SAMPLE CHAPTER

iOS 7 IN ACTION

Brendan G. Lim
Martin Conte Mac Donell

MANNING

iOS 7 in Action

by Brendan G. Lim
Martin Conte Mac Donell

Chapter 2

brief contents

2 Views and view controller basics

This chapter covers

- Enhancing your Hello Time app
- Views and the view coordinate system
- User interface controls and responding to events
- View controller lifecycle and creating views programmatically
- Supporting multiple orientations

You've all used apps on your mobile devices. The way you interact with them is much different than the way you use apps on your laptops or desktop computers. Why is that? The amount of space available to present information on mobile screens is much smaller than on desktop applications. Only a limited amount of data can be displayed on the screen at once to ensure a good visual experience.

You also interact with your apps differently. Instead of using peripherals such as a keyboard and mouse as input devices, you use your fingers. The appearance and design should feel natural to users. You should take all of this into account when creating and designing the views that make up your apps.

In this chapter we'll go over the different parts of what you see in an app—its windows, its views, and the view controllers that manage them. You'll learn about

Figure 2.1 The updated Hello Time application after adding more functionality to it

the different controls you can use within each view of your app, such as buttons, labels, and text fields. You'll then learn how to visually arrange and organize the views within your application, better known as storyboarding. First, we'll revisit the Hello Time app you created in chapter 1 and add more functionality to it.

2.1 Enhancing Hello Time

When you created Hello Time in the previous chapter, the goal was to have a simple application that would act as a clock. In this chapter, you'll be giving it a few enhancements, as shown in figure 2.1.

First, you'll be adding a button to the view that will enable a night mode. This mode should make Hello Time easier on the eyes when you're using it at night. You'll also need a way to switch back to day mode. Next, you'll ensure that Hello Time appears properly when the device it's running on is rotated into landscape mode.

2.1.1 Switching between night and day modes

To begin, launch Xcode and open the Hello Time project. Because you're adding a night mode, you need a way for someone to turn it on. You'll be adding a button below the label that you're using to display the time. In the project navigator (View > Navigators > Show Project Navigator, or press Command-1), choose Main.storyboard to bring up the interface editor. You should see everything just as you left it, as shown in figure 2.2.

Locate the Object Library on the bottom right of your workspace window. Search for "button" to locate Button and drag it underneath the label you've just added, as shown in figure 2.3.

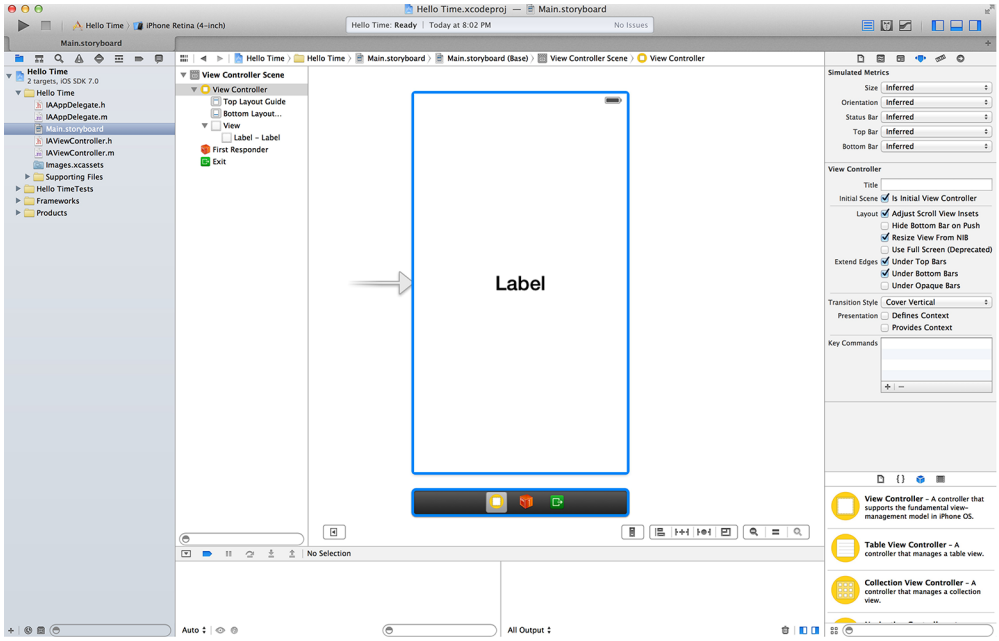

Figure 2.2 The interface for Hello Time just as you left it in the previous chapter

