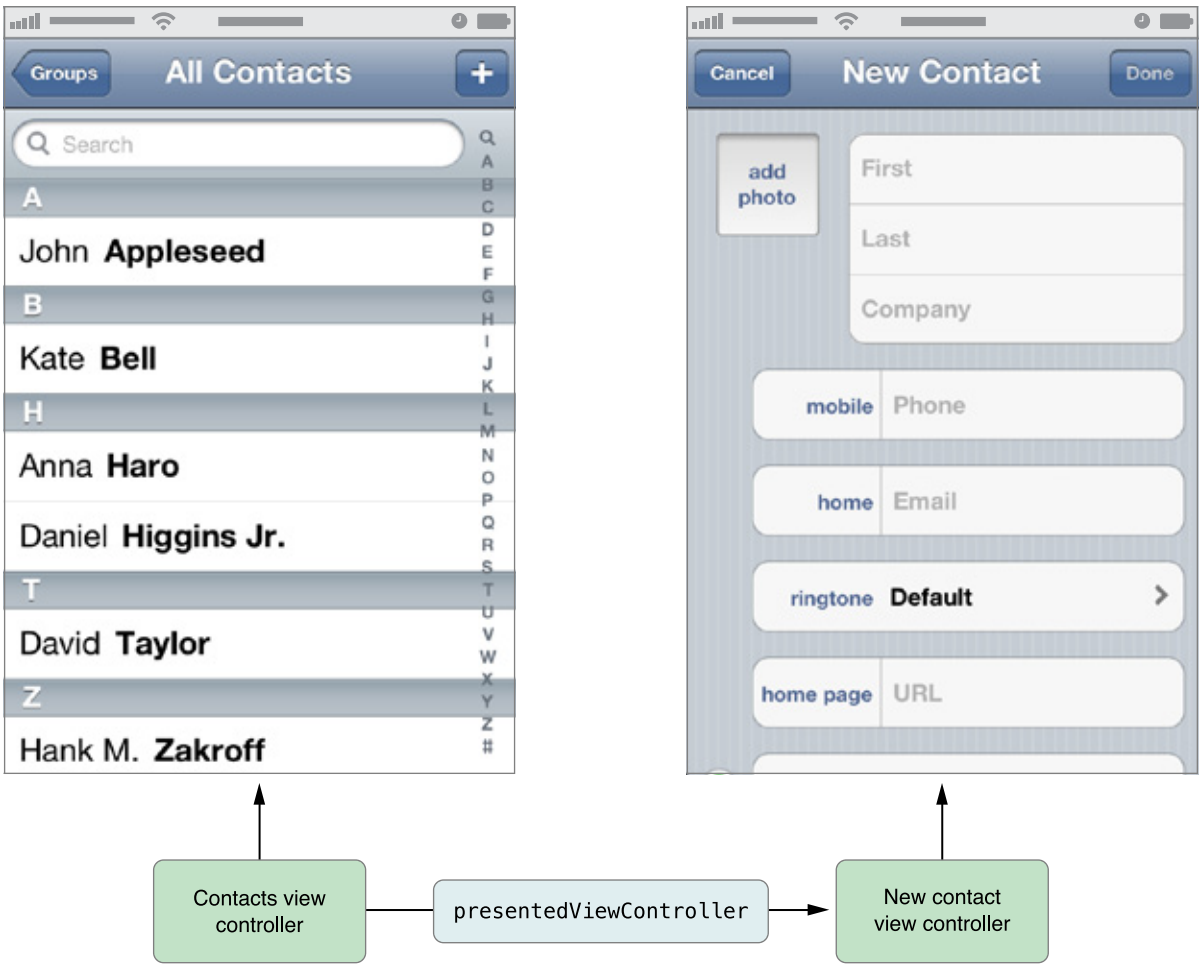
## A View Controller's Content Can Be Displayed in Many Ways

For a view controller's contents to be visible to the user, it must be associated with a window. There are many ways you can do this in your app:

- Make the view controller a window's root view controller.
- Make the view controller a child of a container.
- Show the view controller in a popover control.
- Present it from another view controller.

Figure 1-10 shows an example from the Contacts app. When the user clicks the plus button to add a new contact, the Contacts view controller presents the New Contact view controller. 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 that point the information is transmitted to the Contacts view controller, which then dismisses the controller it presented.

**Figure 1-10** Presenting a view controller




A presented view controller isn't a specific type of view controller—the presented view controller can be either a content or a container view controller with an attached content view controller. In practice, the content view controller is designed specifically to be presented by another controller, so it can be useful to think of it as a variant of a content view controller. Although container view controllers define specific relationships between the managed view controllers, using presentation allows you to define the relationship between the view controller being presented and the view controller presenting it.

Most of the time, you present view controllers to gather information from the user or capture the user's attention for some specific purpose. Once that purpose is completed, the presenting view controller dismisses the presented view controller and returns to the standard app interface.

It is worth noting that a presented view controller can itself present another view controller. This ability to chain view controllers together can be useful when you need to perform several modal actions sequentially. For example, if the user taps the Add Photo button in the New Contact screen in Figure 1-10 and wants to choose an existing image, the New Contact view controller presents an image picker interface. The user must dismiss the image picker screen and then dismiss the New Contact screen separately to return to the list of contacts.

When presenting a view controller, one view controller determines how much of the screen is used to present the view controller. The portion of the screen is called the *presentation context* By default, the presentation context is defined to cover the window.

For more information about how to present view controllers in your app, see "Presenting View Controllers from Other View Controllers" (page 88).

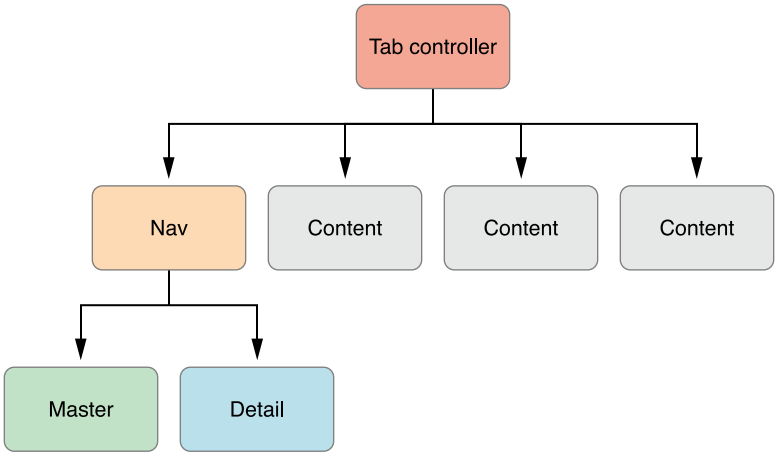
## View Controllers Work Together to Create an App's Interface

View controllers manage their views and other associated objects, but they also work with other view controllers to provide a seamless user interface. The distribution of work and communication between your app's view controllers is an essential part of working with them. Because these relationships are so important to building complex apps, this next section reviews the relationships already discussed and describes them in more detail.

## Parent-Child Relationships Represent Containment

A view controller hierarchy starts with a single parent, the root view controller of a window. If that view controller is a container, it may have children that provide content. Those controllers, in turn, may also be containers with children of their own. Figure 1-11 shows an example of a view controller hierarchy. The root view controller is a tab view controller with four tabs. The first tab uses a navigation controller with children of its own and the other three tabs are managed by content view controllers with no children.

**Figure 1-11** Parent-child relationships




The area each view controller fills is determined by its parent. The root view controller's area is determined by the window. In Figure 1-11, the tab view controller gets its size from the window. It reserves space for its tab bar and gives the remainder of the space to its children. If the navigation controller were the control displayed right now, it reserves space for its navigation bar and hands the rest to its content controller. At each step, the child view controller's view is resized by the parent and placed into the parent's view hierarchy.

This combination of views and view controllers also establishes the responder chain for events handled by your app.

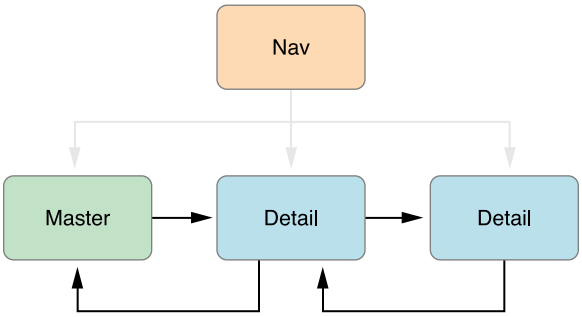
## Sibling Relationships Represent Peers Inside a Container

The kind of container defines the relationships (if any exists) shared by its children. For example, compare the tab view controller and navigation controller.

- In a tab view controller, the tabs represent distinct screens of content; tab bar controllers do not define a relationship between its children, although your app can choose to do so.
- In a navigation controller, siblings display related views arranged in a stack. Siblings usually share a connection with adjacent siblings.

Figure 1-12 shows a common configuration of view controllers associated with a navigation controller. The first child, the master, shows the available content without showing all of the details. When an item is selected, it pushes a new sibling onto the navigation controller so that the user can see the additional details. Similarly, if the user needs to see more details, this sibling can push another view controller that shows the most detailed content available. When siblings have a well defined relationship as in this example, they often coordinate with each other, either directly or through the container controller. See Figure 1-15 (page 32).

**Figure 1-12** Sibling relationships in a navigation controller

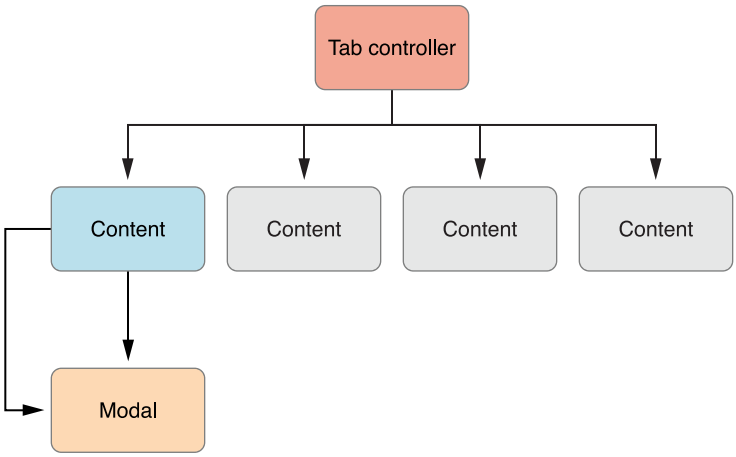



## Presentation Represents a Transient Display of Another Interface

A view controller presents another view controller when it wants that view controller to perform a task. The presenting view controller is in charge of this behavior. It configures the presented view controller, receives information from it, and eventually dismisses it. However, while it is being presented, the presented view controller's view is temporarily added to the window's view hierarchy.

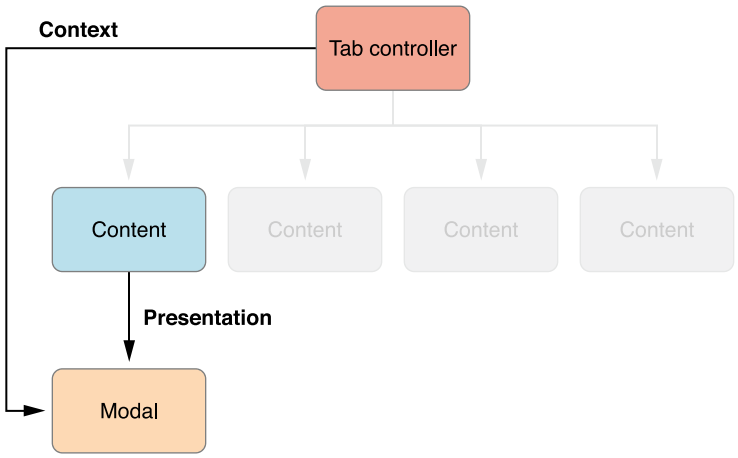
In Figure 1-13, a content view controller attached to the tab view presents a view controller to perform a task. The content controller is the presenting view controller, and the modal view controller is the presented view controller.

**Figure 1-13** Modal presentation by a content view



When a view controller is presented, the portion of the screen that it covers is defined by a presentation context provided to it by another view controller. The view controller that provides the presentation context does not need be the same view controller that presented it. Figure 1-14 shows the same view controller hierarchy that is presented in Figure 1-13. You can see that the content view presented the view controller, but it did not provide the presentation context. Instead, the view controller was presented by the tab controller. Because of this, even though the presenting view controller only covers the portion of the screen provided to it by the tab view controller, the presented view controller uses the entire area owned by the tab view controller.

**Figure 1-14** The actual presentation is performed by the root view controller.




## Control Flow Represents Overall Coordination Between Content Controllers

In an app with multiple view controllers, view controllers are usually created and destroyed throughout the lifetime of the app. During their lifetimes, the view controllers communicate with each other to present a seamless user experience. These relationships represent the control flow of your app.

Most commonly, this control flow happens when a new view controller is instantiated. Usually, a view controller is instantiated because of actions in another view controller. The first view controller, known as the *source view controller* directs the second view controller, the *destination view controller*. If the destination view controller presents data to the user, the source view controller usually provides that data. Similarly, if the source view controller needs information from the destination view controller, it is responsible for establishing the connection between the two view controllers.

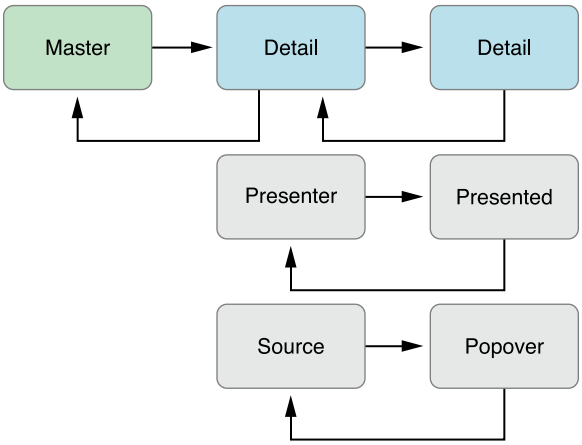
Figure 1-15 shows the most common examples of these relationships.

In the figure:

- A child of a navigation controller pushes another child onto the navigation stack.
- A view controller presents another view controller.

- A view controller displays another view controller in a popover.

**Figure 1-15** Communication between source and destination view controllers




Each controller is configured by the one preceding it. When multiple controllers work together, they establish a communication chain throughout the app.

The control flow at each link in this chain is defined by the destination view controller. The source view controller uses the conventions provided by the destination view controller.

- The destination view controller provides properties used to configure its data and presentation.
- If the destination view controller needs to communicate with view controllers preceding it in the chain, it uses delegation. When the source view controller configures the destination view controller's other properties, it is also expected to provide an object that implements the delegate's protocol.

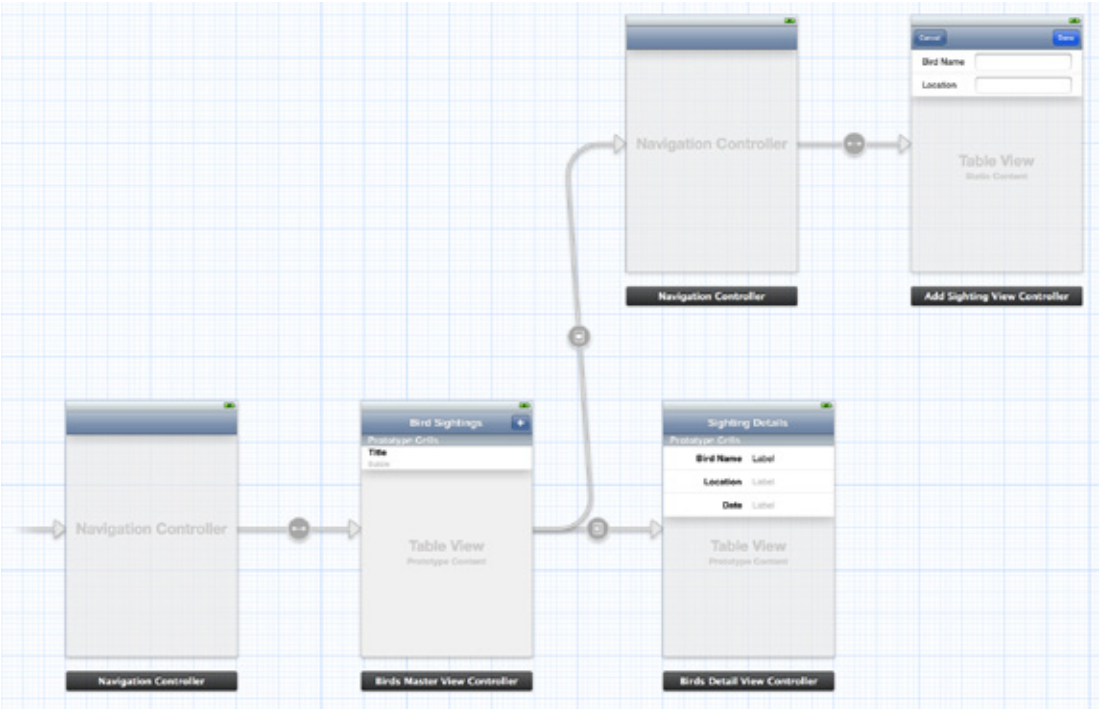
The benefit of using this control flow convention is that there is a clean division of responsibilities between each pair of source and destination view controllers. Data flows down the path when the source view controller asks the destination view controller to perform a task; the source view controller drives the process. For example, it might provide the specific data object that the destination controller should display. In the other direction, data flows up the path when a view controller needs to communicates information back to the source controller that spawned it. For example, it might communicate when the task completes.

Also, by consistently implementing this control flow model, you ensure that destination view controllers never know too much about the source view controller that configured them. When it does know about a view controller earlier in the chain, it knows only that the class implements the delegate protocol, not the class of the class. By keeping view controllers from knowing too much about each other, individual controllers become more reusable. For someone reading your code, a consistently implemented control flow model makes it easy to see the communication path between any pair of controllers.

# Storyboards Help You Design Your User Interface

When you implement your app using storyboards, you use Interface Builder to organize your app's view controllers and any associated views. Figure 1-16 shows an example interface layout from Interface Builder. The visual layout of Interface Builder allows you to understand the flow through your app at a glance. You can see what view controllers are instantiated by your app and their order of instantiation. But more than that, you can configure complex collections of views and other objects in the storyboard. The resulting storyboard is stored as a file in your project. When you build your project, the storyboards in your project are processed and copied into the app bundle, where they are loaded by your app at runtime.

**Figure 1-16** A storyboard diagram in Interface Builder




Often, iOS can automatically instantiate the view controllers in your storyboard at the moment they are needed. Similarly, the view hierarchy associated with each controller is automatically loaded when it needs to be displayed. Both view controllers and views are instantiated with the same attributes you configured in Interface Builder. Because most of this behavior is automated for you, it greatly simplifies the work required to use view controllers in your app.

The full details of creating storyboards are described in *Xcode Overview*. For now, you need to know some of the essential terminology used when implementing storyboards in your app.

A **scene** represents an onscreen content area that is managed by a view controller. You can think of a scene as a view controller and its associated view hierarchy.

You create **relationships** between scenes in the same storyboard. Relationships are expressed visually in a storyboard as a connection arrow from one scene to another. Interface Builder usually infers the details of a new relationship automatically when you make a connection between two objects. Two important kinds of relationships exist:

- **Containment** represents a parent-child relationship between two scenes. View controllers contained in other view controllers are instantiated when their parent controller is instantiated. For example, the first connection from a navigation controller to another scene defines the first view controller pushed onto the navigation stack. This controller is automatically instantiated when the navigation controller is instantiated.

  An advantage to using containment relationships in a storyboard is that Interface Builder can adjust the appearance of the child view controller to reflect the presence of its ancestors. This allows Interface Builder to display the content view controller as it appears in your final app.

- A **segue** represents a visual transition from one scene to another. At runtime, segues can be triggered by various actions. When a segue is triggered, it causes a new view controller to be instantiated and transitioned onscreen.

  Although a segue is always from one view controller to another, sometimes a third object can be involved in the process. This object actually triggers the segue. For example, if you make a connection from a button in the source view controller's view hierarchy to the destination view controller, when the user taps the button, the segue is triggered. When a segue is made directly from the source view controller to the destination view controller, it usually represents a segue you intend to trigger programatically.