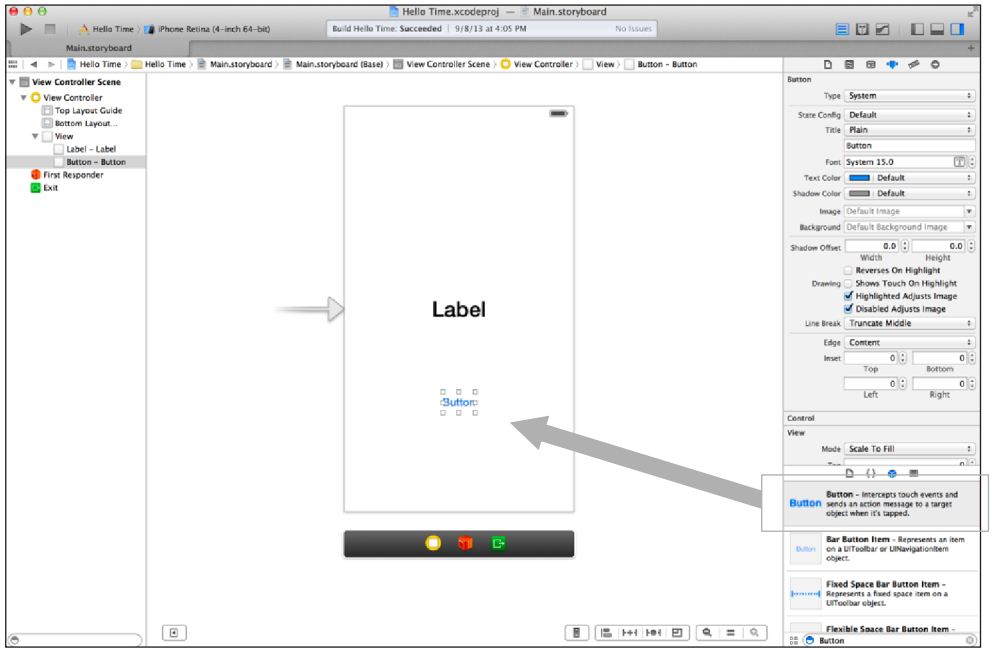

Figure 2.3 Dragging a button from the Object Library onto the bottom of your view

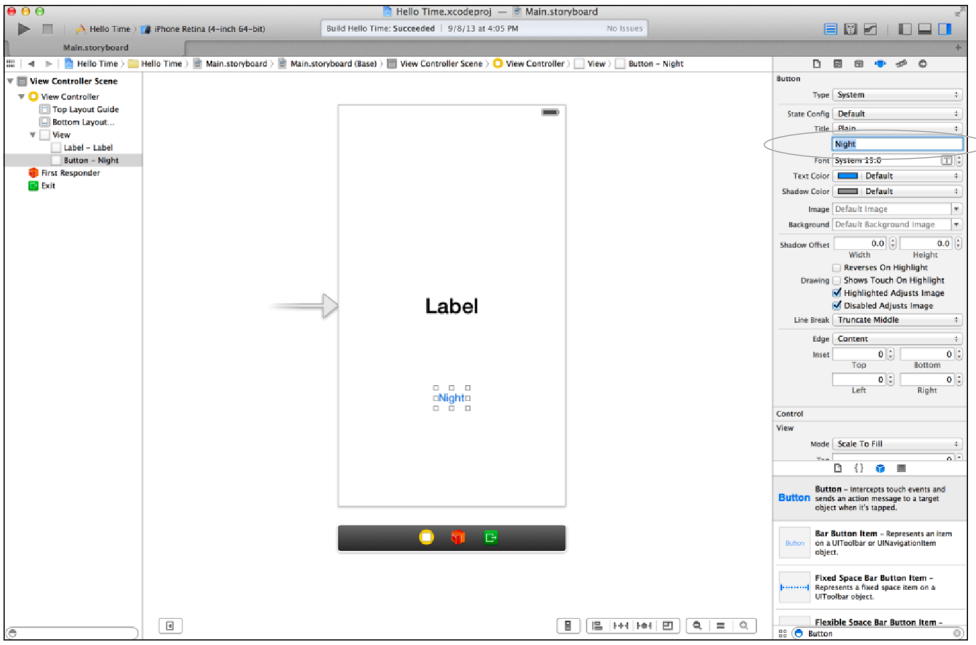

Figure 2.4 Change the title of the button to "Night" by editing the Title property in the inspector.

Next, you can change the text of this button from "Button" to "Night." On the right side of your workspace window, with your button selected, look for the Title property and change its entry to "Night." You can see this in figure 2.4.

You've now finished adding the button to your interface. The next step is to connect the button so that the code you're going to write can interact with it. First, open the assistant editor by choosing View > Assistant Editor > Show Assistant Editor (Option-Command-Return) in the application menu. This should bring up IAViewController.h on the right side of your workspace. Hold down the Control key on your keyboard and drag an outlet into `IAViewController`, as shown in figure 2.5.

When you let go, you'll be prompted to set a name for this new connection. Call it `modeButton`. Once you make the connection, the following code will be inserted:

```
@property(weak, nonatomic) IBOutlet UIButton *modeButton;
```

This will allow you to make changes to the button such as modifying the label you're using to display the current time. What about when someone taps the button? How do you know when that happens? Let's create something to handle that event.

In the same way that you dragged a connection to create an outlet, hold down Control and drag the connection into the assistant editor. In the pop-up that appears, change the connection to Action, change the event to Touch Up Inside, and set the name to `toggleMode`, as shown in figure 2.6.

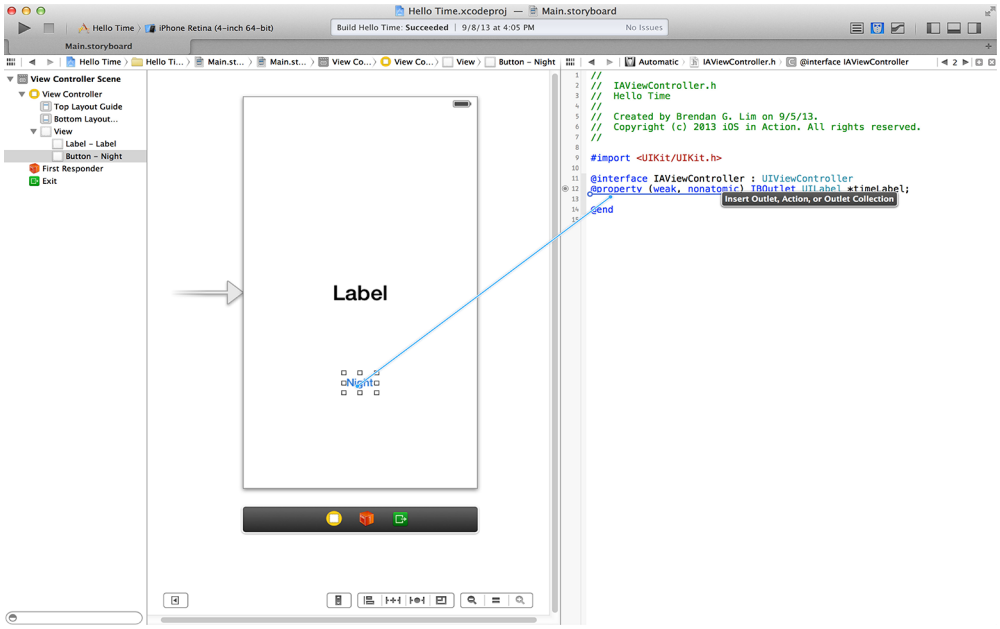

Figure 2.5 Drag a connection from the button in your view to `IAViewController`'s interface.

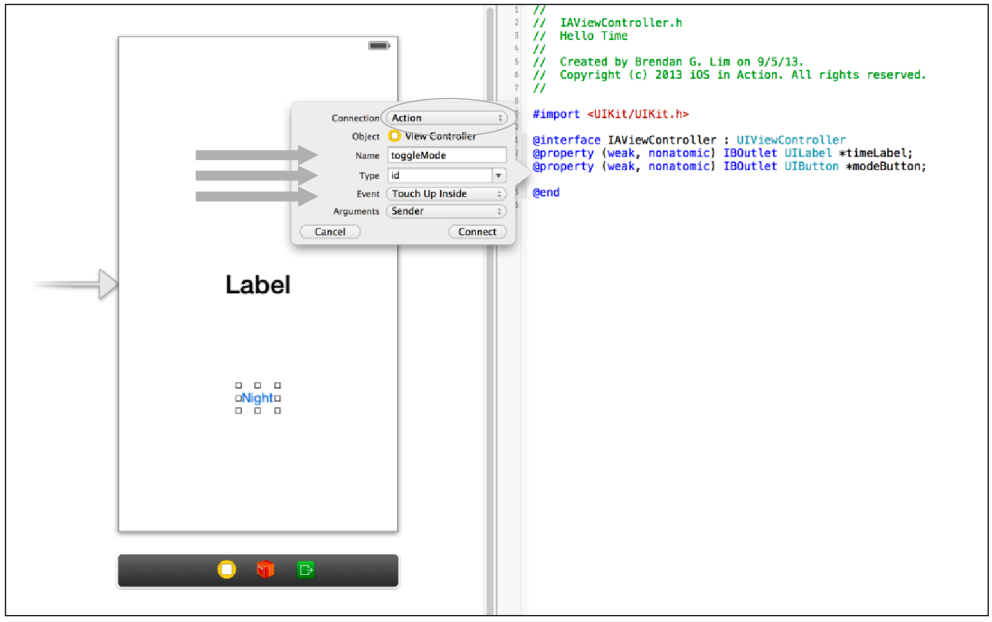

Figure 2.6 Create an action for the Touch Up Inside event called `toggleMode`.

Once the connection is made, Xcode will generate a `toggleMode:` method that will serve as an action that will be triggered when your button is touched. Let's add the code for this action that will enable or disable night mode.

Go back to the standard editor by selecting the application menu and choosing View > Standard Editor > Show Standard Editor (Command-Return). Next, select the project navigator and choose IAViewController.m. Inside the editor you'll see a blank `toggleMode:` method that Xcode generated. Replace it with the code shown in the following listing.

Listing 2.1 Toggling between night and day modes

```
- (IBAction)toggleMode:(id)sender {
    if ([self.modeButton.titleLabel.text isEqualToString:@"Night"]) {
        self.view.backgroundColor = [UIColor blackColor];
        self.timeLabel.textColor = [UIColor whiteColor];
        [self.modeButton setTitle:@"Day" forState:UIControlStateNormal];
    } else {
        self.view.backgroundColor = [UIColor whiteColor];
        self.timeLabel.textColor = [UIColor blackColor];
        [self.modeButton setTitle:@"Night" forState:UIControlStateNormal];
    }
}
```

If you take a look at figure 2.7, you'll see this code added to `IAViewController`.