  Different kinds of segues provide the common transitions needed between two different view controllers:

  - A **push segue** pushes the destination view controller onto a navigation controller's stack.
  - A **modal segue** presents the destination view controller.
  - A **popover segue** displays the destination view controller in a popover.
  - A **custom segue** allows you to design your own transition to display the destination view controller.

  Segues share a common programming model. In this model, the destination controller is instantiated automatically by iOS and then the source view controller is called to configure it. This behavior matches the control flow model described in "Control Flow Represents Overall Coordination Between Content Controllers" (page 31).

You can also create connections between a view controller and objects stored in the same scene. These outlets and actions enable you to carefully define the relationships between the view controller and its associated objects. Connections are not normally visible in the storyboard itself but can be viewed in the Connections Inspector of Interface Builder.

# Using View Controllers in Your App

Whether you are working with view controllers provided by iOS, or with custom controllers you've created to show your app's content, you use a similar set of techniques to actually work with the view controllers.

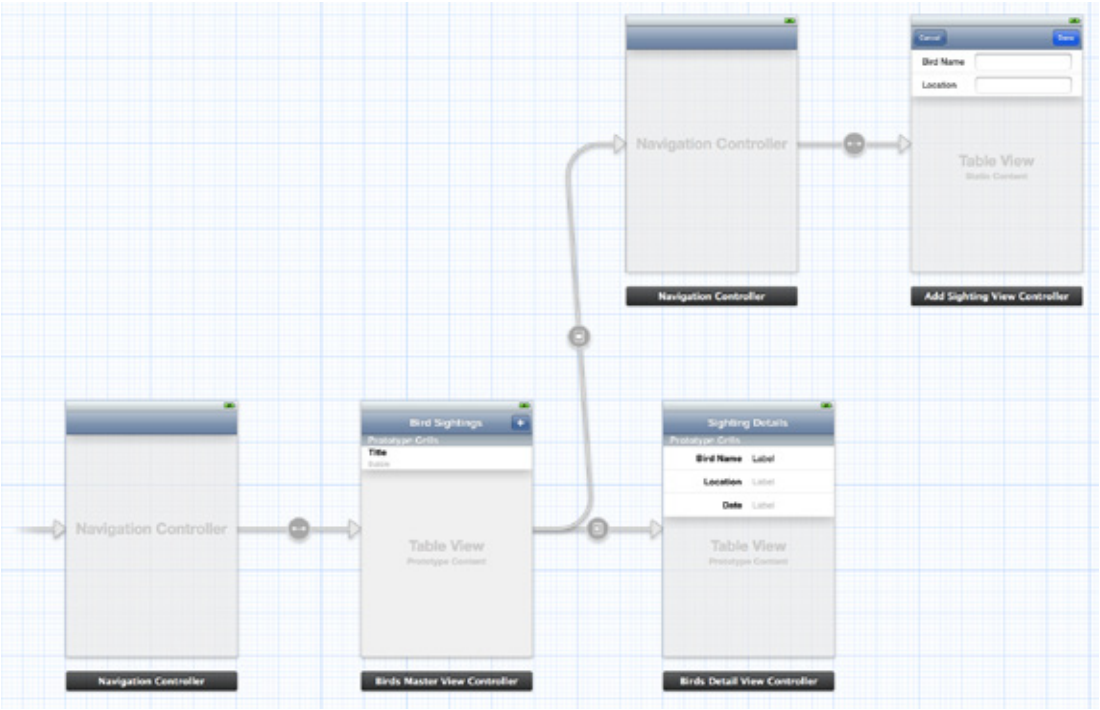
The most common technique for working with view controllers is to place them inside a storyboard. Placing view controllers in a storyboard allows you to directly establish relationships between the view controllers in your app without writing code. You can see the flow of control—from controllers created when your app first launches, to controllers that are instantiated in response to a user's actions. iOS manages most of this process by instantiating these view controllers only when the app needs them.

Sometimes you may need to create a view controller by allocating and initializing it programmatically. When working with view controllers directly, you must write code that instantiates the view controller, configures it, and displays it.

# Working with View Controllers in Storyboards

Figure 2-1 shows an example of a storyboard. This storyboard includes view controllers, associated views, and connection arrows that establish relationships between the view controllers. In effect, this storyboard tells a story, starting with one scene and later showing others in response to a user's actions.

**Figure 2-1** A storyboard holds a set of view controllers and associated objects




A storyboard may designate one view controller to be the initial view controller. If a storyboard represents a specific workflow through part of your UI, the initial view controller represents the first scene in that workflow.

You establish relationships from the initial view controller to other view controllers in the storyboard. In turn, you establish relationships from those view controllers to others, eventually connecting most or all of the storyboard's scenes into a single connected graph. The type of relationship you establish determines when a connected view controller is instantiated by iOS, as follows:

- If the relationship is a segue, the destination view controller is instantiated when the segue is triggered.
- If the relationship represents containment, the child view controller is instantiated when its parent is instantiated.
- If the controller is not the destination or child of another controller, it is never instantiated automatically. You must instantiate it from the storyboard programmatically.

To identify a specific view controller or segue inside a storyboard, use Interface Builder to assign it an identifier string that uniquely identifies it. To programmatically load a view controller from the storyboard, you must assign it an identifier. Similarly, to trigger a segue programmatically, it must also be assigned an identifier. When a segue is triggered, that segue's identifier is passed to the source view controller, which uses it to determine which segue was triggered. For this reason, consider labeling every segue with an identifier.

When you build an app using storyboards, you can use a single storyboard to hold all of its view controllers, or you can create multiple storyboards and implement a portion of the user interface in each. One storyboard in your app is almost always designated as the main storyboard. If there is a main storyboard, iOS loads it automatically; other storyboards must be explicitly loaded by your app.

## The Main Storyboard Initializes Your App's User Interface

The main storyboard is defined in the app's Information property list file. If a main storyboard is declared in this file, then when your app launches, iOS performs the following steps:

1. It instantiates a window for you.
2. It loads the main storyboard and instantiates its initial view controller.
3. It assigns the new view controller to the window's `rootViewController` property and then makes the window visible on the screen.

Your app delegate is called to configure the initial view controller before it is displayed. The precise set of steps iOS uses to load the main storyboard is described in "Coordinating Efforts Between View Controllers" (page 98).

## Segues Automatically Instantiate the Destination View Controller

A segue represents a triggered transition that brings a new view controller into your app's user interface.

Segues contain a lot of information about the transition, including the following:

- The object that caused the segue to be triggered, known as the *sender*
- The source view controller that starts the segue
- The destination view controller to be instantiated
- The kind of transition that should be used to bring the destination view controller onscreen
- An optional identifier string that identifies that specific segue in the storyboard

When a segue is triggered, iOS takes the following actions:

1. It instantiates the destination view controller using the attribute values you provided in the storyboard.

2. It gives the source view controller an opportunity to configure the new controller.
3. It performs the transition configured in the segue.

> **Note:** When you implement custom view controllers, each destination view controller declares public properties and methods used by the source view controller to configure its behavior. In return, your custom source view controllers override storyboard methods provided by the base class to configure the destination view controller. The precise details are in "Coordinating Efforts Between View Controllers" (page 98).

## Triggering a Segue Programmatically

A segue is usually triggered because an object associated with the source view controller, such as a control or gesture recognizer, triggered the segue. However, a segue can also be triggered programmatically by your app, as long as the segue has an assigned identifier. For example, if you are implementing a game, you might trigger a segue when a match ends. The destination view controller then displays the match's final scores.

You programmatically trigger the segue by calling the source view controller's `performSegueWithIdentifier:sender:` method, passing in the identifier for the segue to be triggered. You also pass in another object that acts as the sender. When the source controller is called to configure the destination view controller, both the sender object and the identifier for the segue are provided to it.

Listing 2-1 shows a simple method that triggers a segue. This example is a portion of a larger example described in "Creating an Alternate Landscape Interface" (page 80). In this abbreviated form, you can see that the view controller is receiving an orientation notification. When the view controller is in portrait mode and the device is rotated into landscape orientation, the method uses a segue to present a different view controller onscreen. Because the notification object in this case provides no useful information for performing the segue command, the view controller makes itself the sender.

**Listing 2-1** Triggering a segue programmatically

```
- (void)orientationChanged:(NSNotification *)notification
{
    UIDeviceOrientation deviceOrientation = [UIDevice currentDevice].orientation;
    if (UIDeviceOrientationIsLandscape(deviceOrientation) &&
        !isShowingLandscapeView)
    {
        [self performSegueWithIdentifier:@"DisplayAlternateView" sender:self];
        isShowingLandscapeView = YES;
    }
```

```
// Remainder of example omitted.
}
```

If a segue can be triggered only programmatically, you usually draw the connection arrow directly from the source view controller to the destination view controller.

## Instantiating a Storyboard's View Controller Programmatically

You may want to programmatically instantiate a view controller without using a segue. A storyboard is still valuable, because you can use it to configure the attributes of the view controller as well as its view hierarchy. However, if you do instantiate a view controller programmatically, you do not get any of the behavior of a segue. To display the view controller, you must implement additional code. For this reason, you should rely on segues where possible and use this technique only when needed.

Here are the steps your code needs to implement:

1. Obtain a storyboard object (an object of the `UIStoryboard` class).

   If you have an existing view controller instantiated from the same storyboard, read its `storyboard` property to retrieve the storyboard. To load a different storyboard, call the `UIStoryboard` class's `storyboardWithName:bundle:` class method, passing in the name of the storyboard file and an optional bundle parameter.

2. Call the storyboard object's `instantiateViewControllerWithIdentifier:` method, passing in the identifier you defined for the view controller when you created it in Interface Builder.

   Alternatively, you can use the `instantiateInitialViewController` method to instantiate the initial view controller in a storyboard, without needing to know its identifier.

3. Configure the new view controller by setting its properties.

4. Display the new view controller. See "Displaying a View Controller's Contents Programmatically" (page 41).

Listing 2-2 shows an example of this technique. It retrieves the storyboard from an existing view controller and instantiates a new view controller using it.

**Listing 2-2** Instantiating another view controller inside the same storyboard

```
- (IBAction)presentSpecialViewController:(id)sender {
    UIStoryboard *storyboard = self.storyboard;
    SpecialViewController *svc = [storyboard
instantiateViewControllerWithIdentifier:@"SpecialViewController"];
```

```
    // Configure the new view controller here.

    [self presentViewController:svc animated:YES completion:nil];
}
```

Listing 2-3 shows another frequently used technique. This example loads a new storyboard and instantiates its initial view controller. It uses this view controller as the root view controller for a new window being placed on an external screen. To display the returned window, your app calls the window's `makeKeyAndVisible` method.

**Listing 2-3** Instantiating a view controller from a new storyboard

```
- (UIWindow*) windowFromStoryboard: (NSString*) storyboardName
                                  onScreen: (UIScreen*) screen
{
    UIWindow *window = [[UIWindow alloc] initWithFrame:[screen bounds]];
    window.screen = screen;

    UIStoryboard *storyboard = [UIStoryboard storyboardWithName:storyboardName
bundle:nil];
    MainViewController *mainViewController = [storyboard
instantiateInitialViewController];
    window.rootViewController = mainViewController;

    // Configure the new view controller here.

    return window;
}
```

## Transitioning to a New Storyboard Requires a Programmatic Approach

Segues connect only scenes that are stored in the same storyboard. To display a view controller from another storyboard, you must explicitly load the storyboard file and instantiate a view controller inside it.

There is no requirement that you create multiple storyboards in your app. Here, though, are a few cases where multiple storyboards might be useful to you:

- You have a large programming team, with different portions of the user interface assigned to different parts of the team. In this case, each sub team owns a storyboard limiting the number of team members working on any specific storyboard's contents.
- You purchased or created a library that predefines a collection of view controller types; the contents of those view controllers are defined in a storyboard provided by the library.
- You have content that needs to be displayed on an external screen. In this case, you might keep all of the view controllers associated with the alternate screen inside a separate storyboard. An alternative pattern for the same scenario is to write a custom segue.

### Containers Automatically Instantiate Their Children

When a container in a storyboard is instantiated, its children are automatically instantiated at the same time. The children must be instantiated at the same time to give the container controller some content to display.

Similarly, if the child that was instantiated is also a container, its children are also instantiated, and so on, until no more containment relationships can be traced to new controllers. If you place a content controller inside a navigation controller inside a tab bar controller, when the tab bar is instantiated, the three controllers are simultaneously instantiated.

The container and its descendants are instantiated before your view controller is called to configure them. Your source view controller (or app delegate) can rely on all the children being instantiated. This instantiation behavior is important, because your custom configuration code rarely configures the container(s). Instead, it configures the content controllers attached to the container.

## Instantiating a Non-Storyboard View Controller

To create a view controller programmatically without the use of the storyboard, you use Objective-C code to allocate and initialize the view controller. You gain none of the benefits of storyboards, meaning you have to implement additional code to configure and display the new view controller.

## Displaying a View Controller's Contents Programmatically

For a view controller's content to be useful, it needs to be displayed on screen. There are several options for displaying a view controller's contents:

- Make the view controller the root view controller of a window.
- Make it a child of a visible container view controller.
- Present it from another visible view controller.

- Present it using a popover (iPad only).

In all cases, you assign the view controller to another object—in this case, a window, a view controller, or a popover controller. This object resizes the view controller's view and adds it to its own view hierarchy so that it can be displayed.

Listing 2-4 shows the most common case, which is to assign the view controller to a window. This code assumes that a storyboard is not being used, so it performs the same steps that are normally done on your behalf by the operating system: It creates a window and sets the new controller as the root view controller. Then it makes the window visible.

**Listing 2-4** Installing the view controller as a window's root view controller

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {
    UIWindow *window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen]
bounds]];
    levelViewController = [[LevelViewController alloc] init];
    window.rootViewController = levelViewController;
    [window makeKeyAndVisible];
}
```

**Important:** Never install the view controller's view into a view hierarchy directly. To present and manage views properly, the system makes a note of each view (and its associated view controller) that you display. It uses this information later to report view controller–related events to your app. 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, the system may handle these events incorrectly.

If you are implementing your own custom container controller, you add another view controller's view to your own view hierarchy, but you also create a parent-child relationship first. This ensures that events are delivered correctly. See "Creating Custom Container View Controllers" (page 111).

# Creating Custom Content View Controllers

Custom content view controllers are the heart of your app. You use them to present your app's unique content. All apps need at least one custom content view controller. Complex apps divide the workload between multiple content controllers.

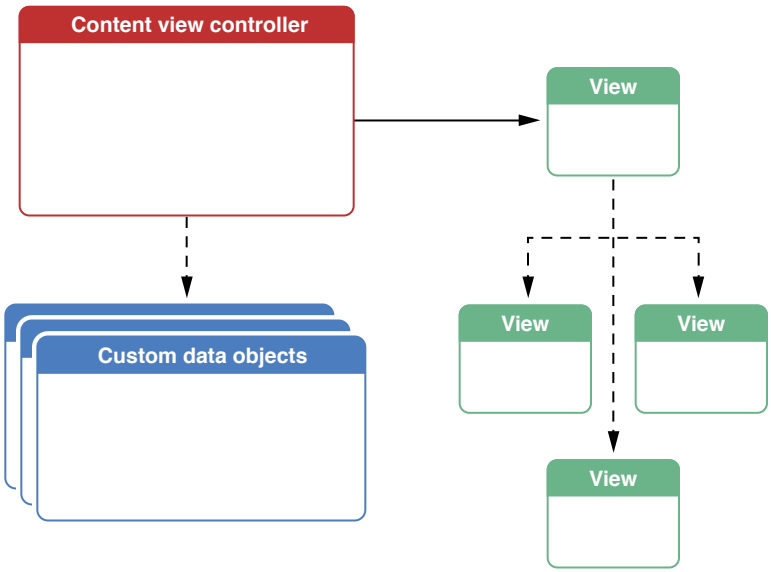
A view controller has many responsibilities. Some of these responsibilities are things that iOS requires the view controller to do. Other responsibilities are things you assign to the view controller when you define its role in your app.

## Anatomy of a Content View Controller

The `UIViewController` class provides the fundamental infrastructure for implementing all custom view controllers. You define a custom subclass of `UIViewController`. That subclass provides the necessary code to populate views with data and respond to user actions. When you want to make adjustments to the default behavior of the view controller, you override methods of the `UIViewController` class. Your view controller may also interact with other UIKit classes to implement the behavior you want.

Figure 3-1 shows some of the key objects associated directly with a content 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 have additional subviews attached to this view as well as custom objects containing the data they need to display.

**Figure 3-1** Anatomy of a content view controller




When you design a new view controller, it potentially has many responsibilities. Some of those responsibilities look inward, to the views and other objects it controls. Other responsibilities look outward to other controllers. The following sections enumerate many of the common responsibilities for a view controller.

## View Controllers Manage Resources

Some objects are instantiated when the view controller is initialized and are disposed of when your view controller is released. Other objects, like views, are needed only when the view controller's contents are visible onscreen. As a result, view controllers use resources efficiently and should be prepared to release resources when memory is scarce. Properly implementing this behavior in your app's view controllers makes it so your app uses memory and other resources—such as CPU, GPU, and battery—more efficiently.

See "Resource Management in View Controllers" (page 56).

## View Controllers Manage Views

View controllers manage their view and its subviews, but the view's frame—its position and size in its parent's view—is often determined by other factors, including the orientation of the device, whether or not the status bar is visible and even how the view controller's view is displayed in the window. Your view controller should be designed to layout its view to fit the frame provided to it.

View management has other aspects as well. Your view controller is notified when its view is about to appear and disappear from the screen. Your view controller can use this notification to perform other actions necessary to its operation.

See "Resizing the View Controller's Views" (page 69), "Supporting Multiple Interface Orientations" (page 74), "Responding to Display-Related Notifications" (page 66).

## View Controllers Respond to Events

Your view controller is often the central coordinating object for its views and controls. Typically, you design your user interface so that controls send messages to the controller when a user manipulates them. Your view controller is expected to handle the message, making any necessary changes to the views or data stored in the view controller.

Your view controller also participates in the responder chain used to deliver events to your app. You can override methods in your view controller class to have it participate directly in event handling. View controllers also are good objects to implement other behaviors—such as responding to system notifications, timers or events specific to your app.

See "Using View Controllers in the Responder Chain" (page 72).

## View Controllers Coordinate with Other Controllers

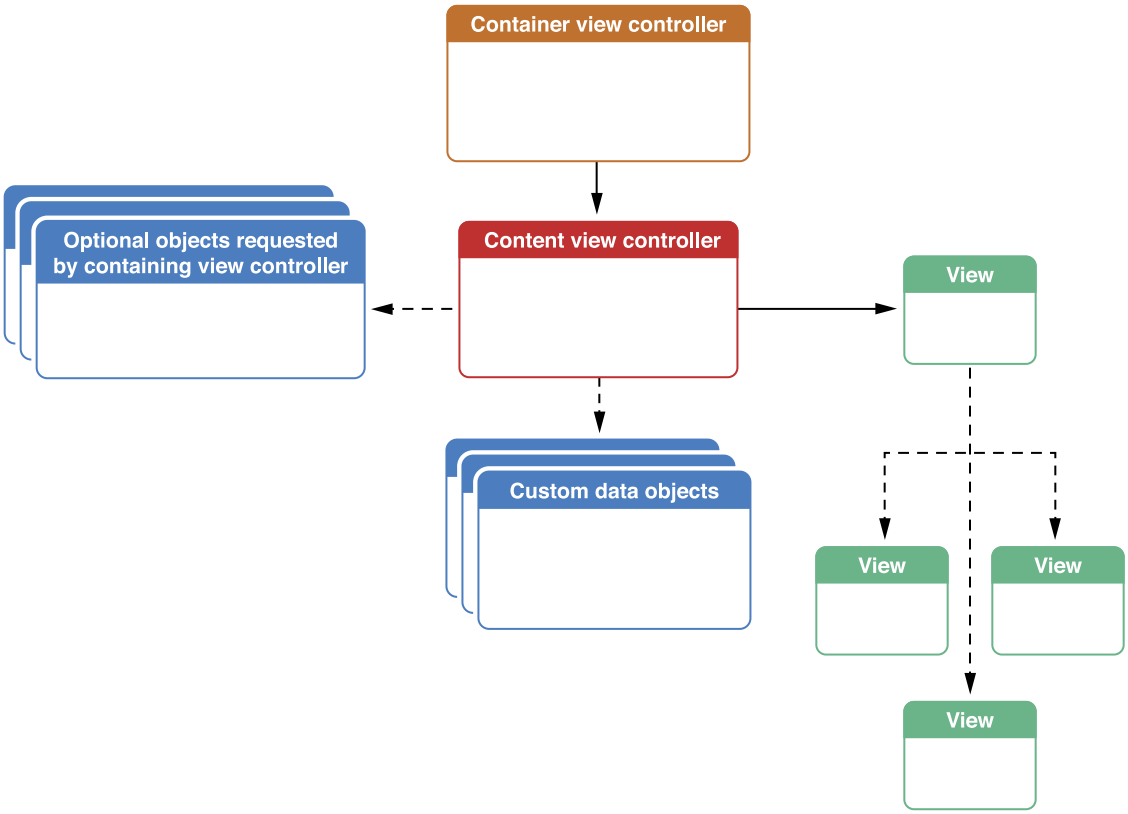
Although a view controller may create and manage many other objects, it does not usually need to expose these objects publicly to inspection or modification. It may collaborate with other objects (especially other view controllers), but it should expose the fewest number of properties and methods necessary to allow its collaborators to communicate with it. Exposing too many implementation details in your view controller class makes it difficult to modify your view controller's implementation. Collaborators that rely on these implementation details would need to be modified to continue to work with your view controller class.

See "Coordinating Efforts Between View Controllers" (page 98).

## View Controllers Often Work with Containers

If your view controller is placed inside a container view controller, the container imposes additional constraints, as shown in Figure 3-2. The container may ask your view controller to provide other objects used to configure the container's user interface. For example, a content view controller placed inside a tab view controller provides a tab bar item to display for that tab.

**Figure 3-2** A container view controller imposes additional demands on its children




The properties used to configure the containers provided by UIKit are defined by the `UIViewController` class. For more information on specific types of containers and the properties you configure to support them, see *View Controller Catalog for iOS*.

## View Controllers May Be Presented by Other View Controllers

Some view controllers you design are intended to be presented by other view controllers. You might present your view controller directly, or you might make it a child of a container view controller and present the container instead. When presented, it moves onscreen, remaining there until it is dismissed.

There are several reasons you might present a view controller:

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

Most of these reasons involve interrupting your app's workflow temporarily in order to gather or display some information. In almost all cases, the presented view controller implements a delegate. The presented view controller uses the delegate to communicate with the presenting view controller. After your app has the information it needs (or the user finishes viewing the presented information), the presented view controller communicates this back to the presenting view controller. The presenting view controller dismisses the presented view controller to return the app to its previous state.

See "Presenting View Controllers from Other View Controllers" (page 88).

## Designing Your Content View Controller

Before writing any code in your view controller, you should be able to answer some basic questions about how you intend to use it. The questions provided below are designed to help you narrow the focus of your view controller and to help you understand the role it plays in your app. In particular, it helps you identify connections—usually to other controllers—your view controller needs to perform its tasks.

- Are you using a storyboard to implement the view controller?
- When is it instantiated?
- What data does it show?
- What tasks does it perform?
- How is its view displayed onscreen?
- How does it collaborate with other view controllers?

Your answers to these questions need not be precise if you are still working out the role it plays. Still, it helps to have a general sense of what your view controller does and how other objects interact with it.

The questions above don't ask you to define the appearance of your view controller or to be precise about the implementation details of how it performs the tasks you've assigned to it. Those are important questions you need to answer, but neither of these things should affect your view controller's public interface. You want the flexibility to be able to change the visual design of your view controller without changing the class declaration that defines how other controllers collaborate with it.

## Use a Storyboard to Implement Your View Controller

You might consider whether or not to use a storyboard as an implementation detail, but the approach you take affects how you implement the view controller and how other objects collaborate with it. You always use a storyboard unless you have a strong reason not to.

When you use storyboards:

- iOS usually instantiates your view controller for you automatically.
- To finish instantiating it, you override its `awakeFromNib` method.
- Other objects configure it through its properties.
- You create its view hierarchy and other related objects in Interface Builder. These objects are loaded automatically when the view is needed.
- Relationships with other view controllers are created in the storyboard.

If you design your view controller to be used programmatically:

- The view controller is instantiated by allocating and initializing it.
- You create an custom initialization method to initialize the view controller.
- Other objects configure the view controller using its initialization method and by configuring its properties.
- You override the `loadView` method to programmatically create and configure its view hierarchy.
- Relationships with other view controllers are created by writing code.

## Know When Your Controller Is Instantiated

Knowing when your view controller is instantiated usually implies other details for how your app operates. For example, you might know that your view controller is always instantiated by the same object. Often the objects that instantiate view controllers are themselves view controllers; this is almost always the case in an app that uses storyboards. In any case, knowing when, why, and by what object your view controller is instantiated gives you insight into the information exchanged between your view controller and the object that created it.

## Know What Data Your View Controller Shows and Returns

When you answer these two questions, you are working to understand the data model for your app and also whether that data needs to be exchanged between your view controllers.

Here are some common patterns you should expect to see in your view controllers:

- The view controller receives data from another controller and displays it, without offering a way to edit it. No data is returned.
- The view controller allows the user to enter new data. After the user finishes editing the data, it sends the new data to another controller.
- The view controller receives data from another controller and allows the user to edit it. After the user finishes editing the data, it sends the new data to another controller.
- The view controller doesn't send or receive data. Instead, it shows static views.
- The view controller doesn't send or receive data. Instead, its implementation loads its data without exposing this mechanism to other view controllers. For example, the `GKAchievementViewController` class has built-in functionality to determine which player is authenticated on the device. It also knows how to load that player's data from Game Center. The presenting view controller does not need to know what data is loaded or how it was loaded.

You are not constrained to use only these designs.

When data travels into or out of your view controller, consider defining a data model class to hold the data to be transferred to the new controller. For example, in *Your Second iOS App: Storyboards*, the master controller uses a `BirdSighting` object to send data associated with a sighting to the detail controller. Using an object for this makes it easier to update the data to include additional properties without changing the method signatures in your controller classes.

## Know What Tasks Your Controller Allows the User to Perform

Some view controllers allow users to view, create, or edit data. Other view controllers allow users to navigate to other screens of content. And some allow users to perform tasks provided by the view controller. For example, the `MFMailComposeViewController` class allows a user to compose *and send* emails to other users. It handles the low-level details of sending mail messages.

As you determine which tasks your view controller performs, decide how much control over those tasks your view controller exposes to other controllers. Many view controllers can perform tasks without exposing configuration data to other controllers. For example, the `GKAchievementViewController` class displays achievements to the user without exposing any properties to configure how it loads or presents its data. The `MFMailComposeViewController` class presents a slightly different scenario by exposing some properties that another controller can use to configure the initial content it displays. After that, a user can edit the content and send the email message without giving other controller objects a chance to affect that process.

### Know How Your View Controller Is Displayed Onscreen

Some view controllers are designed to be root view controllers. Others expect to be presented by another view controller or placed in a container controller. Occasionally, you design controllers that can be displayed in multiple ways. For example, a split view controller's master view is displayed in the split view in landscape mode and in a popover control in portrait mode.

Knowing how your view controller is displayed gives you insight into how its view is sized and placed onscreen. It also affects other areas, such as determining what other controllers your view controller collaborates with.

### Know How Your Controller Collaborates with Other Controllers

By this point, you already know some things about collaboration. For example, if your view controller is instantiated from a segue, then it probably collaborates with the source view controller that configures it. And if your controller is a child of a container, then it collaborates with the container. But there are relationships in the other direction as well. For example, your view controller might defer some of its work and hand it off to another view controller. It might even exchange data with an existing view controller.

With all of these connections, your view controller provides an interface that other controllers use, or it is aware of other controllers and it uses their interfaces. These connections are essential to providing a seamless experience, but they also represent design challenges because they introduce dependencies between classes in your app. Dependencies are a problem because they make it more difficult to change any one class in isolation from the other classes that make up your app. For this reason, you need to balance the needs of your app now against the potential need to keep your app design flexible enough to change later.

## Examples of Common View Controller Designs

Designing a new view controller can be challenging. It helps to look at existing designs and understand what they do and why. This next section talks about some common view controller styles used in iOS apps. Each example includes a description of the role the view controller plays, a brief description of how it works at a high level, and one possible list of answers to the design questions listed above.

### Example: Game Title Screen

#### Mission Statement

A view controller that allows the user to select between different styles of game play.

## Description

When a game is launched, it rarely jumps right into the actual game. Instead, it displays a title screen that identifies the game and presents a set of game variants to the player. For example, a game might offer buttons that allow a player to start a single player or multiplayer game. When the user selects one of the options, the app configures itself appropriately and launches into its gameplay.

A title screen is interesting specifically because its contents are static; they don't need data from another controller. As such, this view controller is almost entirely self-sufficient. It does, however, know about other view controllers, because it instantiates other view controllers to launch its gameplay.

## Design

- **Are you using a storyboard to implement the view controller?** Yes.
- **When is it instantiated?** This view controller is the initial scene in the main storyboard.
- **What data does it show?** This class displays preconfigured controls and images; it does not present user data. It does not include configurable properties.
- **What tasks does it perform?** When a user taps on a button, it triggers a segue to instantiate another view controller. Each segue is identified so that the appropriate game play can be configured.
- **How is its view displayed onscreen?** It is installed automatically as the root view controller of the window.
- **How does it collaborate with other view controllers?** It instantiates another view controller to present a gameplay screen. When gameplay ends, the other view controller sends a message to the title screen controller to inform it that play has ended. The title screen controller then dismisses the other view controller.

## Alternative Design Considerations

The default answers assume that no user data is displayed. Some games include player data to configure the views or controls. For example:

- You might want the view controller to display the user's Game Center alias.
- You might want it to enable or disable buttons based on whether the device is connected to Game Center.
- You might want it to enable or disable buttons based on In-App Purchase items the user has purchased.

When these additional items are added to the design, the view controller takes on a more traditional role. It might receive data objects or data controllers from the app delegate so that it can query and update this state as necessary. Or, as it is the root view controller for the window, you might simply implement those behaviors directly in the title screen controller. The actual design likely depends on how flexible you need your code to be.

# Example: Master View Controller

## Mission Statement

The initial view controller of a navigation controller, used to display a list of the app's available data objects.

## Description

A master view controller is a very common part of a navigation-based app. For example, *Your Second iOS App: Storyboards* uses a master view to display the list of bird sightings. When a user selects a sighting from the list, the master view controller pushes a new detail controller onto the screen.

Because this view controller displays a list of items, it subclasses `UITableViewController` instead of `UIViewController`.

## Design

- **Are you using a storyboard to implement the view controller?** Yes.
- **When is it instantiated?** As the root view controller of a navigation controller, it is instantiated at the same time as its parent.
- **What data does it show?** A high-level view of the app's data. It implements properties that the app delegate uses to provide data to it. For example, the bird watching app provides a custom data controller object to the master view controller.
- **What tasks does it perform?** It implements an Add button to allow users to create new records.
- **How is its view displayed onscreen?** It is a child of a navigation controller.
- **How does it collaborate with other view controllers?** When the user taps on an item in the list, it uses a push segue to show a detail controller. When the user taps on the Add button, it uses a modal segue to present a new view controller that edits a new record. It receives data back from this modal view controller and sends this data to the data controller to create a new bird sighting.

## Alternative Design Considerations

A navigation controller and an initial view controller is used when building an iPhone app. When designing the same app for the iPad, the master view controller is a child of a split view controller instead. Most other design decisions stay the same.

# Example: Detail View Controller

## Mission Statement

A controller pushed onto a navigation stack to display the details for a list item selected from the master view controller.

## Description

The detail view controller represents a more detailed view of a list item displayed by the master view controller. As with the master view controller, the list appears inside a navigation bar interface. When the user finishes viewing the item they click a button in the navigation bar to return to the master view.

*Your Second iOS App: Storyboards* uses the `UITableViewController` class to implement its detail view. It uses a static table cells, each of which accesses one piece of the bird sighting data. A static table view is a good way to implement this design.

## Design

- **Are you using a storyboard to implement the view controller?** Yes.
- **When is it instantiated?** It is instantiated by a push segue from the master view controller.
- **What data does it show?** It shows the data stored in a custom data object. It declares properties configured by the source view controller to provide this data.
- **What tasks does it perform?** It allows the user to dismiss the view.
- **How is its view displayed onscreen?** It is a child of a navigation controller.
- **How does it collaborate with other view controllers?** It receives data from the master view controller.

## Alternative Design Considerations

A navigation controller is most often used when building an iPhone app. When designing the same app for the iPad, the detail view controller is a child of a split view controller instead. Many of the other implementation details stay the same.

If your app needs more custom view behavior, it might subclass the `UIViewController` class and implement its own custom view hierarchy.

## Example: Mail Compose View Controller

### Mission Statement

A view controller that allows the user to compose and send an email.

### Description

The Message UI framework provides the `MFMailComposeViewController` class. This class allows a user to compose and send an email. This view controller is interesting because it does more than simply show or edit data—it actually sends the email.

Another interesting design choice in this class is that it allows an app to provide an initial configuration for the email message. After the initial configuration has been presented, the user can override these choices before sending the mail.

### Design

- **Are you using a storyboard to implement the view controller?** No.
- **When is it instantiated?** It is instantiated programmatically.
- **What data does it show?** It shows the various parts of an email message, including a recipients list, title, attachments and the email message itself. The class provides properties that allow another view controller to preconfigure the email message.
- **What tasks does it perform?** It sends email.
- **How is its view displayed onscreen?** The view controller is presented by another view controller.
- **How does it collaborate with other view controllers?** It returns status information to its delegate. This status allows the presenting view controller to know whether an email was sent.

# Implementation Checklist for Custom Content View Controllers

For any custom content view controllers you create, there are a few tasks that you must have your view controller handle:

- You must configure the view to be loaded by your view controller.

  Your custom class may need to override specific methods to manage how its view hierarchy is loaded and unloaded. These same methods might manage other resources that are created at the same time. See "Resource Management in View Controllers" (page 56).

- You must decide which device orientations your view controller supports and how it reacts to a change in device orientation; see "Supporting Multiple Interface Orientations" (page 74).

As you implement your view controller, you will likely discover that you need to define action methods or outlets to use with its views. For example, if the 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 probably want those controls to call an associated action method on the view controller. 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:

- Declared properties pointing to the objects containing the data to be displayed by the corresponding views
- Public methods and properties that expose your view controller's custom behavior to other view controllers
- Outlets pointing to views in the view hierarchy with which your view controller must interact
- Action methods that perform tasks associated with buttons and other controls in the view hierarchy

**Important:** Clients of your view controller class do not need to know what views your view controller displays or what actions those views might trigger. Whenever possible, outlets and actions should be declared in a category inside your class's implementation file. For example, if your class is named `MyViewController`, you implement the category by adding the following declaration to `MyViewController.m`:

```
@interface MyViewController()
// Outlets and actions here.
@end
@implementation MyViewController
// Implementation of the privately declared category must go here.
@end
```

When you declare a category without a name, the properties and actions must be implemented in the same implementation block as any methods or properties declared in your public interface. The outlets and actions defined in the private category are visible to Interface Builder, but not to other classes in your app. This strategy allows you to gain the benefits of Interface Builder without exposing your class's secrets.

If another class needs access to your view controller's functionality, add public methods and properties to access this functionality instead.

# Resource Management in View Controllers

View controllers are an essential part of managing your app's resources. View controllers allow you to break your app up into multiple parts and instantiate only the parts that are needed. But more than that, a view controller itself manages different resources and instantiates them at different times. For example, a view controller's view hierarchy is instantiated only when the view is accessed; typically, this occurs only when the view is displayed on screen. If multiple view controllers are pushed onto a navigation stack at the same time, only the topmost view controller's contents are visible, which means only its views are accessed. Similarly, if a view controller is not presented by a navigation controller, it does not need to instantiate its navigation item. By deferring most resource allocation until it is needed, view controllers use less resources.

When memory available to the app runs low, all view controllers are automatically notified by the system. This allows the view controller to purge caches and other objects that can be easily recreated later when memory is more plentiful. The exact behavior varies depending on which version of iOS your app is running on, and this has implications for your view controller design.

Carefully managing the resources associated with your view controllers is critical to making your app run efficiently. You should also prefer lazy allocation; objects that are expensive to create or maintain should be allocated later and only when needed. For this reason, your view controllers should separate objects needed throughout the lifetime of the view controller from objects that are only necessary some of the time. When your view controller receives a low-memory warning, it should be prepared to reduce its memory usage if it is not visible onscreen.

## Initializing a View Controller

When a view controller is first instantiated, it creates or loads objects it needs through its lifetime. It should not create its view hierarchy or objects associated with displaying content. It should focus on data objects and objects needed to implement its critical behaviors.

### Initializing a View Controller Loaded from a Storyboard

When you create a view controller in a storyboard, the attributes you configure in Interface Builder are serialized into an archive. Later, when the view controller is instantiated, this archive is loaded into memory and processed. The result is a set of objects whose attributes match those you set in Interface Builder. The archive is loaded

by calling the view controller's `initWithCoder:` method. Then, the `awakeFromNib` method is called on any object that implements that method. You use this method to perform any configuration steps that require other objects to already be instantiated.

For more on archiving and archiving, see *Archives and Serializations Programming Guide*.

### Initializing View Controllers Programmatically

If a view controller allocates its resources programmatically, create a custom initialization method that is specific to your view controller. This method should call the super class's `init` method and then perform any class specific initialization.

In general, do not write complex initialization methods. Instead, implement a simple initialization method and then provide properties for clients of your view controller to configure its behaviors.

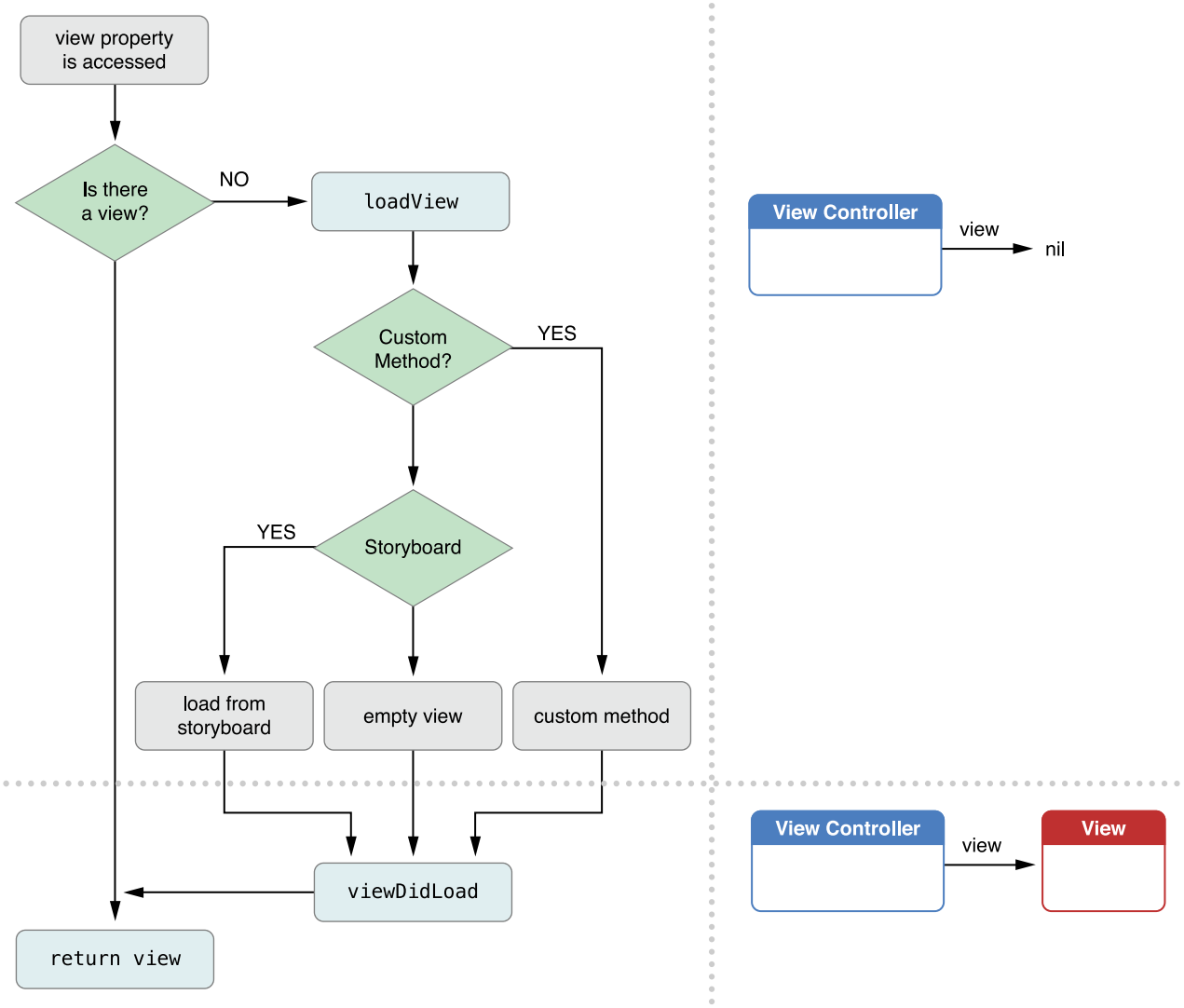
## A View Controller Instantiates Its View Hierarchy When Its View is Accessed

Whenever some part of your app asks the view controller for its view object and that object is not currently in memory, the view controller loads the view hierarchy into memory and stores it in its `view` property for future reference. The steps that occur during the load cycle are:

1. The view controller calls its `loadView` method. The default implementation of the `loadView` method does one of two things:
    - If the view controller is associated with a storyboard, it loads the views from the storyboard.
    - If the view controller is not associated with a storyboard, an empty `UIView` object is created and assigned to the `view` property.
2. The view controller calls its `viewDidLoad` method, which enables your subclass to perform any additional load-time tasks.

Figure 4-1 shows a visual representation of the load cycle, including several of the methods that are called. Your app can override both the `loadView` and the `viewDidLoad` methods as needed to facilitate the behavior you want for your view controller. For example, if your app does not use storyboards but you want additional views to be added to the view hierarchy, you override the `loadView` method to instantiate these views programatically.

**Figure 4-1** Loading a view into memory



# Loading a View Controller's View from a Storyboard

Most view controllers load their view from an associated storyboard. The advantage of using storyboards is that they allow you to lay out and configure your views graphically, making it easier and faster to adjust your layout. You can iterate quickly through different versions of your user interface to end up with a polished and refined design.

## Creating the View in Interface Builder

Interface Builder is part of Xcode and provides an intuitive way to create and configure the views for your view controllers. Using Interface Builder, 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 storyboard file, which stores the collection of objects you assembled along with information about all the customizations you made.

### Configuring the View Display Attributes in Interface Builder

To help you layout 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. If the controller is connected to container controllers in the storyboard, it can infer these settings from the container, making it easier to see exactly how it should appear at runtime.

### Configuring Actions and Outlets for Your View Controller

Using Interface Builder, you create connections between the views in your interface and your view controller.

Listing 4-1 shows the declaration of a custom `MyViewController` class's two custom outlets (designated by the `IBOutlet` keyword) and a single action method (designated by the `IBAction` return type). The declarations are made in a category inside the implementation file. The outlets store references to a button and a text field in the storyboard, while the action method responds to taps in the button.

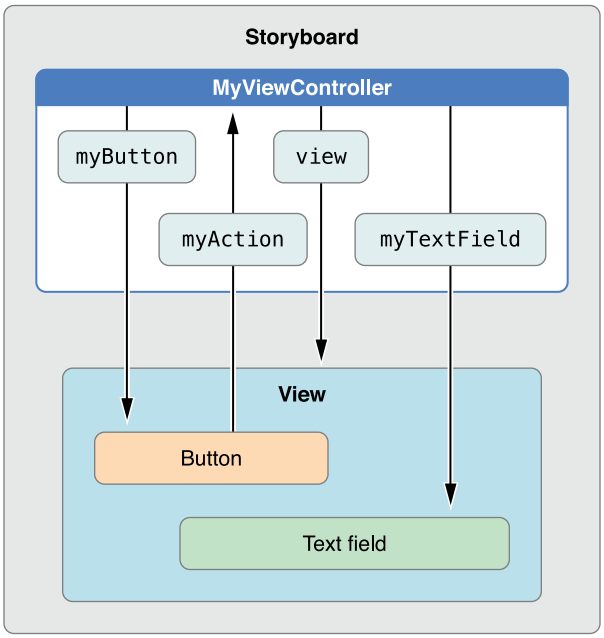
**Listing 4-1** Custom view controller class declaration