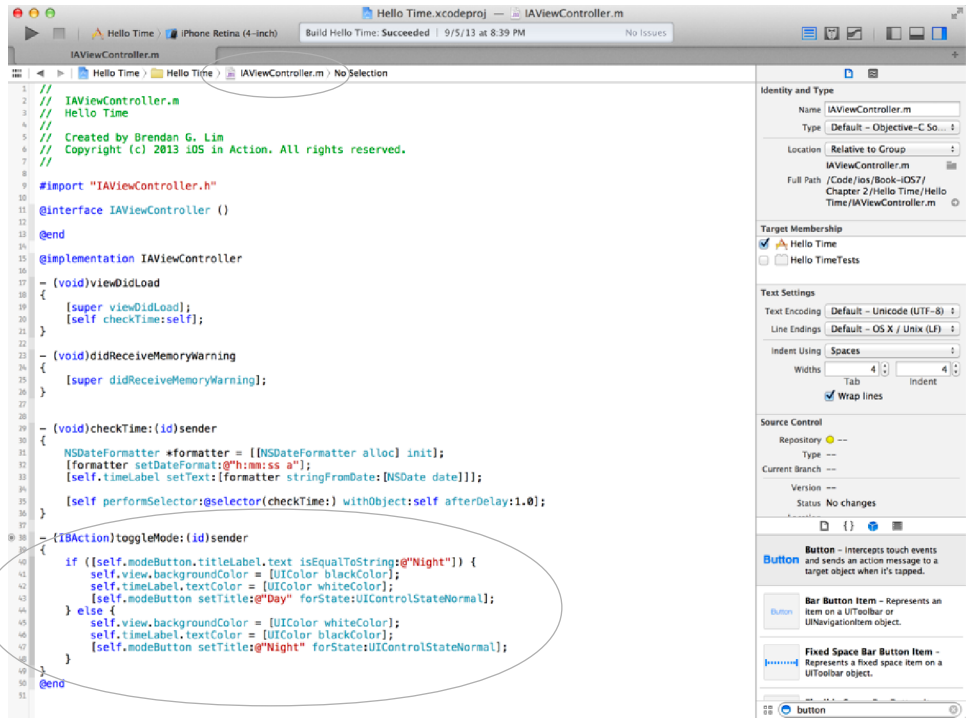

Figure 2.7 Your code added to the `toggleMode:` action within `IAViewController`

Figure 2.8 The Hello Time app currently doesn't support landscape orientation. You're going to fix this.

This will take care of switching between night and day modes. The next thing you're going to add to Hello Time is support for landscape mode.

2.1.2 Adding support for landscape mode

If you were to view your application in landscape mode right now, it would look pretty strange. The time label and the button don't reposition themselves properly when the orientation changes. You can see this in figure 2.8.

You'll alleviate this by adding support for landscape mode. You'll make the label appear in the center of the view and make the button disappear. This will make toggling between night and day modes possible only within portrait mode.

Jump back into Xcode and into IAViewController.m. You're going to add two new methods. One of them will declare the orientations that you'll support. The other will adjust your view so that it appears different in landscape versus portrait mode. Add the two methods in the following listing to the bottom of IAViewController.m.