```
@interface MyViewController()
@property (nonatomic) IBOutlet id myButton;
@property (nonatomic) IBOutlet id myTextField;

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

Figure 4-2 shows the connections you would create among the objects in such a `MyViewController` class.

**Figure 4-2** Connections in the storyboard




When the previously configured `MyViewController` class is created and presented, the view controller infrastructure automatically loads the views from the storyboard 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 storyboards so powerful.

## Creating a View Programmatically

If you prefer to create views programmatically, instead of using a storyboard, you do so by overriding your view controller's `loadView` method. Your implementation of this method should do the following:

1. Create a root view object.

   The root view contains all other views associated with your view controller. You typically define the frame for this view to match the size of the app window, which itself should fill the screen. However, the frame is adjusted based on how your view controller is displayed. See "Resizing the View Controller's Views" (page 69).

   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 additional subviews and add them to the root view.

For each view, you should:

a. Create and initialize the view.

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

3. If you are using auto layout, assign sufficient constraints to each of the views you just created to control the position and size of your views. Otherwise, implement the `viewWillLayoutSubviews` and `viewDidLayoutSubviews` methods to adjust the frames of the subviews in the view hierarchy. See "Resizing the View Controller's Views" (page 69).

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

Listing 4-2 shows an example implementation of the `loadView` method. This method creates a pair of custom views in a view hierarchy and assigns them to the view controller.

**Listing 4-2** Creating views programmatically

```
- (void)loadView
{
    CGRect applicationFrame = [[UIScreen mainScreen] applicationFrame];
    UIView *contentView = [[UIView alloc] initWithFrame:applicationFrame];
    contentView.backgroundColor = [UIColor blackColor];
    self.view = contentView;

    levelView = [[LevelView alloc] initWithFrame:applicationFrame
viewController:self];
    [self.view addSubview:levelView];
}
```

**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 usually just wastes 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 "A View Controller Instantiates Its View Hierarchy When Its View is Accessed" (page 57).

# Managing Memory Efficiently

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

- How to allocate memory efficiently

- When and how to release memory

Although some aspects of memory allocation are strictly yours to decide, the `UIViewController` class provides some methods that usually have some connection to memory management tasks. Table 4-1 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 4-1** 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 using storyboards, the views are loaded automatically from the storyboard file. |
| Creating custom objects | Custom properties and methods | Although you are free to use other designs, consider using a pattern similar the `loadView` method. Create a property that holds the object and a matched method to initialize it. When the property is read and its value is `nil`, call the associated load method. |
| Allocating or loading data to be displayed in your view | `viewDidLoad` | Data objects are typically provided by configuring your view controller's properties. Any additional data objects your view controller wants to create should be done by overriding the `viewDidLoad` method. By the time this method is called, your view objects are guaranteed to exist and to be in a known good state. |
| Responding to low-memory notifications | `didReceiveMemory–Warning` | Use this method to deallocate all noncritical objects associated with your view controller. On iOS 6, you can also use this method to release references to view objects. |

| Task | Methods | Discussion |
| --- | --- | --- |
| Releasing critical data structures required by your view controller | `dealloc` | Override this method only to perform any last-minute cleanup of your view controller class. Objects stored in instance variables and properties are automatically released; you do not need to release them explicitly. |

## On iOS 6 and Later, a View Controller Unloads Its Own Views When Desired

The default behavior for a view controller is to load its view hierarchy when the `view` property is first accessed and thereafter keep it in memory until the view controller is disposed of. The memory used by a view to draw itself onscreen is potentially quite large. However, the system automatically releases these expensive resources when the view is not attached to a window. The remaining memory used by most views is small enough that it is not worth it for the system to automatically purge and recreate the view hierarchy.

You can explicitly release the view hierarchy if that additional memory is necessary for your app. Listing 4-3 overrides the `didReceiveMemoryWarning` method to accomplish this. First, is calls the superclass's implementation to get any required default behavior. Then, it cleans up the view controller's resources. Finally, it tests to see if the view controller's view is not onscreen. If the view is associated with a window, then it cleans up any of the view controller's strong references to the view and its subviews. If the views stored data that needs to be recreated, the implementation of this method should save that data before releasing any of the references to those views.

**Listing 4-3** Releasing the views of a view controller not visible on screen

```
- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
   // Add code to clean up any of your own resources that are no longer necessary.
    if ([self.view window] == nil)
    {
        // Add code to preserve data stored in the views that might be
        // needed later.

        // Add code to clean up other strong references to the view in
        // the view hierarchy.
        self.view = nil;
    }
```

The next time the `view` property is accessed, the view is reloaded exactly as it was the first time.

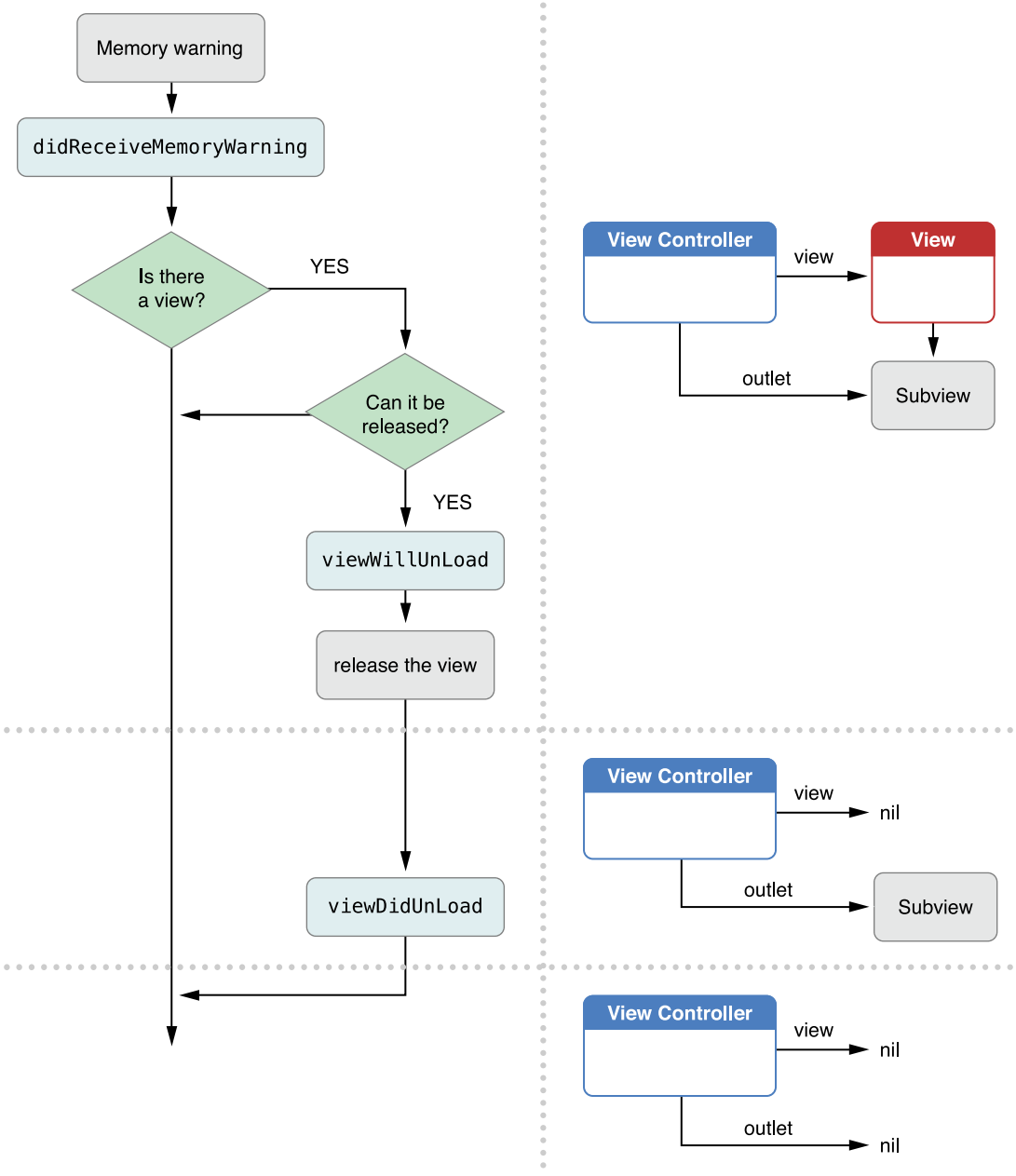
## On iOS 5 and Earlier, the System May Unload Views When Memory Is Low

In earlier versions of iOS, the system automatically attempts to unload a view controller's views when memory is low. The steps that occur during the unload cycle are as follows:

1. The app 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 memory or objects that your view controller object no longer needs. You must call `super` at some point in your implementation to ensure that the default implementation runs. On iOS 5 and earlier, the default implementation attempts to release the view. On iOS 6 and later, the default implementation exits.
3. If the view cannot be safely released (for example, it is visible onscreen), the default implementation exits.
4. The view controller calls its `viewWillUnload` method. A subclass typically overrides this method when it needs to save any view properties before the views are destroyed.
5. It sets its `view` property to `nil`.
6. The view controller calls its `viewDidUnload` method. A subclass typically overrides this method to release any strong references it has to those views.

Figure 4-3 shows a visual representation of the unload cycle for a view controller.

**Figure 4-3** Unloading a view from memory



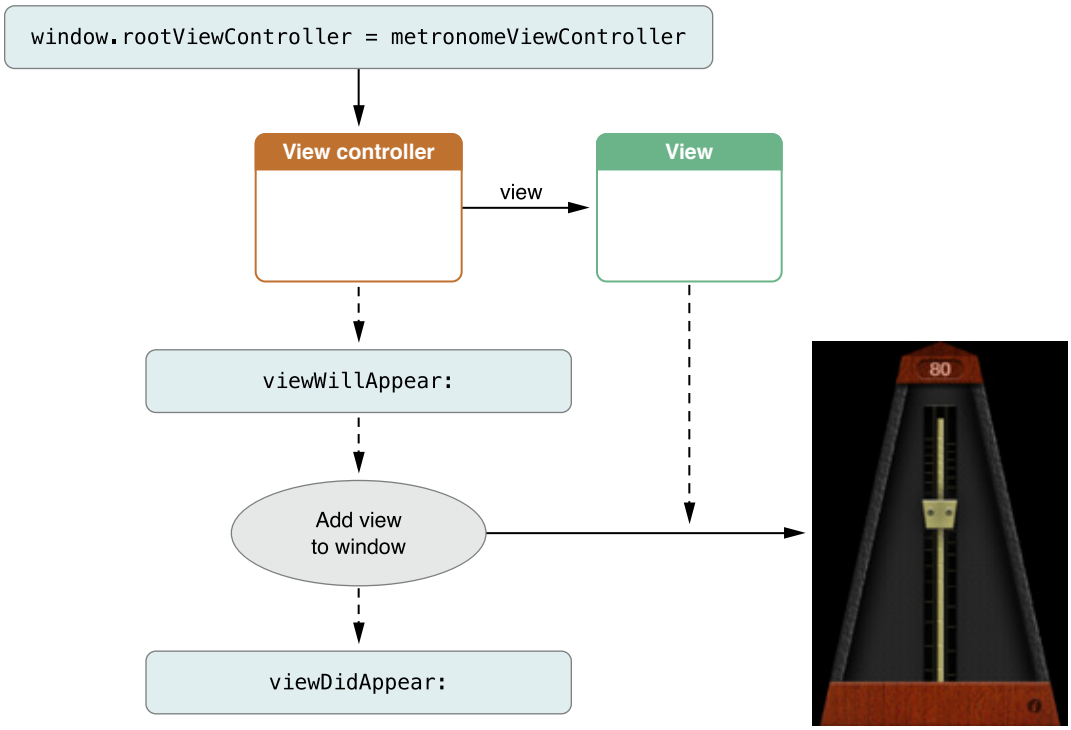
# Responding to Display-Related Notifications

When the visibility of a view controller's view changes, the view controller calls some built-in methods to notify subclasses of the changes. You can override these methods to override how your subclass reacts to the change. For example, you can 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.

## Responding When a View Appears

Figure 5-1 shows the sequence of events that occurs when a view controller's view is added to a window's view hierarchy. The `viewWillAppear:` and `viewDidAppear:` methods give subclasses a chance to perform any additional actions related to the appearance of the view.

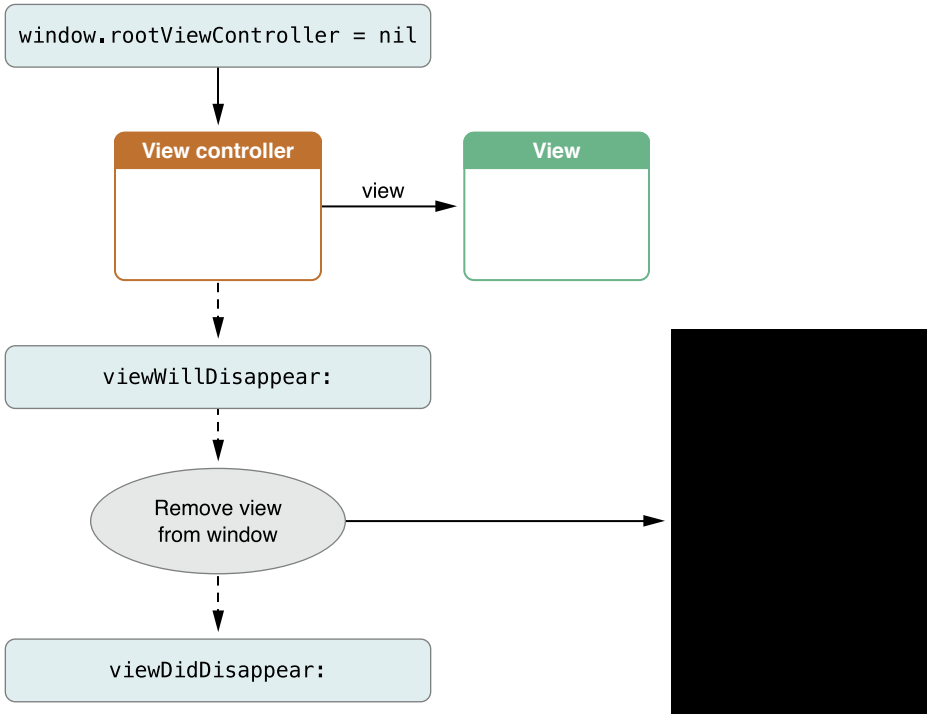
**Figure 5-1** Responding to the appearance of a view



# Responding When a View Disappears

Figure 5-2 shows the sequence of events that occurs when a view is removed from its window. 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 5-2** Responding to the disappearance of a view




# Determining Why a View's Appearance Changed

Occasionally, it can be useful to know why a view is appearing or disappearing. For example, you might want to know whether a view appeared because it was just added to a container or whether it appeared because some other content that obscured it was removed. This particular example often appears when using navigation controllers; your content controller's view may appear because the view controller was just pushed onto the navigation stack or it might appear because controllers previously above it were popped from the stack.

The `UIViewController` class provides methods your view controller can call to determine why the appearance change occurred. Table 5-1 describes the methods and their usage. These methods can be called from inside your implementation of the `viewWillAppear:`, `viewDidAppear:`, `viewWillDisappear:` and `viewDidDisappear:` methods.

**Table 5-1** Methods to call to determine why a view's appearance changed

| Method Name | Usage |
| --- | --- |
| `isMovingFromParentViewController` | You call this method inside your `viewWillDisappear:` and `viewDidDisappear:` methods to determine if the view controller's view is being hidden because the view controller was removed from its container view controller. |
| `isMovingToParentViewController` | You call this method inside your `viewWillAppear:` and `viewDidAppear:` methods to determine if the view controller's view is being shown because the view controller was just added to a container view controller. |
| `isBeingPresented` | You call this method inside your `viewWillAppear:` and `viewDidAppear:` methods to determine if the view controller's view is being shown because the view controller was just presented by another view controller. |
| `isBeingDismissed` | You call this method inside your `viewWillDisappear:` and `viewDidDisappear:` methods to determine if the view controller's view is being hidden because the view controller was just dismissed. |

# Resizing the View Controller's Views

A view controller owns its own view and manages the view's contents. In the process, the view controller also manages the view's subviews. But in most cases, the view's frame is not set directly by the view controller. Instead, the view's frame is determined by how the view controller's view is displayed. More directly, it is configured by the object used to display it. Other conditions in the app, such as the presence of the status bar, can also cause the frame to change. Because of this, your view controller should be prepared to adjust the contents of its view when the view's frame changes.

## A Window Sets the Frame of Its Root View Controller's View

The view associated with the window's root view controller gets a frame based on the characteristics of the window. The frame set by the window can change based on a number of factors:

- The frame of the window
- Whether or not the status bar is visible
- Whether or not the status bar is showing additional transient information (such as when a phone call is in progress)
- The orientation of the user interface (landscape or portrait)
- The value stored in the root view controller's `wantsFullScreenLayout` property

If your app displays the status bar, the view shrinks so that it does not underlap the status bar. After all, if the status bar is opaque, there is no way to see or interact with the content lying underneath it. However, if your app displays a translucent status bar, you can set the value of your view controller's `wantsFullScreenLayout` property to `YES` to allow your view to be displayed full screen. The status bar is drawn over the top of the view.

Full screen is useful when you want to maximize the amount of space available for displaying your content. When displaying content under the status bar, place 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 manually modify the `contentInset` property of your scroll view.

## A Container Sets the Frames of Its Children's Views

When a view controller is a child of a container view controller, its parent decides which children are visible. When it wants to show the view, it adds it as a subview in its own view hierarchy and sets its frame to fit it into its user interface. For example:

- A tab view controller reserves space at the bottom of its view for the tab bar. It sets the currently visible child's view to use the remainder of the space.
- A navigation view controller reserves space at the top for the navigation bar. If the currently visible child wants a navigation bar to be displayed, it also places a view at the bottom of the screen. The remainder of its view is given to the child to fill.

A child gets its frame from the parent all the way up to the root view controller, which gets its frame from the window.

## A Presented View Controller Uses a Presentation Context

When a view controller is presented by another view controller, the frame it receives is based on the presentation context used to display the view controller. See "Presentation Contexts Provide the Area Covered by the Presented View Controller" (page 95).

## A Popover Controller Sets the Size of the Displayed View

A view controller displayed by a popover controller can determine the size of its view's area by setting its own `contentSizeForViewInPopover` property value to the size it wants. If the popover controller sets its own `popoverContentSize` property to a different view size, its size value overrides the view controller's setting. To match the model used by other view controllers, use the popover controller's properties to control its size and position.

## How View Controllers Participate in the View Layout Process

When the size of a view controller's view changes, its subviews are repositioned to fit the new space available to them. The views in the controller's view hierarchy perform most of this work themselves through the use of layout constraints and autoresizing masks. However, the view controller is also called at various points so that it can participate in the process. Here's what happens:

1. The view controller's view is resized to the new size.

2. If autolayout is not in use, the views are resized according to their autoresizing masks.
3. The view controller's `viewWillLayoutSubviews` method is called.
4. The view's `layoutSubviews` method is called. If autolayout is used to configure the view hierarchy, it updates the layout constraints by executing the following steps:
    a. The view controller's `updateViewConstraints` method is called.
    b. The `UIViewController` class's implementation of the `updateViewConstraints` method calls the view's `updateConstraints` method.
    c. After the layout constraints are updated, a new layout is calculated and the views are repositioned.
5. The view controller's `viewDidLayoutSubviews` method is called.

Ideally, the views themselves perform all of the necessary work to reposition themselves, without requiring the view controller to participate in the process at all. Often, you can configure the layout entirely within Interface Builder. However, if the view controller adds and removes views dynamically, a static layout in Interface Builder may not be possible. In this case, the view controller is a good place to control the process, because often the views themselves only have a limited picture of the other views in the scene. Here are the best approaches to this in your view controller:

- Use layout constraints to automatically position the views (iOS 6 and later). You override `updateViewConstraints` to add any necessary layout constraints not already configured by the views. Your implementation of this method must call `[super updateViewConstraints]`.

  For more information on layout constraints, see *Auto Layout Guide*.
- Use a combination of autoresizing masks and code to manually position the views (iOS 5.x). You override `layoutSubviews` and use it to reposition any views whose positions cannot be set automatically through the use of resizing masks.

  For more information on the autoresizing properties of views and how they affect the view, see *View Programming Guide for iOS*.

# Using View Controllers in the Responder Chain

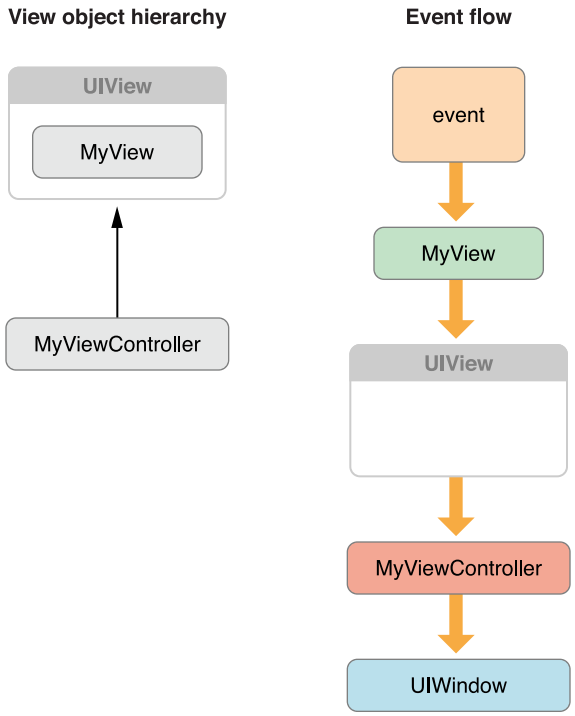
View controllers are descendants of the `UIResponder` class and are therefore capable of handling all sorts of events. When a view does not respond to a given event, it passes that event to its superview, traveling up the view hierarchy all the way to the root view. However, if any view in the chain is managed by a view controller, it passes the event to the view controller object before passing it up to the superview. In this way, the view controller can respond to events that are not handled by its views. If the view controller does not handle the event, that event moves on to the view's superview as usual.

## The Responder Chain Defines How Events Are Propagated to the App

Figure 7-1 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 in turn is 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 is the window object.

**Figure 7-1** Responder chain for view controllers

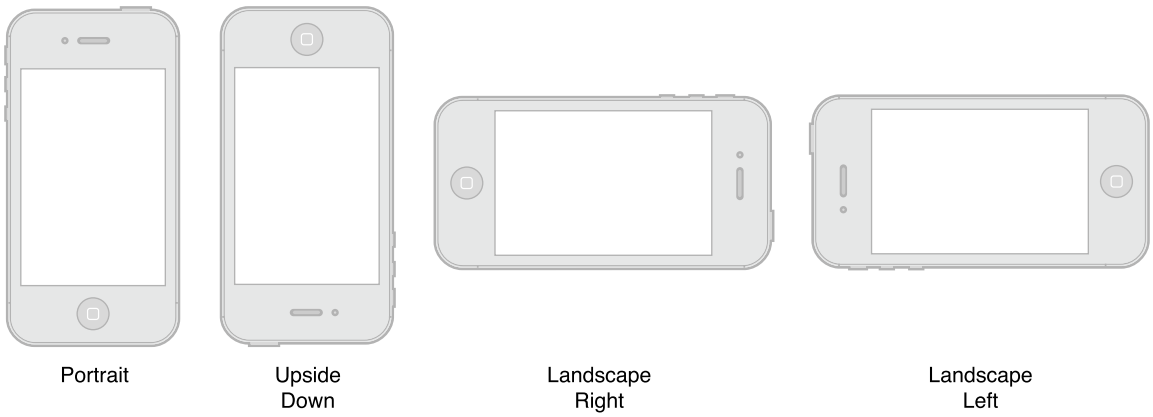



> **Note:** The message-passing relationship between a view controller and its view is managed privately by the view controller and cannot be programmatically modified by your app.

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 apps, see *Event Handling Guide for iOS*.

# Supporting Multiple Interface Orientations

The accelerometers in iOS–based devices make it possible to determine the current orientation of the device. By default, an app supports both portrait and landscape orientations. 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 listens for 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.




When the user interface rotates, the window is resized to match the new orientation. The window adjusts the frame of its root view controller to match the new size, and this size in turn is propagated down the view hierarchy to other views. Thus, the simplest way to support multiple orientations in your view controller is to configure its view hierarchy so that the positions of subviews are updated whenever its root view's frame changes. In most cases, you already need this behavior because other conditions may cause the view controller's visible area to change. For more information on configuring your view layout, see "Resizing the View Controller's Views" (page 69).

If the default behavior is not what you want for your app, you can take control over:

- The orientations supported by your app.
- How a rotation between two orientations is animated onscreen.

View controllers that do not fill the screen usually should not care about the orientation of the user interface. Instead, fill the area provided by the parent view controller. A root view controller (or a view controller presented full screen) is more likely to be interested in the orientation of the device.

# Controlling What Interface Orientations Are Supported (iOS 6)

When UIKit receives an orientation notification, it uses the `UIApplication` object and the root view controller to determine whether the new orientation is allowed. If both objects agree that the new orientation is supported, then the user interface is rotated to the new orientation. Otherwise the device orientation is ignored.

When a view controller is presented over the root view controller, the system behavior changes in two ways. First, the presented view controller is used instead of the root view controller when determining whether an orientation is supported. Second, the presented view controller can also provide a **preferred orientation**. If the view controller is presented full screen, the user interface is presented in the preferred orientation. The user is expected to see that the orientation is different from the device orientation and rotate the device. A preferred orientation is most often used when the content must be presented in the new orientation.

## Declaring a View Controller's Supported Interface Orientations

A view controller that acts as the root view controller of the main window or is presented full screen on the main window can declare what orientations it supports. It does this by overriding the `supportedInterfaceOrientations` method. By default, view controllers on devices that use the iPad idiom support all four orientations. On devices that use the iPhone idiom, all interface orientations but upside-down portrait are supported.

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 your app did this, you would still have to implement the necessary code to support all possible orientations, so you might as well just choose to support the orientation or not up front.

Listing 8-1 shows a fairly typical implementation of the `supportedInterfaceOrientations` method for a view controller that supports the portrait orientation and the landscape-left orientation. Your own implementation of this method should be just as simple.

**Listing 8-1** Implementing the `supportedInterfaceOrientations` method

```
- (NSUInteger)supportedInterfaceOrientations
{
    return UIInterfaceOrientationMaskPortrait |
UIInterfaceOrientationMaskLandscapeLeft;
}
```

## Dynamically Controlling Whether Rotation Occurs

Sometimes you may want to dynamically disable automatic rotation. For example, you might do this when you want to suppress rotation completely for a short period of time. You must temporarily disable orientation changes you want to manually control the position of the status bar (such as when you call the `setStatusBarOrientation:animated:` method).

If you want to temporarily disable automatic rotation, avoid manipulating the orientation masks to do this. Instead, override the `shouldAutorotate` method on the topmost view controller. This method is called before performing any autorotation. If it returns `NO`, then the rotation is suppressed.

## Declaring a Preferred Presentation Orientation

When a view controller is presented full-screen to show its content, sometimes the content appears best when viewed in a particular orientation in mind. If the content can only be displayed in that orientation, then you simply return that as the only orientation from your `supportedInterfaceOrientations` method. If the view controller supports multiple orientations but appears better in a different orientation, you can provide a preferred orientation by overriding the `preferredInterfaceOrientationForPresentation` method. Listing 8-2 shows an example used by a view controller whose content should be presented in landscape orientation. The preferred interface orientation must be one of the orientations supported by the view controller.

**Listing 8-2** Implementing the `preferredInterfaceOrientationForPresentation` method

```
- (UIInterfaceOrientation)preferredInterfaceOrientationForPresentation
{
    return UIInterfaceOrientationLandscapeLeft;
}
```

For more on presentation, see "Presenting View Controllers from Other View Controllers" (page 88).

## Declaring the App's Supported Interface Orientations

The easiest way to set an app's app's supported interface orientations is to edit the project's `Info.plist` file. As in the case of the view controller, you define which of the four interface orientations are permitted. For more information, see *Information Property List Key Reference*.

If you restrict the app's supported orientations, then those restrictions apply globally to all of the app's view controllers, even when your app uses system view controllers. At any given time, the mask of the topmost view controller is logically ANDed with the app's mask to determine what orientations are permitted. The result of this calculation must never be `0`. If it is, the system throws a `UIApplicationInvalidInterfaceOrientationException` exception.

Because the app's mask is applied globally, use it sparingly.

> **Important:** The combination of the app and the view controller's orientation masks must result in at least one useable orientation. An exception is thrown if there is no available orientation.

# Understanding the Rotation Process (iOS 5 and earlier)

On iOS 5 and earlier, a view controller can sometimes participate in the rotation process even when it isn't the topmost full-screen view controller. This generally occurs when a container view controller asks its children for their supported interface orientations. In practice, the ability for children to override the parents is rarely useful. With that in mind, you should consider emulating the iOS 6 behavior as much as possible in an app that must also support iOS 5:

- In a root view controller or a view controller that is presented full screen, choose a subset of interface orientations that make sense for your user interface.
- In a child controller, support all the default resolutions by designing an adaptable view layout.

## Declaring the Supported Interface Orientations

To declare your supported interface orientations, 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 8-3 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 8-3** 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 app 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.

## Responding to Orientation Changes in a Visible View Controller

When a rotation occurs, the view controllers play an integral part of the process. Visible view controllers are notified at various stages of the rotation to give them 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 app 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 (as described in "Creating an Alternate Landscape Interface" (page 80)).

The rotation methods are sent to the root view controller. The root view controller passes these events on as necessary to its children, and so on down the view controller hierarchy. Here is the sequence of events that occur when a rotation is triggered:

1. The window calls the root view controller's `willRotateToInterfaceOrientation:duration:` method.

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

2. The window adjusts the bounds of the view controller's view. This causes the view to layout its subviews, triggering the view controller's `viewWillLayoutSubviews` method. When this method runs, you can query the app object's `statusBarOrientation` property to determine the current user interface layout.

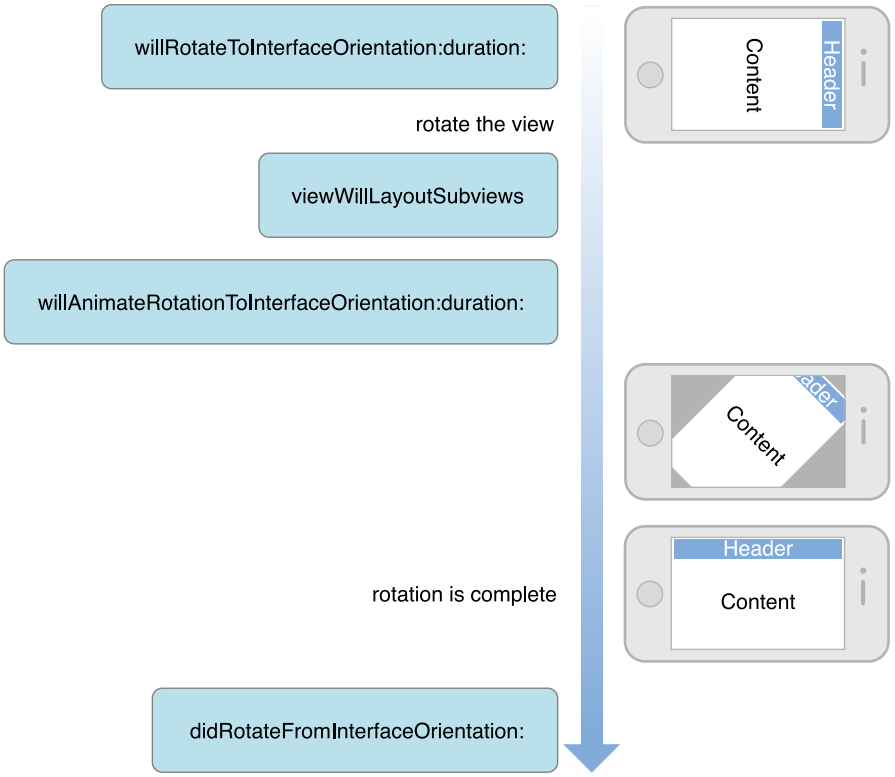
   See "How View Controllers Participate in the View Layout Process" (page 70).

3. The view controller's `willAnimateRotationToInterfaceOrientation:duration:` method is called. This method is called from within an animation block so that any property changes you make are animated at the same time as other animations that comprise the rotation.
4. The animation is executed.
5. The window calls the view controller's `didRotateFromInterfaceOrientation:` method.

   Container view controllers forward this message to the currently displayed content view controllers. This action 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 app.

Figure 8-1 shows a visual representation of the preceding steps. It also shows how the interface looks at various stages of the process.

**Figure 8-1** Processing an interface rotation



## Rotations May Occur When Your View Controller Is Hidden

If your view controller's contents are not onscreen when a rotation occurs, then it does not see the list of rotation messages. For example, consider the following sequence of events:

1. Your view controller presents another view controller's contents full screen.
2. The user rotates the device so that the user interface orientation changes.
3. Your app dismisses the presented view controller.

In this example, the presenting view controller was not visible when the rotation occurred, so it does not receive any rotation events. Instead, when it reappears, its views are simply resized and positioned using the normal view layout process. If your layout code needs to know the current orientation of the device, it can read the app object's `statusBarOrientation` property to determine the current orientation.

## 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.

To support an alternate landscape interface, you must do the following:

- Implement two view controller objects. One to present a portrait-only interface, and the other to 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. The primary view controller dismisses the alternate view controller when the orientation returns to the primary orientation.