Listing 2.2 Changing the view depending on orientation

```
- (void) willAnimateRotationToInterfaceOrientation:
(UIInterfaceOrientation)toInterfaceOrientation
 duration:(NSTimeInterval)duration
{
    CGRect timeFrame = self.timeLabel.frame;
    float viewHeight = self.view.frame.size.height;
    float viewWidth = self.view.frame.size.width;
    float fontSize = 30.0f;
    BOOL hideButton = YES;

    if (UIInterfaceOrientationIsLandscape(self.interfaceOrientation)) {
        fontSize = 40.0f;
        timeFrame.origin.y = (viewWidth / 2) - timeFrame.size.height;
        timeFrame.size.width = viewHeight;
    } else {
        hideButton = NO;
        timeFrame.origin.y = (viewHeight / 2) - timeFrame.size.height;
        timeFrame.size.width = viewWidth;
    }
```

```
    [self.modeButton setHidden:hideButton];
    [self.timeLabel setFont:[UIFont boldSystemFontOfSize:fontSize]];
    [self.timeLabel setFrame:timeFrame];
}

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

You can see these two methods added to `IAViewController` by taking a look at figure 2.9.

That's it! You've finished adding enhancements to your Hello Time app. You'll be referencing what you've done throughout the chapter. But first, we should take a closer look at what views are and how they are used within an app.

2.2 Introducing views

It's important to familiarize yourself with the basic components that make up the visual layer of your application. These include screens, windows, views, and controls. You'll learn what you're actually seeing in an iOS app and how apps work inside and out.

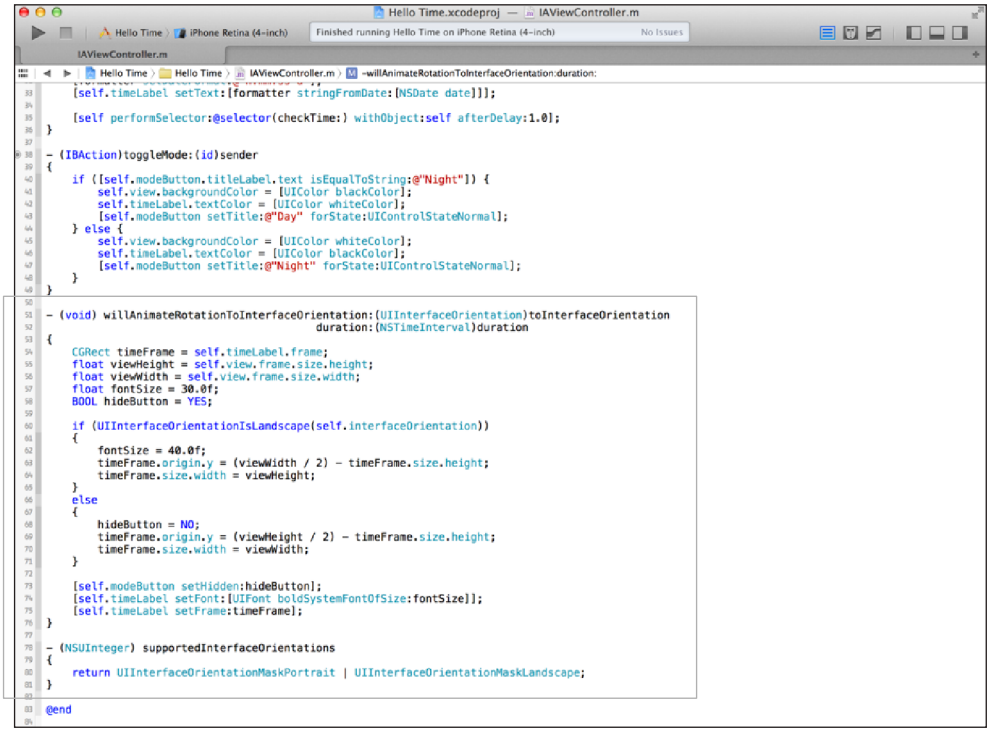

Figure 2.9 Adding two methods to `IAViewController` to support two different orientations

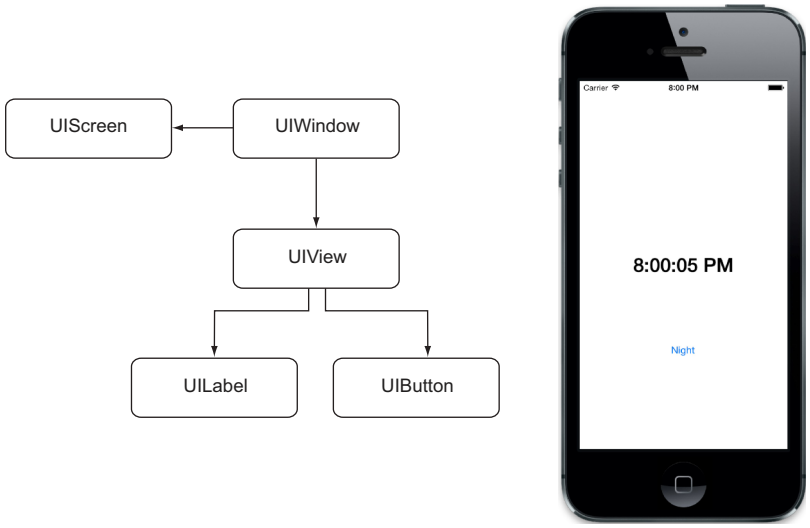

Figure 2.10 The Hello Time app, which has a label and button within a view, which is contained within a window inside a screen

2.2.1 Screens, windows, and views

Everything you see in an app is primarily made up of views. These views are displayed within a window, and that window is contained within a screen. Take a look at the anatomy of a view you'd see within your Hello Time application by focusing on figure 2.10.

Two items are shown on the screen of the iPhone in figure 2.10—a label and a button. Labels (`UILabel`) and buttons (`UIButton`) are known as controls—as are text fields (`UITextField`), images (`UIImageView`), and the like. These controls are all subclasses of `UIView`.

Notice how these two controls are contained within a white area? This white area is a view (`UIView`) that has a property that specifies its background color. We changed this background property when switching to night mode using the following code.

```
self.view.backgroundColor = [UIColor blackColor];
```

This view is contained within a single window (`UIWindow`), which is contained within a screen (`UIScreen`). You generally have only one screen or window when working with an application. You may have more screens and windows if you're using video-out or AirPlay to display something on a different screen or device. Here's an overview of these three types of classes:

- `UIScreen`—Represents the physical screen of the device
- `UIWindow`—Provides drawing support for displaying views on a `UIScreen`
- `UIView`—Represents a user interface element within a rectangular area

As you learned in the previous chapter, every class is a subclass of an `NSObject`. A `UIView` also inherits from `UIResponder`, which means it's capable of responding to actionable

events. A `UIButton`'s direct parent is `UIControl`, which is a subclass of `UIView`. `UIControl` subclasses are views that you directly interact with, like buttons, labels, or text fields. Views can also be nested or layered within a hierarchy. View controls, like a label or button that you're using in Hello Time, are nested as subviews of a parent view.

Views can also be animated, which is used heavily in iOS. Almost every time you transition from one view to another in an app, the current view slides out to the left while the new view slides in from the right. When a modal-style view appears, it comes in from the bottom of the screen and slides upward until it's fully in view. When you delete an item from the Mail app, the row that contains the email you're deleting slides out. When you delete a photo from the Photos app, the `UIImageView` that contains your photo gets sucked into the trashcan button. Animations can be applied as changes to properties of a particular view for a specific duration. You'll learn more about animations in a later chapter. Next, you'll learn about the coordinate system and how views are represented within a window.

2.2.2 Views and the coordinate system

When you lay out your views, you do so by providing X and Y coordinates as well as a width and height. These coordinates are placed within a view coordinate system with the point (0,0) at the top left of the screen. The X coordinate, Y coordinate, width, and height are contained within a `CGRect` and are referred to as a view's frame. You can provide a view with a frame either programmatically or by using Interface Builder.

Most initializer methods for a `UIView` ask for a frame to be provided. For example, let's say you want to create a `UIView` that is 200 x 200 and positioned at the X coordinate 10 and the Y coordinate 10. You would use the `initWithFrame:` method and supply a `CGRect` as your parameter by using the `CGRectMake()` function. This function takes four parameters: `x`, `y`, `width`, and `height`:

```
CGRect frame = CGRectMake(10,10,200,200);
UIView *foo = [[UIView alloc] initWithFrame:frame];
```

Let's give your view a red background to make it stand out. A white view on top of another white view would make it impossible to see.