Listing 8-4 shows the key methods you need to implement in a primary view controller that supports a portrait orientation. When the primary view controller is loaded from the storyboard, it 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 8-4** Presenting the landscape view controller

```
@implementation PortraitViewController
- (void)awakeFromNib
{
    isShowingLandscapeView = NO;
    [[UIDevice currentDevice] beginGeneratingDeviceOrientationNotifications];
    [[NSNotificationCenter defaultCenter] addObserver:self
                                 selector:@selector(orientationChanged:)
                                 name:UIDeviceOrientationDidChangeNotification
                                 object:nil];
}

- (void)orientationChanged:(NSNotification *)notification
{
    UIDeviceOrientation deviceOrientation = [UIDevice currentDevice].orientation;
    if (UIDeviceOrientationIsLandscape(deviceOrientation) &&
        !isShowingLandscapeView)
    {
        [self performSegueWithIdentifier:@"DisplayAlternateView" sender:self];
        isShowingLandscapeView = YES;
    }
    else if (UIDeviceOrientationIsPortrait(deviceOrientation) &&
             isShowingLandscapeView)
    {
        [self dismissViewControllerAnimated:YES completion:nil];
        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 no code 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 app contains a map view, save the visible map region value prior to beginning any 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. After 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 app's state information.** If your app 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.

# Accessibility from the View Controller's Perspective

Aside from managing a view's behavior, a view controller can also help control an app's accessibility. An accessible app is one that can be used by everyone, regardless of disability or physical impairment, while retaining its functionality and usability as a helpful tool.

To be accessible, an iOS app must supply information about its user interface elements to VoiceOver users. VoiceOver, a screen-reading technology designed to assist the visually impaired, speaks aloud text, images, and UI controls displayed the screen, so that people who cannot see can still interact with these elements. UIKit objects are accessible by default, but there are still things you can do from the view controller's perspective to address accessibility. At a high level, this means you should make sure that:

- Every user interface element users can interact with is accessible. This includes elements that merely supply information, such as static text, as well as controls that perform actions.
- All accessible elements supply accurate and helpful information.

In addition to these fundamentals, a view controller can enhance the VoiceOver user's experience by setting the position of the VoiceOver focus ring programmatically, responding to special VoiceOver gestures, and observing accessibility notifications.

## Moving the VoiceOver Cursor to a Specific Element

When the layout of a screen changes, the VoiceOver focus ring, also known as the *VoiceOver cursor*, resets its position to the first element displayed on the screen from left to right and top to bottom. You might decide to change the first element the VoiceOver cursor lands on when views are presented onscreen.

For example, when a navigation controller pushes a view controller onto the navigation stack, the VoiceOver cursor falls on the Back button of the navigation bar. Depending on your app, it might make more sense to move it to the heading of the navigation bar instead, or to any other element.

To do so, call `UIAccessibilityPostNotification` using both the notification `UIAccessibilityScreenChangedNotification` (which tells VoiceOver that the contents of the screen has changed) and the element you'd like to give focus to, as shown in Listing 9-1.

**Listing 9-1** Posting an accessibility notification can change the first element read aloud

```
@implementation MyViewController
- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];

    UIAccessibilityPostNotification(UIAccessibilityScreenChangedNotification,
                                    self.myFirstElement);
}
@end
```

If only the layout changes rather than the contents of the screen, such as when switching from portrait to landscape mode, use the notification `UIAccessibilityLayoutChangedNotification` instead of `UIAccessibilityScreenChangedNotification`.

> **Note:** Device rotation triggers a layout change, which resets the VoiceOver cursor's position.

## Responding to Special VoiceOver Gestures

There are special gestures that VoiceOver users can perform to trigger custom actions. These gestures are special because you are allowed to define their behavior, unlike standard VoiceOver gestures. You can detect the gestures by overriding certain methods in your views or view controllers.

A gesture first checks the view that has VoiceOver focus for instruction and continues up the responder chain until it finds an implementation of the corresponding VoiceOver gesture method. If no implementation is found, the system default action for that gesture is triggered. For example, the Magic Tap gesture plays and pauses music playback from the Music app if no Magic Tap implementation is found from the current view to the app delegate.

Although you can provide any custom logic you want, VoiceOver users expect the actions of these special gestures to follow certain guidelines. Like any gesture, your implementation of a VoiceOver gesture should follow a pattern so that interaction with an accessible app remains intuitive.

There are five special VoiceOver gestures:

- **Escape.** A two-finger Z-shaped gesture that dismisses a modal dialog, or goes back one level in a navigation hierarchy.

- **Magic Tap.** A two-finger double-tap that performs the most-intended action.
- **Three-Finger Scroll.** A three-finger swipe that scrolls content vertically or horizontally.
- **Increment and Decrement.** A one-finger swipe up or down that adds or subtracts a given value from an element with the adjustable trait. Elements with the Adjustable accessibility trait must implement these methods.

> **Note:** All special VoiceOver gesture methods return a Boolean value that determine whether to propagate through the responder chain. To halt propagation, return YES; otherwise, return NO.

## Escape

If you present a view that overlays content—such as a modal dialog or an alert—you should override the `accessibilityPerformEscape` method to dismiss the overlay. The function of the Escape gesture is like the function of the `Esc` key on a computer keyboard; it cancels a temporary dialog or sheet to reveal the main content.

Another use case to override the Escape gesture would be to go back up one level in a navigation hierarchy. `UINavigationController` implements this functionality by default. If you're designing your own kind of navigation controller, you should set the Escape gesture to traverse up one level of your navigation stack, because that is the functionality VoiceOver users expect.

## Magic Tap

The purpose of the Magic Tap gesture is to quickly perform an often-used or most-intended action. For example, in the Phone app, it picks up or hangs up a phone call. In the Clock app, it starts and stops the stopwatch. If you want an action to fire from a gesture regardless of the view the VoiceOver cursor is on, you should implement the `accessibilityPerformMagicTap` method in your view controller.

> **Note:** If you'd like the Magic Tap gesture to perform the same action from anywhere within your app, it is more appropriate to implement the `accessibilityPerformMagicTap` method in your app delegate.

## Three-Finger Scroll

The `accessibilityScroll:` method fires when a VoiceOver user performs a three-finger scroll. It accepts a `UIAccessibilityScrollDirection` parameter from which you can determine the direction of the scroll. If you have a custom scrolling view, it may be more appropriate to implement this on the view itself.

### Increment and Decrement

The `accessibilityIncrement` and `accessibilityDecrement` methods are required for elements with the adjustable trait and should be implemented on the views themselves.

## Observing Accessibility Notifications

You can listen for accessibility notifications to trigger callback methods. Under certain circumstances, UIKit fires accessibility notifications which your app can observe to extend its accessible functionality.

For example, if you listen for the notification `UIAccessibilityAnnouncementDidFinishNotification`, you can trigger a method to follow up the completion of VoiceOver's speech. Apple does this in the iBooks app. iBooks fires a notification when VoiceOver finishes speaking a line in a book that triggers the next line to be spoken. If it is the last line on the page, the logic in the callback tells iBooks to turn the page and continue reading as soon as the last line ends speaking. This allows for a line-by-line degree of granularity for navigating text while providing a seamless, uninterrupted reading experience.

To register as an observer for accessibility notifications, use the default notification center. Then create a method with the same name that you provide for the `selector` argument, as shown in Listing 9-2.

**Listing 9-2** Registering as an observer for accessibility notifications

```
@implementation MyViewController
- (void)viewDidLoad
{
    [super viewDidLoad];

    [[NSNotificationCenter defaultCenter]
        addObserver:self
           selector:@selector(didFinishAnnouncement:)
               name:UIAccessibilityAnnouncementDidFinishNotification
             object:nil];
}

- (void)didFinishAnnouncement:(NSNotification *)dict
{
    NSString *valueSpoken = [[dict userInfo]
objectForKey:UIAccessibilityAnnouncementKeyStringValue];
```

```
    NSString *wasSuccessful = [[dict userInfo]
objectForKey:UIAccessibilityAnnouncementKeyWasSuccessful];
    // ...
}
@end
```

`UIAccessibilityAnnouncementDidFinishNotification` expects an `NSNotification` dictionary as a parameter from which you can determine the value spoken and whether or not the speaking has completed uninterrupted. Speaking may become interrupted if the VoiceOver user performs the stop speech gesture or swipes to another element before the announcement finishes.

Another helpful notification to subscribe to is `UIAccessibilityVoiceOverStatusChanged`. It can detect when VoiceOver becomes toggled on or off. If VoiceOver is toggled outside of your app, you receive the notification when your app is brought back into the foreground. Because `UIAccessibilityVoiceOverStatusChanged` doesn't expect any parameters, the method in your selector doesn't need to append a trailing colon (`:`).

For a full list of possible notifications you can observe, consult "Notifications" in *UIAccessibility Protocol Reference* . Remember that you may only observe the notifications that can be posted by UIKit, which are `NSString` objects, and not notifications that can be posted by your app, which are of type `int`.

# Presenting View Controllers from Other View Controllers

The ability to present view controllers is a tool that you have at your disposal for interrupting the current workflow and displaying a new set of views. Most commonly, an app presents a view controller as a temporary interruption to obtain important information from the user. However, you can also use presented view controllers to implement alternate interfaces for your app at specific times.

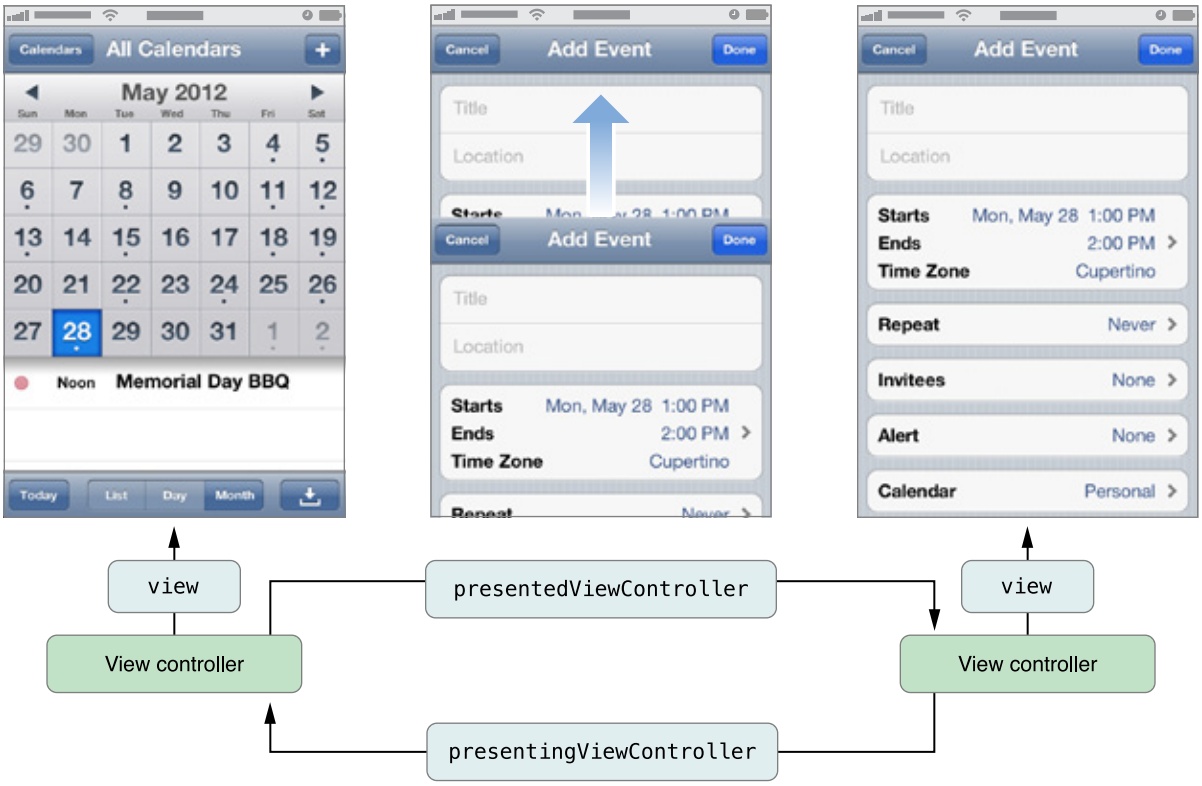
## How View Controllers Present Other View Controllers

A presented view controller is not a specific subclass of `UIViewController` (as `UITabBarController` or `UINavigationController` is). Instead, any view controller can be presented by your app. However, like tab bar and navigation controllers, you present view controllers when you want to convey a specific meaning about the relationship between the previous view hierarchy and the newly presented view hierarchy.

When you present a modal view controller, the system creates a 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 `presentedViewController` property to point to its presented view controller.

Similarly, the presented view controller updates its `presentingViewController` property to point back to the view controller that presented it. Figure 10-1 shows the relationship between the view controller managing the main screen in the Calendar app and the presented view controller used to create new events.

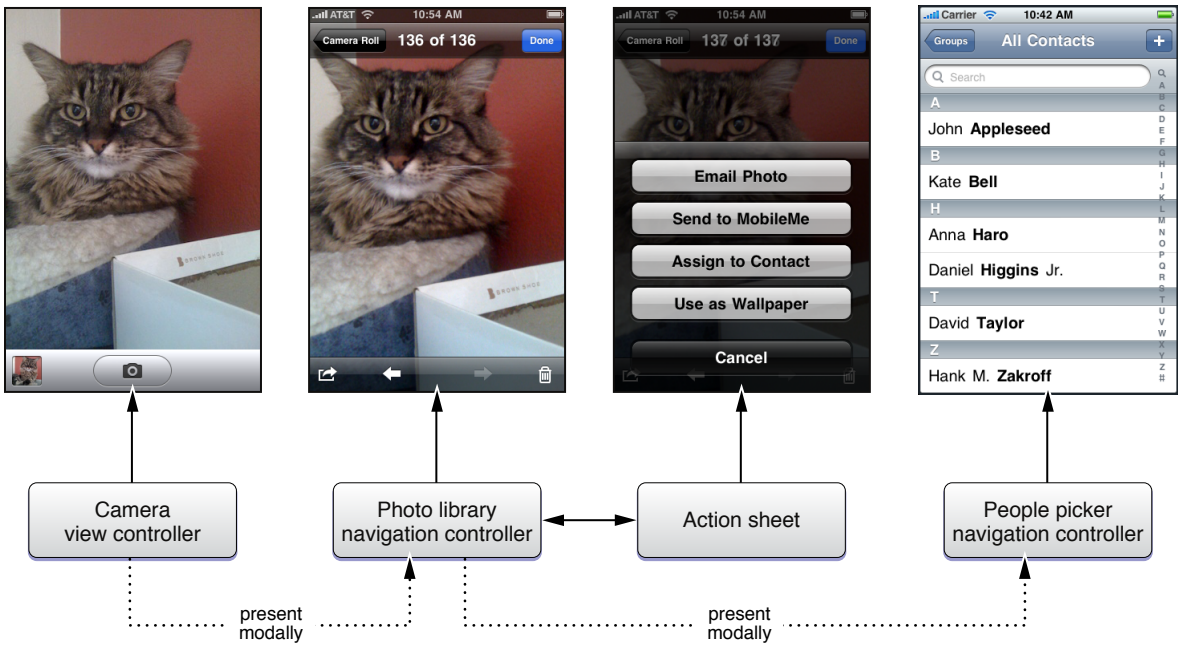
**Figure 10-1** Presented views in the Calendar app.




Any view controller object can present a single view controller at a time. This is true even for view controllers that were themselves presented by another view controller. In other words, you can chain presented view controllers together, presenting new view controllers on top of other view controllers as needed. Figure 10-2 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 app presents a 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 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 10-2** Creating a chain of modal view controllers



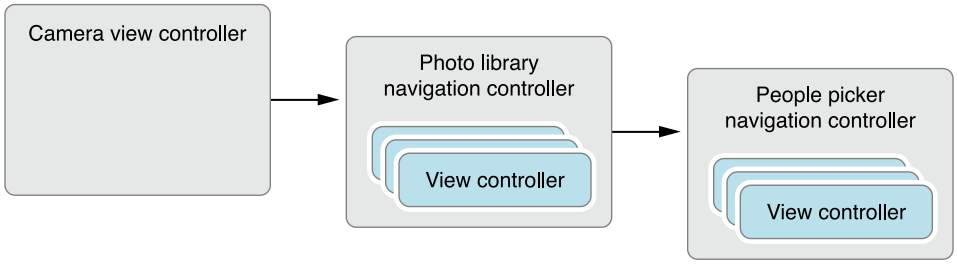

Each view controller in a chain of presented view controllers has pointers to the other objects surrounding it in the chain. In other words, a presented view controller that presents another view controller has valid objects in both its `presentingViewController` and `presentedViewController` properties. 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 can remove all objects in the chain by dismissing the first presented view controller. Dismissing a view controller dismisses not only that view controller but also any view controllers it presented.

In Figure 10-2 (page 90), a point worth noting is that the presented view controllers are both navigation controllers. You can present `UINavigationController` objects in the same way that you would present a content view controller.

When presenting a navigation controller, 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 present other view controllers, including other navigation controllers. Figure 10-3

shows more detail of the objects that are 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 content view controllers on its navigation stack.

**Figure 10-3** Presenting navigation controllers modally

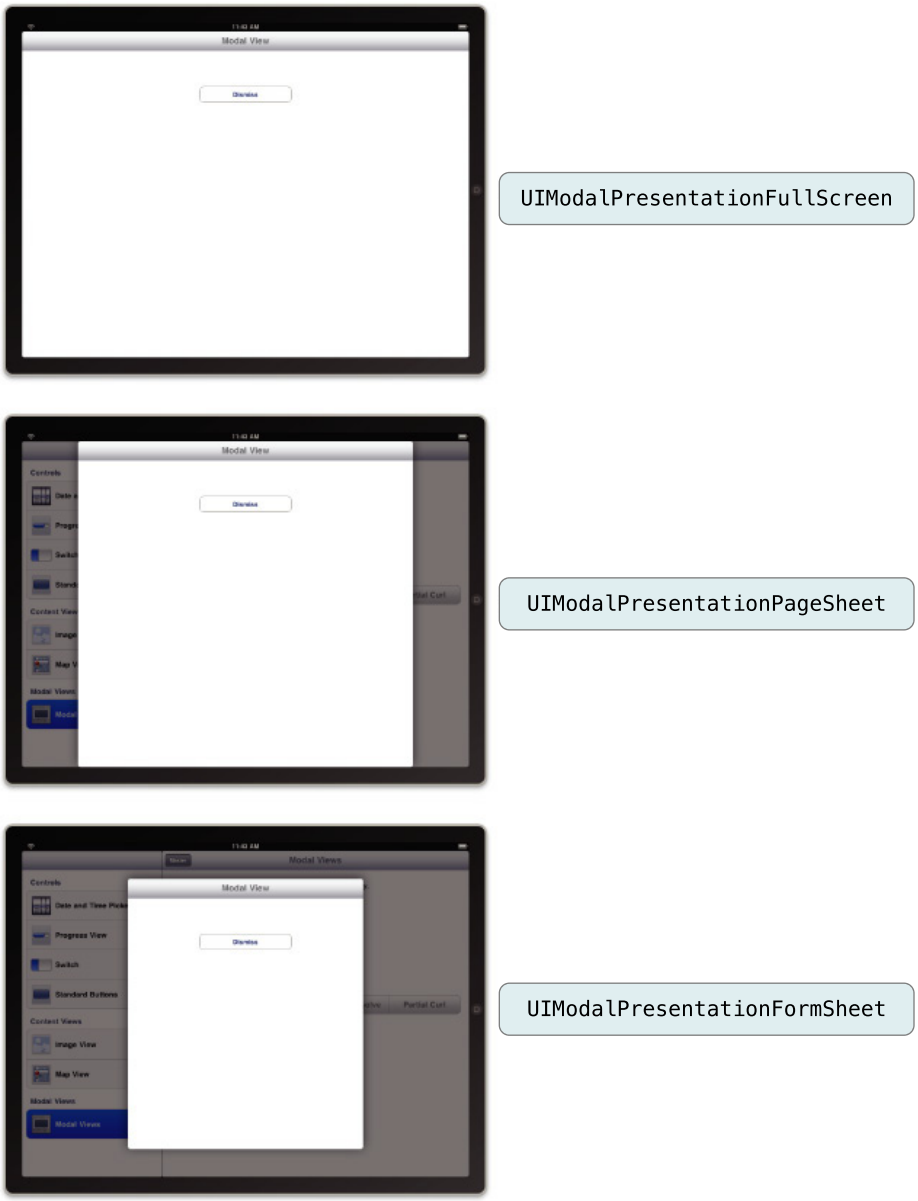



## Presentation Styles for Modal Views

For iPad apps, you can present content using several different styles. In iPhone apps, 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. Different options for this property allow you to present the view controller so that it fills all or only part of the screen.

Figure 10-4 shows the core presentation styles that are available. (The `UIModalPresentationCurrentContext` style lets a view controller adopt the presentation style of its parent.) In each presentation style, the dimmed areas show the underlying content but do not allow taps in that content. Therefore, unlike a popover, your presented views must still have controls that allow the user to dismiss the view.

**Figure 10-4** iPad presentation styles




For guidance on when to use the different presentation styles, see "Modal View" in *iOS Human Interface Guidelines*.

# Presenting a View Controller and Choosing a Transition Style

When a view controller is presented using a storyboard segue, it is automatically instantiated and presented. The presenting view controller can configure the destination view controller before it is presented. For more information, see "Configuring the Destination Controller When a Segue is Triggered" (page 101).

If you need to present a view controller programmatically, 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 to the view controller. Typically, the delegate is the presenting view controller. The delegate is used by the presented view controllers to notify the presenting view controller when it is ready to be dismissed. It may also communicate other information back to the delegate.
4. Call the `presentViewController:animated:completion:` method of the current view controller, passing in the view controller you want to present.

The `presentViewController:animated:completion:` method presents the view for the specified view controller object and configures the presenting-presented relationships between the new view controller and the current view controller. Unless you are restoring your app to some previous state, you usually want to animate the appearance of the new view controller. The transition style you should use depends on how you plan to use the presented view controller. Table 10-1 lists the transition styles you can assign to the `modalTransitionStyle` property of the presented view controller and describes how you might use each one.

**Table 10-1** 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.<br>For this style of transition, content view controllers should provide buttons to dismiss the view controller explicitly. Typically, these are a Done button and an optional Cancel button.<br>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 app temporarily. The most common usage for this style is to display settings that might change frequently, such as in the Stocks and Weather apps. These settings can be meant for the entire app or they can be specific to the current screen.<br>For this style of transition, you usually provide some sort of button to return the user to the normal running mode of your app. |

| Transition style | Usage |
| --- | --- |
| `UIModalTransitionStyleCrossDissolve` | Use this style to present an alternate interface when the device changes orientations. In such a case, your app is responsible for presenting and dismissing the alternate interface in response to orientation change notifications.<br>Media-based apps can also use this style to fade in screens displaying media content.<br>For an example of how to implement an alternate interface in response to device orientation changes, see "Creating an Alternate Landscape Interface" (page 80). |

Listing 10-1 shows how to present a view controller programmatically. When the user adds a new recipe, the app prompts the user for basic information about the recipe by presenting a navigation controller. 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 10-1** Presenting a view controller programmatically

```
- (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]
                       init];

   // 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.
   UINavigationController *navigationController = [[UINavigationController alloc]
                              initWithRootViewController:addController];
   [self presentViewController:navigationController animated:YES completion: nil];
}
```

When the user taps either the Done or the Cancel button from the new recipe entry interface, the app dismisses the view controller and returns the user to the main view. See "Dismissing a Presented View Controller" (page 95).

## Presentation Contexts Provide the Area Covered by the Presented View Controller

The area of the screen used to define the presentation area is determined by the presentation context. By default, the presentation context is provided by the root view controller, whose frame is used to define the frame of the presentation context. However, the presenting view controller, or any other ancestor in the view controller hierarchy, can choose to provide the presentation context instead. In that case, when another view controller provides the presentation context, its frame is used instead to determine the frame of the presented view. This flexibility allows you to limit the modal presentation to a smaller portion of the screen, leaving other content visible.

When a view controller is presented, iOS searches for a presentation context. It starts at the presenting view controller by reading its `definesPresentationContext` property. If the value of this property is `YES`, then the presenting view controller defines the presentation context. Otherwise, it continues up through the view controller hierarchy until a view controller returns `YES` or until it reaches the window's root view controller.

When a view controller defines a presentation context, it can also choose to define the presentation style. Normally, the presented view controller determines how it presented using its `modalTransitionStyle` property. A view controller that sets `definesPresentationContext` to `YES` can also set `providesPresentationContextTransitionStyle` to `YES`. If `providesPresentationContextTransitionStyle` is set to `YES`, iOS uses the presentation context's `modalPresentationStyle` to determine how the new view controller is presented.

## Dismissing a Presented View Controller

When it comes time to dismiss a presented view controller, the preferred approach is to let the presenting view controller dismiss it. In other words, whenever possible, the same view controller that presented the view controller should also take responsibility for dismissing it. Although there are several techniques for notifying the presenting view controller that its presented view controller should be dismissed, the preferred technique is delegation. For more information, see "Using Delegation to Communicate with Other Controllers" (page 103).

# Presenting Standard System View Controllers

A number of standard system view controllers are designed to be presented by your app. The basic rules for presenting these view controllers are the same as the rules for presenting your custom content view controllers. However, because your app 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 either 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 presented view controller.)

Table 10-2 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 10-2** Standard system view controllers

| Framework | View controllers |
| --- | --- |
| Address Book UI | `ABNewPersonViewController`<br>`ABPeoplePickerNavigationController`<br>`ABPersonViewController`<br>`ABUnknownPersonViewController` |
| Event Kit UI | `EKEventEditViewController`<br>`EKEventViewController` |
| Game Kit | `GKAchievementViewController`<br>`GKLeaderboardViewController`<br>`GKMatchmakerViewController`<br>`GKPeerPickerController`<br>`GKTurnBasedMatchmakerViewController` |
| Message UI | `MFMailComposeViewController`<br>`MFMessageComposeViewController` |
| Media Player | `MPMediaPickerController`<br>`MPMoviePlayerViewController` |

| Framework | View controllers |
| --- | --- |
| UIKit | `UIImagePickerController`<br>`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.)

# Coordinating Efforts Between View Controllers

Few iOS apps show only a single screenful of content. Instead, they show some content when first launched and then show and hide other content in response to user actions. These transitions provide a single unified user interface that display a lot of content, just not all at once.

By convention, smaller pieces of content are managed by different view controller classes. This coding convention allows you to create smaller and simpler controller classes that are easy to implement. However, dividing the work between multiple classes imposes additional requirements on your class designs. To maintain the illusion of a single interface, your view controllers must exchange messages and data to coordinate transitions from controller to another. Thus, even as your view controller classes look inwards to control views and perform the tasks assigned to them, they also look outwards to communicate with other collaborating view controllers.

## When Coordination Between View Controllers Occurs

Communication between view controllers is tied to the role those view controllers play in your app. It would be impossible to describe all of the possible interactions between view controllers, because the number and nature of these relationships is dependent on the design of your app. However, it is possible to describe when these interactions occur and to give some examples of the kinds of coordination that might take place in your app.

The lifetime of a view controller has three stages during which it might coordinate with other objects:

**View controller instantiation.** In this stage, when a view controller is created, an existing view controller or another object was responsible for its creation. Usually, this object knows why the view controller was created and what task it should perform. Thus, after a view controller is instantiated, this intent must be communicated to it.

The exact details of this initial configuration vary. Sometimes, the existing view controller passes data objects to the new controller. At other times, it may configure the presentation style for that, or establish lasting links between the two view controllers. These links allow further communication later in the view controller's lifetime.

**During the view controller's lifetime.** In this stage, some view controllers communicate with other view controllers during their lifetime. The recipient of these messages could be the view controller that created it, peers with similar lifetimes, or even a new view controller that it itself created. Here are a few common designs:

- The view controller sends notifications that the user performed a specific action. Because this is a notification, the object receiving this message is just being notified that something happened.
- The view controller sends data to another view controller. For example, a tab bar controller doesn't establish a built-in relationship between its children, but your app might establish such a relationship when the tabs are displaying the same data object, just in different ways. When a user leaves one tab, the view controller associated with that tab sends the selection information to the view controller about to be displayed. In return, the new view controller uses this data to configure its views so that the transition appears seamless. In this particular case, no new view controller is instantiated by the action. Instead, the two view controllers are peers with the same lifetime and can continue to coordinate as the user switches between them.
- A view controller sends messages to give another view controller authority over its actions. For example, if a view controller allows users to enter data, it might send messages to allow another controller to decide whether the data the user entered is valid. If the data is invalid, the view controller can disallow the user from accepting the invalid data or adjust its interface to display an error.

**View controller destruction.** In this stage, many view controllers send messages when their task completes. These messages are common because the convention is for the controller that created a view controller to also release it. Sometimes, these messages simply convey that the user finished the task. At other times, such as when the task being performed generated new data objects, the message communicates the new data back to another controller.

During a view controller's lifetime, it is common for it to exchange information with other view controllers. These messages are used to notify other controllers when things happen, send them data, or even ask them to exert control over the controller's activities.

## With Storyboards, a View Controller is Configured When It Is Instantiated

Storyboards provide direct support for configuring newly instantiated controllers before they are displayed. When a storyboard instantiates new view controllers automatically, it calls an object in your app to allow it to configure the new controller or to create links to or from the new controller. When your app first launches, the app delegate configures the initial view controller. When a segue is triggered, the source view controller configures the destination view controller.

There are a few conventions used to implement destination view controllers:

- A destination view controller exposes properties and methods used to configure it.

- A destination view controller communicates as little as possible with view controllers that it did not create personally. When it does so, these communication paths should use delegation. The destination view controller's delegate is configured as one of its properties.

Carefully following these conventions helps organize your configuration code and carefully limits the direction of dependencies between view controller classes in your app. By isolating dependencies in your app, you increase the opportunity for code reuse. You also design view controllers that are easier to test in isolation from the rest of your app.

## Configuring the Initial View Controller at Launch

If you define a main storyboard in your project, iOS automatically does a lot of work for you to set up your app. When your app calls the `UIApplicationMain` function, iOS performs the following actions:

1. It instantiates the app delegate based on the class name you passed into the `UIApplicationMain` function.
2. It creates a new window attached to the main screen.
3. If your app delegate implements a `window` property, iOS sets this property to the new window.
4. It loads the main storyboard referenced in the app's information property list file.
5. It instantiates the main storyboard's initial view controller.
6. It sets the window's `rootViewController` property to the new view controller.
7. It calls the app delegate's `application:didFinishLaunchingWithOptions:` method. Your app delegate is expected to configure the initial view controller (and its children, if it is a container view controller).
8. It calls the window's `makeKeyAndVisible` method to display the window.

Listing 11-1 shows an implementation of the `application:didFinishLaunchingWithOptions:` method from the *Your Second iOS App: Storyboards* tutorial. In this example, the storyboard's initial view controller is a navigation controller with a custom content controller that displays the master view. The code first retrieves references to the view controller it is interested in. Then, it performs any configuration that could not be performed in Interface Builder. In this example, a custom data controller object is provided to the master view controller by a custom data controller object.

**Listing 11-1** The app delegate configures the controller

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

{
```

```
    UINavigationController *navigationController = (UINavigationController*)
self.window.rootViewController;
    BirdsMasterViewController * firstViewController = [[navigationController
viewControllers] objectAtIndex:0];

    BirdSightingDataController *dataController = [[BirdSightingDataController
alloc] init];
    firstViewController.dataController = dataController;

    return YES;
}
```

If your project does not identify the main storyboard, the `UIApplicationMain` function creates the app delegate and calls it but does not perform any of the other steps described earlier. You would need to write code to perform those steps yourself. Listing 11-2 shows the code you might implement if you needed to perform these steps programmatically.

**Listing 11-2** Creating the window when a main storyboard is not being used

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    UIStoryboard *storyboard = [UIStoryboard storyboardWithName:@"MyStoryboard"
bundle:nil];
    MainViewController *mainViewController = [storyboard
instantiateInitialViewController];
    self.window.rootViewController = mainViewController;

    // Code to configure the view controller goes here.

    [self.window makeKeyAndVisible];
    return YES;
}
```

## Configuring the Destination Controller When a Segue is Triggered

iOS performs the following tasks when a segue is triggered:

1. It instantiates the destination view controller.
2. It instantiates a new segue object that holds all the information for the segue being triggered.

> **Note:** A popover segue also provides a property that identifies the popover controller used to control the destination view controller.

3. It calls the source view controller's `prepareForSegue:sender:` method, passing in the new segue object and the object that triggered the segue.
4. It calls the segue's `perform` method to bring the destination controller onto the screen. The actual behavior depends on the kind of segue being performed. For example, a modal segue tells the source view controller to present the destination view controller.
5. It releases the segue object and the segue is complete.

The source view controller's `prepareForSegue:sender:` method performs any necessary configuration of the destination view controller's properties, including a delegate if the destination view controller implements one.

Listing 11-3 shows an implementation of the `prepareForSegue:sender:` method from the *Your Second iOS App: Storyboards* tutorial.

**Listing 11-3** Configuring the destination controller in a segue

```
- (void) prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    if ([[segue identifier] isEqualToString:@"ShowSightingsDetails"])
    {
        DetailViewController *detailViewController = [segue
destinationViewController];
        detailViewController.sighting = [self.dataController
objectInListAtIndex:[self.tableView indexPathForSelectedRow].row];
    }

    if ([[segue identifier] isEqualToString:@"ShowAddSightingView"])
    {
        AddSightingViewController *addSightingViewController = [[[segue
destinationViewController] viewControllers] objectAtIndex:0];
        addSightingViewController.delegate = self;
```

```
    }
}
```

This implementation, from the master view controller for the app, actually handles two different segues configured in the storyboard. It distinguishes between the two segues using the segue's `identifier` property. In both cases, it follows the coding convention established earlier, by first retrieving the view controller and then configuring it.

When the segue is to the detail view controller, the segue occurred because the user selected a row in the table view. In this case, the code transfers enough data to the destination view controller so that the destination view controller can display the sighting. The code uses the user's selection to retrieve a sighting object from the master view controller's data controller. It then assigns this sighting to the destination controller.

In the other case, the new view controller allows the user to add a new bird sighting. No data needs to be sent to this view controller. However, the master view controller needs to receive data when the user finishes entering the data. To receive that information, the source view controller implements the delegate protocol defined by the Add view controller (not shown here) and makes itself the destination view controller's delegate.

## Using Delegation to Communicate with Other Controllers

In a delegate-based model, the view controller defines a protocol for its delegate to implement. The protocol defines methods that are called by the view controller in response to specific actions, such as taps in a Done button. The delegate is then responsible for implementing these methods. For example, when a presented view controller finishes its task, it sends a message to the presenting view controller and that controller dismisses it.

Using delegation to manage interactions with other app objects has key advantages over other techniques:

- The delegate object has the opportunity to validate or incorporate changes from the view controller.
- The use of a delegate promotes better encapsulation because the view controller does not have to know anything about the class of the delegate. This enables you to reuse that view controller in other parts of your app.

To illustrate the implementation of a delegate protocol, consider the recipe view controller example that was used in "Presenting a View Controller and Choosing a Transition Style" (page 93). In that example, a recipes app presented a view controller in response to the user wanting to add a new recipe. Prior to presenting the view controller, the current view controller made itself the delegate of the `RecipeAddViewController` object. Listing 11-4 shows the definition of the delegate protocol for `RecipeAddViewController` objects.

**Listing 11-4** Delegate protocol for dismissing a presented 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 11-5 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. 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.) Then the delegate method dismisses the presented view controller.

**Listing 11-5** Dismissing a presented 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 dismissViewControllerAnimated:YES completion: nil];
}
```

# Guidelines for Managing View Controller Data

Carefully managing how data and control flows between your view controllers is critical to understanding how your app operates and avoiding subtle errors. Consider the following guidelines when designing your view controllers:

- A destination view controller's references to app data should come from the source view controller unless the destination view controller represents a self-contained (and therefore self-configuring) view controller.
- Perform as much configuration as possible using Interface Builder, rather than configuring your controller programmatically in your code.
- Always use a delegate to communicate information back to other controllers. Your content view controller should never need to know the class of the source view controller or any controllers it doesn't create.
- Avoid unnecessary connections to objects external to your view controller. Each connection represents a dependency that makes it harder to change your app design.

  For example, the children of a navigation controller should be aware of the parent navigation controller and of the siblings immediately above and below them on the stack. They rarely need to communicate with other siblings.

# Enabling Edit Mode in a View Controller

You can use the same view controller both to display and to edit content. When the editing mode is toggled, your custom view controller performs the necessary work to transition its view from a display mode to an editing mode (or vice versa).

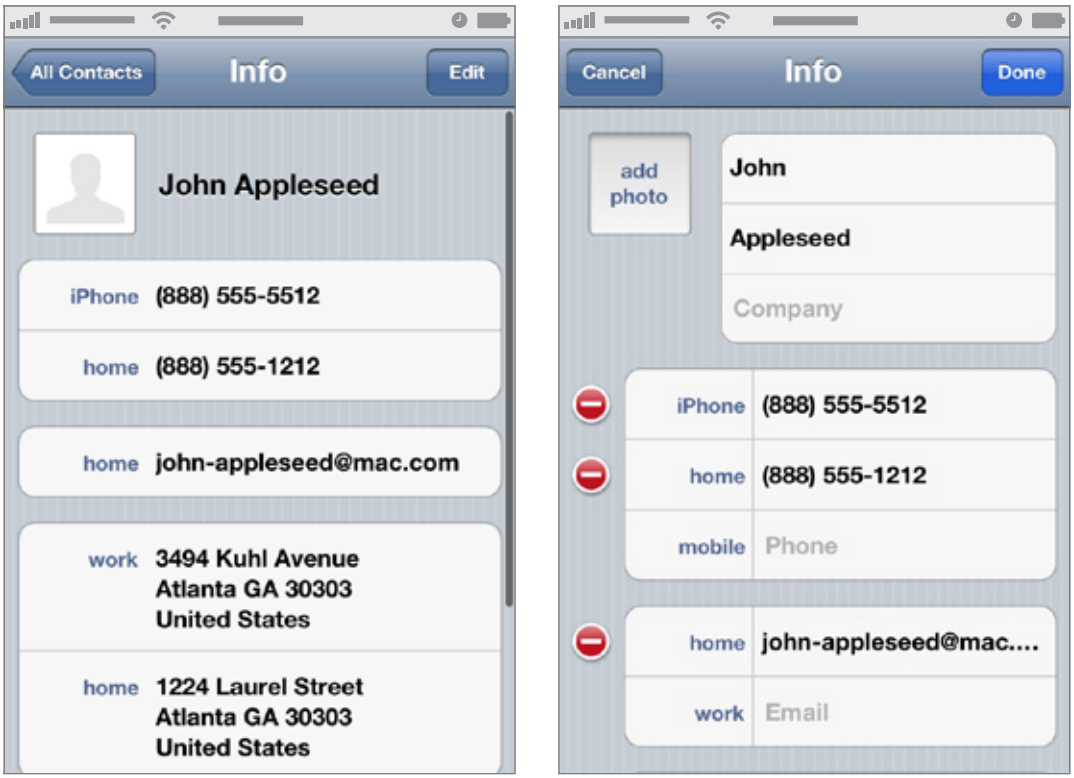
## Toggling Between Display and Edit Mode

To allow a custom view controller class to be used to both display and edit content, override the `setEditing:animated:` method. 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 in order 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. The 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 or use a navigation controller to present the new views.

Figure 12-1 shows a view from the Contacts app 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 12-1** Display and edit modes of a view




Display mode

Edit mode

The implementation of your own `setEditing:animated:` method is relatively straightforward—you 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 the views to noneditable.
    }
}
```

## Presenting Editing Options to the User

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 *View Controller Catalog for iOS*.

# Creating Custom Segues

Interface Builder provides segues for all of the standard ways to transition from one view controller to another—from presenting a view controller to displaying a controller in a popover. However, if one of those segues doesn't do what you want, you can create a custom segue.

## The Life Cycle of a Segue

To understand how custom segues work, you need to understand the life cycle of a segue object. Segue objects are instances of `UIStoryboardSegue` or one of its subclasses. Your app never creates segue objects directly; they are always created on your behalf by iOS when a segue is triggered. Here's what happens:

1. The destination controller is created and initialized.
2. The segue object is created and its `initWithIdentifier:source:destination:` method is called. The identifier is the unique string you provided for the segue in Interface Builder, and the two other parameters represent the two controller objects in the transition.
3. The source view controller's `prepareForSegue:sender:` method is called. See "Configuring the Destination Controller When a Segue is Triggered" (page 101).
4. The segue object's `perform` method is called. This method performs a transition to bring the destination view controller on-screen.
5. The reference to the segue object is released, causing it to be deallocated.

## Implementing a Custom Segue

To implement a custom segue, you subclass `UIStoryboardSegue` and implement the two methods described earlier:

- If you override the `initWithIdentifier:source:destination:` method, call the superclass's implementation, then initialize your subclass.
- Your `perform` method must make whatever view controller calls are necessary to perform the transition you want. Typically, you use any of the standard ways to display a new view controller, but you can embellish this design with animations and other effects.

> **Note:** If your implementation adds properties to configure the segue, you cannot configure these attributes in Interface Builder. Instead, configure the custom segue's additional properties in the `prepareForSegue:sender:` method of the source view controller that triggered the segue.

"Creating Custom Segues" shows a very simple custom segue. This example simply presents the destination view controller without any sort of animation, but you can extend this idea with your own animations as necessary.

**Listing 13-1** A custom segue

```
- (void)perform
{
// Add your own animation code here.