```
[foo setBackgroundColor:[UIColor redColor]];
```

Your view will appear on the top-left corner of its parent view. You can see this in figure 2.11.

This is because the coordinate system for iOS assumes (0,0) to be at the top-left corner. The coordinate system is also *relative* to the parent of the view. For instance, if you were to add a view within the red `UIView` shown in figure 2.11 at (10,10), it would appear 10 points to the right and to the bottom of the red view's top-left corner. This is because (0,0) of the red view is at its absolute top-left corner. To visualize this, take a look at figure 2.12.

Here we've added a white view at (10,10) with a size of 150 x 150 as a subview of our red view. Instead of appearing at the exact same position as the original red view

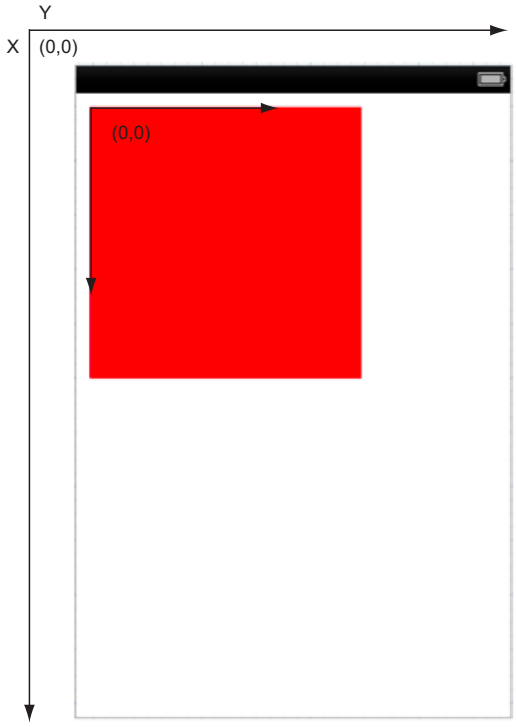

Figure 2.11 A view at coordinates (10,10) and measuring 200 x 200 will appear at the top left of its parent view.

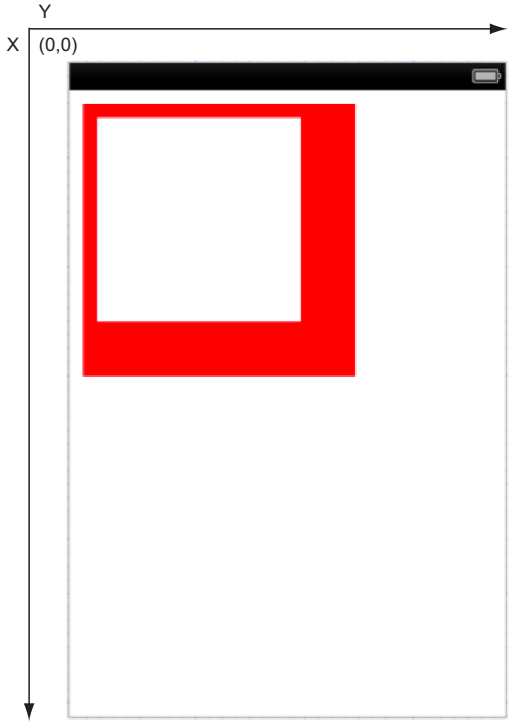

Figure 2.12 A view added as a subview to our red view at (10,10) with a size of 150 x 150

shown in figure 2.11, it assumes a new coordinate system relative to its parent. This means that the origin point (0,0) is at the very top left of the red view.

When you rotate the Hello Time app, you're retrieving the frame of `timeLabel` and editing its Y origin. The new Y origin point was calculated by referencing the height of its parent view. By editing its frame you're able to change its vertical positioning to center it within the view depending on the orientation.

Now that you've learned about the view coordinate system, you can learn more about different UI controls.

2.2.3 User interface controls

User interface controls (or simply *controls*) are represented by the `UIControl` class. Controls are interface elements that a user can view and interact with. You've interacted with many different types of controls when using iOS apps. Examples of different types of controls include labels, buttons, sliders, text fields, selectors, activity indicators, and search bars. Various controls as listed within Xcode are shown in figure 2.13.

To visualize the hierarchy of a control, take a look at figure 2.14.

When you created the Hello Time app, you used the Single View Application template. This created a single view controller that had a view attached to it. You've been editing this view within Main.storyboard. When you were dragging in your label and your button to this view, you were adding them as subviews.

By doing this programmatically you can see what goes into adding a subview. You'd need to pass in a frame to the `alloc:initWithFrame:` method, which is found on most `UIView` subclasses:

```
CGRect frame = CGRectMake(0,0,100,20);
UILabel *label = [[UILabel alloc] initWithFrame:frame];
```

To add this label to its parent view within a view controller, all you have to do is use the `addSubView:` function.