    [[self sourceViewController] presentModalViewController:[self
destinationViewController] animated:NO];
}
```

# Creating Custom Container View Controllers

Container view controllers are a critical part of iOS app design. They allow you to decompose your app into smaller and simpler parts, each controlled by a view controller dedicated to that task. Containers allow these view controllers to work together to present a seamless interface.

iOS provides many standard containers to help you organize your apps. However, sometimes you need to create a custom workflow that doesn't match that provided by any of the system containers. Perhaps in your vision, your app needs a specific organization of child view controllers with specialized navigation gestures or animation transitions between them. To do that, you implement a custom container.

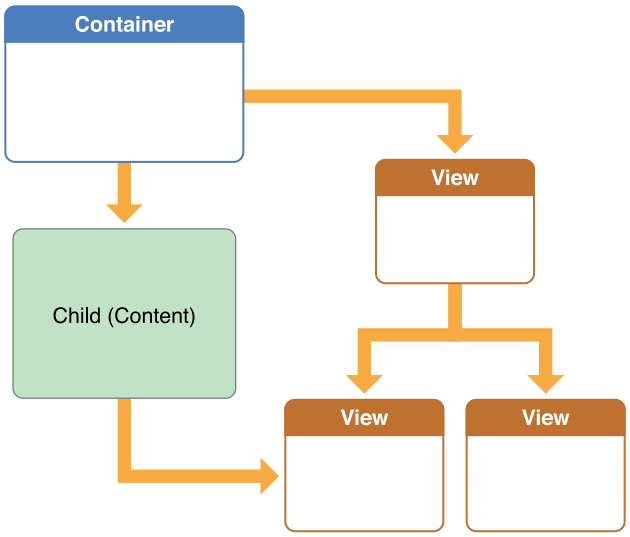
## Designing Your Container View Controller

In most ways, a container view controller is just like a content view controller. It manages views and content, coordinates with other objects in your app, and responds to events in the responder chain. Before designing a container controller, you should already be familiar with designing content view controllers. The design questions in "Creating Custom Content View Controllers" (page 43) also apply when creating containers.

When you design a container, you create explicit parent-child relationships between your container, the parent, and other view controllers, its children. More specifically, Figure 14-1 shows that there are explicit connections between the views as well. Your container adds the content views of other view controllers in its own view

hierarchy. Whenever a child's view is displayed in the container's view hierarchy, your container also establishes a connection to the child view controller and ensures that all appropriate view controller events are sent to the child.

**Figure 14-1** A container view controller's view hierarchy contains another controller's views




Your container should make the rules and its children should follow them; it is up to the parent to decide when a child's content is visible in its own view hierarchy. The container decides where in the hierarchy that view is placed and how it is sized and positioned there. This design principle is no different from that of a content view controller. The view controller is responsible for managing its own view hierarchy and other classes should never manipulate its contents. Where necessary, your container class can expose public methods and properties to allow its behavior to be controlled. For example, if another object needs to be able to tell your container to display a new view, then your container class should expose a public method to allow this transition to occur. The actual implementation that changes the view hierarchy should be in the container class. This guiding principle cleanly separates responsibilities between the container and its children by always making each view controller responsible for its own view hierarchy.

Here are some specific questions you should be able to answer about your container class:

- What is the role of the container and what role do its children play?
- Is there a relationship between siblings?
- How are child view controllers added to or removed from the container? Your container class must provide public properties and methods to allow children to be displayed by it.
- How many children are displayed by the container?

- Are the contents of the container static or dynamic? In a static design, the children are more or less fixed, whereas in a dynamic design, transitions between siblings may occur. You define what triggers a transition to a new sibling. It might be programmatic or it might happen when a user interacts with the container.
- Does the container own any of its own views? For example, your container's user interface may include information about the child view controller or controls to allow navigation.
- Does the container require its children to provide methods or properties other than those found on the `UIViewController` class? There are many reasons why a container might do this. It might need specific information from the child to used to configure other aspects of container display, or it might allow the child to modify the container's behavior. It even might call the child view controller when container-specific events occur.
- Does your container allow its behavior to be configured?
- Are all its children treated identically or does it have multiple types of children, each with specialized behaviors? For example, you might create a container that displays two children, coordinating actions between the two children. Each child implements a distinct set of methods to allow its behavior to be configured.

In summary, a container controller often has more relationships with other objects (especially other view controllers) than a content controller. So, you need to put additional effort into understanding how the container works. Ideally, as with a content controller, you want to hide many of those behaviors behind an excellent public class API.

# Examples of Common Container Designs

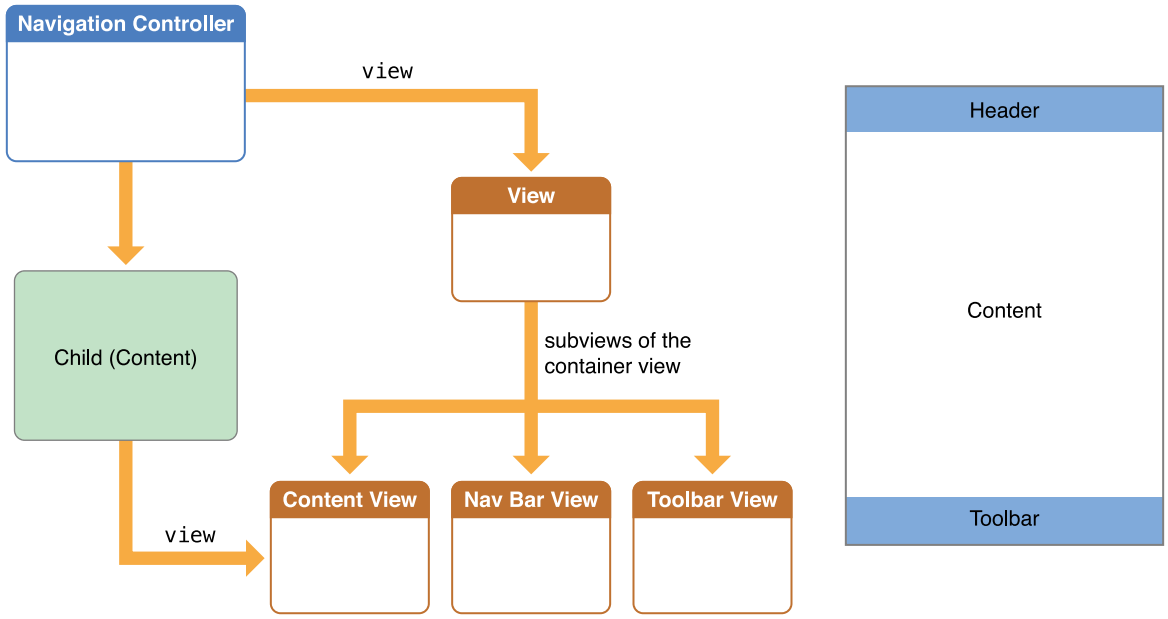
The easiest way to understand how to design a new container class is to examine the behavior and public API of the existing system container classes. Each defines its own navigation metaphor and a programming interface used to configure it. This section takes a look at a few of these classes from the viewpoint of container design. It does not provide a complete description of each class's programming interface, but just looks at some of the critical concepts. For detailed information about using these system containers, see *View Controller Catalog for iOS*.

## A Navigation Controller Manages a Stack of Child View Controllers

A navigation controller allows a sequence of distinct user interface screens to be displayed to the user. The metaphor used by a navigation controller is a stack of child view controllers. The topmost view controller's view is placed in the navigation controller's view hierarchy. To display a new view controller, you push it onto the stack. When you are done, you remove the view controller from the stack.

Figure 14-2 shows that only a single child's view is visible and that the child's view is part of a more complex hierarchy of views provided by the navigation controller.

**Figure 14-2** A navigation controller's view and view controller hierarchy




When a view controller is pushed onto or popped from the stack, the transition can be animated, which means the views of two children are briefly displayed together. In addition to the child views, a navigation controller also includes its own content views to display a navigation bar. The contents of the navigation bar are updated based on the child being displayed.

Here are some of the important methods and properties that the `UINavigationController` class uses to define its behavior:

- The `topViewController` property states which controller is at the top of the stack.
- The `viewControllers` property lists all the children in the stack.
- The `pushViewController:animated:` method pushes a new view controller on the stack. This method does all the work necessary to update the view hierarchy to display the new child's view.
- The `popViewControllerAnimated:` method removes the top view controller from the stack.
- The `delegate` property allows a client of the container to be notified when state transitions occur.

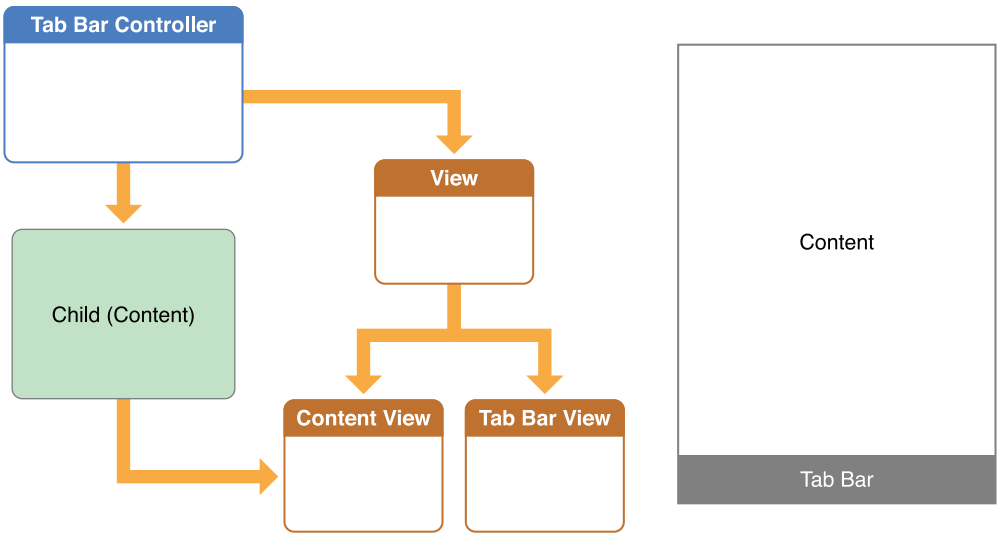
The navigation controller uses properties on the child view controller to adjust the content it displays. These properties are defined by `UIViewController` base class so that some default behavior is available; this allows any view controller to be made a child of a navigation controller. Here are some of the properties the navigation controller looks for:

- The `navigationItem` property provides the contents of the navigation toolbar.
- The `toolbarItems` property provides the contents of the bottom bar.
- The `editButtonItem` property provides access to a view in the navigation item so that the navigation controller can toggle the child view's edit mode.

## A Tab Bar Controller Uses a Collection of Child Controllers

A tab view controller allows a set of distinct user interface screens to be displayed to the user. However, instead of a stack of view controllers, a tab view controller uses a simple array. Figure 14-3 shows that again, only one child view controller's view is displayed at a time. However, these views do not need to be accessed sequentially, and the transition to the new child is usually not animated.

**Figure 14-3** A tab bar controller's view and view controller hierarchy




Here are some of the important methods and properties that `UITabBarController` class uses to allow apps to control what a tab bar controller displays:

- The `viewControllers` property holds the list of child view controllers that act as the tabs of content.
- The `selectedViewController` property allows you to read or change which child is visible.
- The `delegate` property allows a client of the container to be notified when state transitions occur.

A tab bar controller uses the child's `tabBarItem` property to determine how it is displayed in the appropriate tab.

### A Page Controller Uses a Data Source to Provide New Children

A page controller uses pages of content as its metaphor, like the pages of a book. Each page displayed by the container is provided by a child view controller.

Books can have many pages—far more than the number of screens of content in a navigation controller—so keeping all the pages in memory at once may not be possible. Instead, the page view controller keeps child controllers for the visible pages and fetches other pages on demand. When the user wants to see a new page, the container calls the object associated with its `dataSource` property to get the new controller. Thus, a page view controller using a data source uses a pull model rather than having your app directly push new pages onto itself.

A page view controller can also be customized for different kinds of book layouts. The number of pages and the size of the pages can differ. Here are two key properties that affect the page view controller's behavior:

- The `spineLocation` property determines how the pages are organized. Some layouts only display one page at a time. Other layouts display multiple pages.
- The `transitionStyle` property determines how transitions between pages are animated.

## Implementing a Custom Container View Controller

Once you've designed your class's behavior and determined many aspects of its public API, you are ready to start implementing the container. The goal of implementing a container is to be able to add another view controller's view (and associated view hierarchy) as a subtree in your container's view hierarchy. The child remains responsible for its own view hierarchy, save for where the container decides to place it onscreen. When you add the child's view, you need to ensure that events continue to be distributed to both view controllers. You do this by explicitly associating the new view controller as a child of the container.

The `UIViewController` class provides methods that a container view controller uses to manage the relationship between itself and its children. The complete list of methods and properties is in the reference; see "Managing Child View Controllers in a Custom Container" in *UIViewController Class Reference*

> **Important:** These `UIViewController` methods are only intended to be used to implement container view controllers; do not call them in a content view controller.

### Adding and Removing a Child

Listing 14-1 shows a typical implementation that adds a view controller as a child of another view controller. Each numbered step in the listing is described in more detail following the listing.

**Listing 14-1** Adding another view controller's view to the container's view hierarchy

```
- (void) displayContentController: (UIViewController*) content;
{
   [self addChildViewController:content];                 // 1
   content.view.frame = [self frameForContentController]; // 2
   [self.view addSubview:self.currentClientView];
   [content didMoveToParentViewController:self];          // 3
}
```

Here's what the code does:

1. It calls the container's `addChildViewController:` method to add the child. Calling the `addChildViewController:` method also calls the child's `willMoveToParentViewController:` method automatically.
2. It accesses the child's `view` property to retrieve the view and adds it to its own view hierarchy. The container sets the child's size and position before adding the view; containers always choose where the child's content appears. Although this example does this by explicitly setting the frame, you could also use layout constraints to determine the view's position.
3. It explicitly calls the child's `didMoveToParentViewController:` method to signal that the operation is complete.

Eventually, you want to be able to remove the child's view from the view hierarchy. In this case, shown in Listing 14-2, you perform the steps in reverse.

**Listing 14-2** Removing another view controller's view to the container's view hierarchy

```
- (void) hideContentController: (UIViewController*) content
{
   [content willMoveToParentViewController:nil];  // 1
   [content.view removeFromSuperview];            // 2
   [content removeFromParentViewController];      // 3
}
```

Here's what this code does:

1. Calls the child's `willMoveToParentViewController:` method with a parameter of `nil` to tell the child that it is being removed.
2. Cleans up the view hierarchy.

3. Calls the child's `removeFromParentViewController` method to remove it from the container. Calling the `removeFromParentViewController` method automatically calls the child's `didMoveToParentViewController:` method.

For a container with essentially static content, adding and removing view controllers is as simple as that. Whenever you want to add a new view, add the new view controller as a child first. After the view is removed, remove the child from the container. However, sometimes you want to animate a new child onto the screen while simultaneously removing another child. Listing 14-3 shows an example of how to do this.

**Listing 14-3** Transitioning between two view controllers

```
- (void) cycleFromViewController: (UIViewController*) oldC
            toViewController: (UIViewController*) newC
{
    [oldC willMoveToParentViewController:nil];                        // 1
    [self addChildViewController:newC];

    newC.view.frame = [self newViewStartFrame];                       // 2
    CGRect endFrame = [self oldViewEndFrame];

    [self transitionFromViewController: oldC toViewController: newC   // 3
          duration: 0.25 options:0
          animations:^{
             newC.view.frame = oldC.view.frame;                       // 4
             oldC.view.frame = endFrame;
           }
           completion:^(BOOL finished) {
             [oldC removeFromParentViewController];                   // 5
             [newC didMoveToParentViewController:self];
            }];
}
```

Here's what this code does:

1. Starts both view controller transitions.
2. Calculates two new frame positions used to perform the transition animation.

3. Calls the `transitionFromViewController:toViewController:duration:options:animations:completion:` method to perform the swap. This method automatically adds the new view, performs the animation, and then removes the old view.
4. The animation step to perform to get the views swapped.
5. When the transition completes, the view hierarchy is in its final state, so it finishes the operation by sending the final two notifications.

## Customizing Appearance and Rotation Callback Behavior

Once you add a child to a container, the container automatically forwards rotation and appearance callbacks to the child view controllers as soon as an event occurs that requires the message to be forwarded. This is normally the behavior you want, because it ensures that all events are properly sent. However, sometimes the default behavior may send those events in an order that doesn't make sense for your container. For example, if multiple children are simultaneously changing their view state, you may want to consolidate the changes so that the appearance callbacks all happen at the same time in a more logical order. To do this, you modify your container class to take over responsibility for appearance or rotation callbacks.

To take over control of appearance callbacks, you override the `shouldAutomaticallyForwardAppearanceMethods` method to return `NO`. Listing 14-4 shows the necessary code.

**Listing 14-4** Disabling automatic appearance forwarding

```
- (BOOL) shouldAutomaticallyForwardAppearanceMethods
{
    return NO;
}
```

To actually inform the child view controller that an appearance transition is occurring, you call the child's `beginAppearanceTransition:animated:` and `endAppearanceTransition` methods.

If you take over sending these messages, you are also responsible for forwarding them to children when your container view controller appears and disappears. For example, if your container has a single child referenced by a `child` property, your container would forward these messages to the child, as shown in Listing 14-5.

**Listing 14-5** Forwarding appearance messages when the container appears or disappears

```
-(void) viewWillAppear:(BOOL)animated
{
```

```
    [self.child beginAppearanceTransition: YES animated: animated];
}

-(void) viewDidAppear:(BOOL)animated
{
    [self.child endAppearanceTransition];
}

-(void) viewWillDisappear:(BOOL)animated
{
    [self.child beginAppearanceTransition: NO animated: animated];
}

-(void) viewDidDisappear:(BOOL)animated
{
    [self.child endAppearanceTransition];
}
```

Forwarding rotation events works almost identically and can be done independently of forwarding appearance messages. First, you override the `shouldAutomaticallyForwardRotationMethods` method to return `NO`. Then, at times appropriate to your container, you call the following methods:

- `willRotateToInterfaceOrientation:duration:`
- `willAnimateRotationToInterfaceOrientation:duration:`
- `didRotateFromInterfaceOrientation:`

## Practical Suggestions for Building a Container View Controller

Designing, developing, and testing a new container view controller takes time. Although the individual behaviors are straightforward, the controller as a whole can be quite complex. Consider some of the following guidance when implementing your own container classes:

- Design the view controller first as a content view controller, using regular views owned by the container. This allows you to focus on getting layout and animation transitions correct without simultaneously needing to manage parent-child relationships.

- Never access any view other than the top-level view of the child view controller. Similarly, children should have only a minimal knowledge of what the parent is doing with the view; do not expose unnecessary details to the child.
- If the container needs the child to declare methods or properties, it should define a protocol to enforce this:

```
@protocol MyContentContainerProtocol <NSObject>

...

@end

- (void) displayContentController:
(UIViewController<MyContentContainerProtocol>*) content;
```

# Document Revision History

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

| Date | Notes |
| --- | --- |
| 2012-12-13 | Corrected a pair of figures related to display notifications. |
| 2012-09-19 | Added design guidelines for custom container view controllers and accessibility in the view controller. Updated the discussions of view rotation, view layout, and resource management for iOS 6. |
| 2012-02-16 | Edited for clarity. Added new glossary entries. Added a section about determining why a view controller's view appeared or disappeared. |
| 2012-01-09 | This revised treatment has been rewritten around using storyboards and ARC to build new iOS apps. |
| 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. |
| 2008-10-15 | Updated obsolete references to the iOS Programming Guide. |

| Date | Notes |
| --- | --- |
| 2008-09-09 | Corrected typos. |
| 2008-06-23 | New document that explains how to use view controllers to implement radio, navigation, and modal interfaces. |

# 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.

**content view controller** A view controller that displays some content on the screen.

**custom segue** A segue whose transition effect is defined by a custom subclass.

**modal segue** A segue whose transition effect presents the new view controller using an existing view controller.

**navigation controller** A container view controller used to present hierarchical content.

**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.

**page view controller** A container view controller used to display pages of content with an artistic style similar to that of a physical book.

**popover controller** A controller class used to present another view controller's view in a popover control.

**popover segue** A segue whose transition effect displays the new view controller's content in a popover control.

**push segue** A segue whose transition effect pushes the new view controller onto a navigation stack of a navigation controller.

**root view controller** The topmost view controller in a view controller hierarchy.

**scene** A visual representation in Interface Builder of a view controller and its associated objects, including the views it loads when displayed.

**segue** A transition between two scenes, configured in Interface Builder.

**split view controller** A container view controller used in iPad apps to present master-detail interfaces.

**tab bar controller** A container view controller used to present a set of distinct interface screens, each represented by a tab and delivered by a separate content view controller.

**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.

**view controller hierarchy** A set of container and content view controllers arranged in a tree. Non-leaf nodes always represent container view controllers.

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