```
[self.view addSubView:label];
```

This will add the label as a subview and position it at (0,0) with a width of 100 and height of 20. You can make changes to this frame as you did in Hello Time by retrieving the frame, making a change to its size or origin, and then resetting its frame property:

```
CGRect frame = label.frame;
frame.origin.x = 10;
frame.size.width = 200;
[label setFrame:frame];
```

In this example, you're retrieving the frame, and setting it's X origin to 10 and its width to 200. You're then resetting the label's `frame` property with the newly modified frame.

2.2.4 Responding to actions and events

In Hello Time you created an action for the button that you used to toggle between night and day modes. This button triggered an action you defined, called `toggleMode:`,

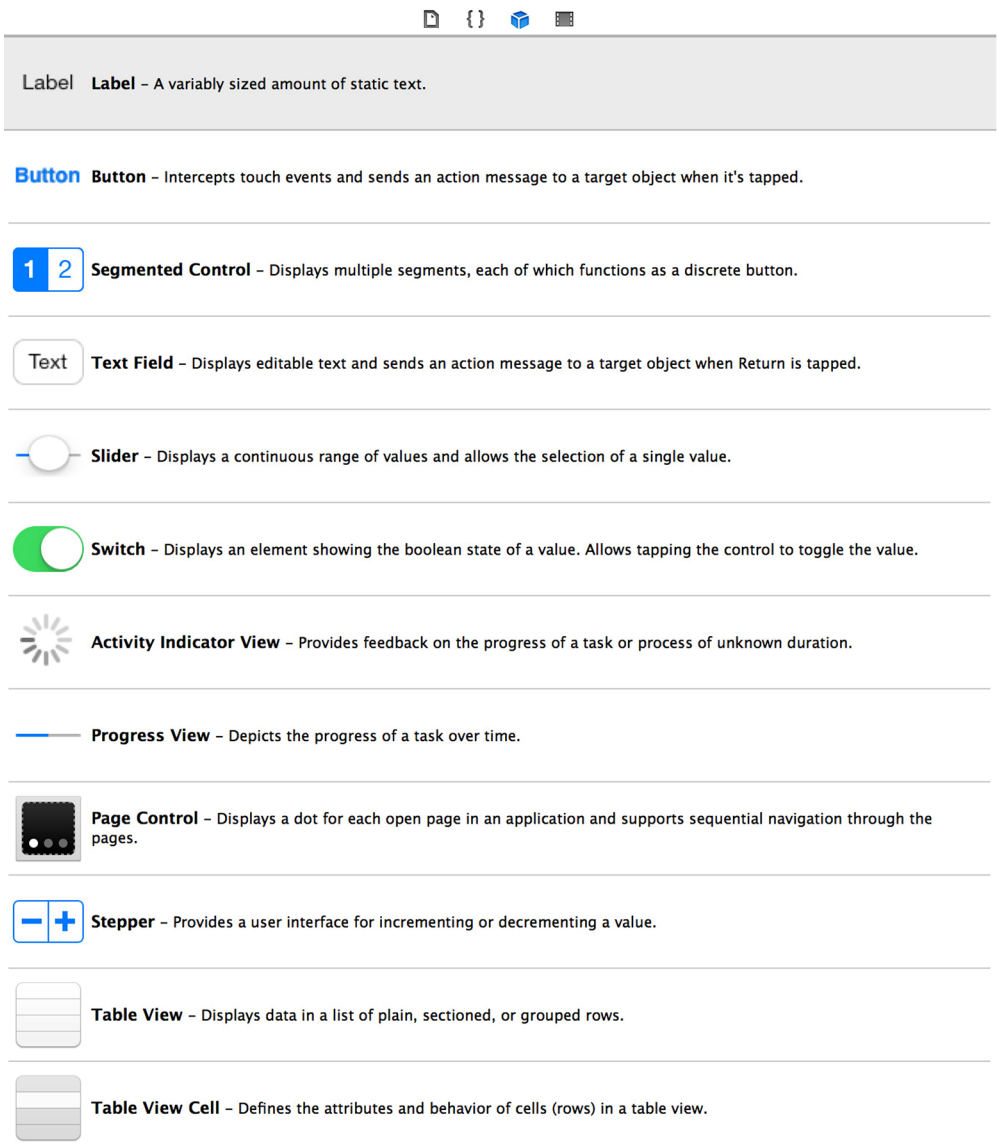

Figure 2.13 List of different user interface controls as shown in Xcode's interface builder

whenever the button was touched. The specific event you used was Touch Up Inside, which is the default event to respond to when someone touches a button. You used Xcode's interface tools by dragging an action connection from the button to your view controller.

Take a second look at the `toggleMode:` action that you have within `IAViewController`. In the method definition it's specified as an `IBAction`.

```
- (IBAction) toggleMode:(id) sender
```

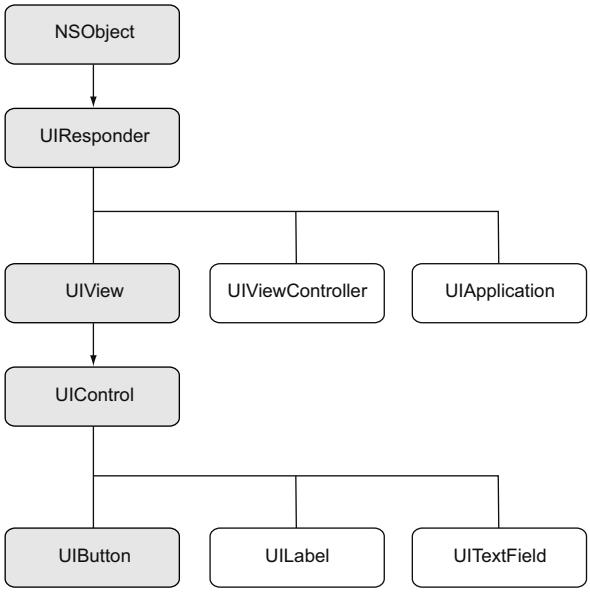

Figure 2.14 Class hierarchy of a `UIResponder` as well as subclasses of a `UIView` and `UIControl`

It also takes in one parameter named `sender`. The object that triggered the action usually fills the `sender` parameter. For instance, your button, `modeButton`, would be accessible through the `sender` parameter within that action because it triggered its execution. If you wanted it to trigger another action for a specific event, you could create another connection through the interface tools, or you could do it programmatically.

By doing it programmatically you can see what's being done to trigger an action for a specific event. Say, for instance, you have another action called `turnRed:`, which changes the color of your label to red. If you want this to happen when your button is touched, you could add the following code:

```
[self.modeButton addTarget:self action:@selector(turnRed:)
➥ forControlEvents:UIControlEventTouchUpInside];
```

This will call the `turnRed:` method on your current class when the Touch Up Inside event is triggered. You'll also notice that for the `action` parameter, you're using `@selector(turnRed:)` instead of just `turnRed:`. The `@selector()` function returns a `SEL`, or selector, which allows you to reference a particular method or action. A selector is essentially a pointer to a method. For example, the previous code example could be written as follows:

```
SEL turnRedSelector = @selector(turnRed:)
[self.modeButton addTarget:self action:turnRedSelector
➥ forControlEvents:UIControlEventTouchUpInside];
```

As we go along, you'll be using controls other than buttons and responding to different types of actions.

2.2.5 Custom tint colors

iOS 7 introduced tint colors to `UIViews`, which are used to define key colors that are used to represent interactivity for user interface elements. Adding a tint color to a view also changes the tint color for all of its subviews. Without manually setting a tint color, the default color will be blue, as you can see in the Hello Time application. The controls that you've added are all shown using the default tint color.

To change the tint color of a view, you just have to modify its `tintColor` property. Shown here is how you'd set the tint color of a particular view and its subviews to red:

```
view.tintColor = [UIColor redColor];
```

If you wanted to change the tint color of an entire application, you could just update the tint color in its `UIWindow`. This is because the `UIWindow` contains all of the subviews within an application. This is shown in the following code example:

```
self.window.tintColor = [UIColor redColor];
```

It's also possible to change the tint color of a view within the interface editor in Xcode. By selecting a view, you can modify its tint color in the attributes inspector. Next, let's take a closer look at view controllers.

2.3 View controller basics

View controllers manage each separate view in your application, which are the screens you see in an iOS application. Although managing views is what a view controller does, that's not its only responsibility. View controllers also handle transitions between other view controllers and transferring data between them. When you tap a button or any other type of control and are transitioned from one view to another, a view controller handles that transition.

2.3.1 Introducing view controllers

Simply put, a view controller manages your views and segues between view controllers. They are a core component of each iOS app and act as the glue between your views and your models. They initialize and set up your models and populate your views. View controllers are also the controller objects in the MVC pattern.

If you go back to the Hello Time project in Xcode, you'll see that two methods were already declared for you: `viewDidLoad` and `didReceiveMemoryWarning`. Both names are self-explanatory. One is for handling when the view has already been loaded, and the other is for when you need to handle a situation where you received a memory warning. These two methods help with the view controller's lifecycle. Let's explore this, and you'll see how you can override certain methods to help manage your controllers.

Naming conventions for view controllers

It's good practice to use very specific names for view controllers throughout your project. This makes maintenance and reusability much easier to manage. It also makes it easy to work with other developers. In your Hello Time project, you're using `IAViewController` for your main view. This is because it was generated for you using the Single View Application template. You can change its name if you want to. If you were building a photo-picker controller for a photo-sharing application, you could name it something like `IAPhotoSelectionViewController`.

2.3.2 The view controller lifecycle

Understanding the lifecycle of a view controller allows you to properly manage the models and views contained within it. It helps you understand when your view will be ready for you to manipulate and when you should have to clean things up when your view is about to be removed. Having a deep knowledge of the view lifecycle is a core part of becoming a great iOS developer because everything revolves around view controllers and using them effectively.

When any part of your application asks a specific view controller for its `view` property, it will kick off a chain of events. If the `view` property for this view controller is not yet loaded into memory, it will call a method called `loadView`. Once this view is fully loaded, it will call the `viewDidLoad` method. This is where you can start initializing any data needed for your views. You can see this flow in figure 2.15.

Looking at the flow for retrieving and setting up a view in a view controller, you can see that you can override the `loadView` function to programmatically create a `UIView` to set as the view controller's `view` property. This is opposed to loading it from a nib or from a storyboard, as you've been doing in Hello Time.

There are a few more methods throughout the lifespan of a view controller that are triggered by various view events. These methods give you more precise control throughout a view controller view's different states. They are listed in table 2.1.

Table 2.1 Methods related to the lifecycle of a `UIViewController`

Method	Description
`loadView`	Creates or returns a view for the view controller.
`viewDidLoad`	View has finished loading.
`viewWillAppear:`	View is about to appear with or without animation.
`viewDidAppear:`	View did appear with or without animation.
`viewWillDisappear:`	View is about to disappear with or without animation.
`viewDidDisappear:`	View did disappear with or without animation.
`viewWillLayoutSubviews`	View is about to lay out its subviews.
`viewDidLayoutSubviews`	View did lay out its subviews.
`didReceiveMemoryWarning`	View detected low memory conditions.

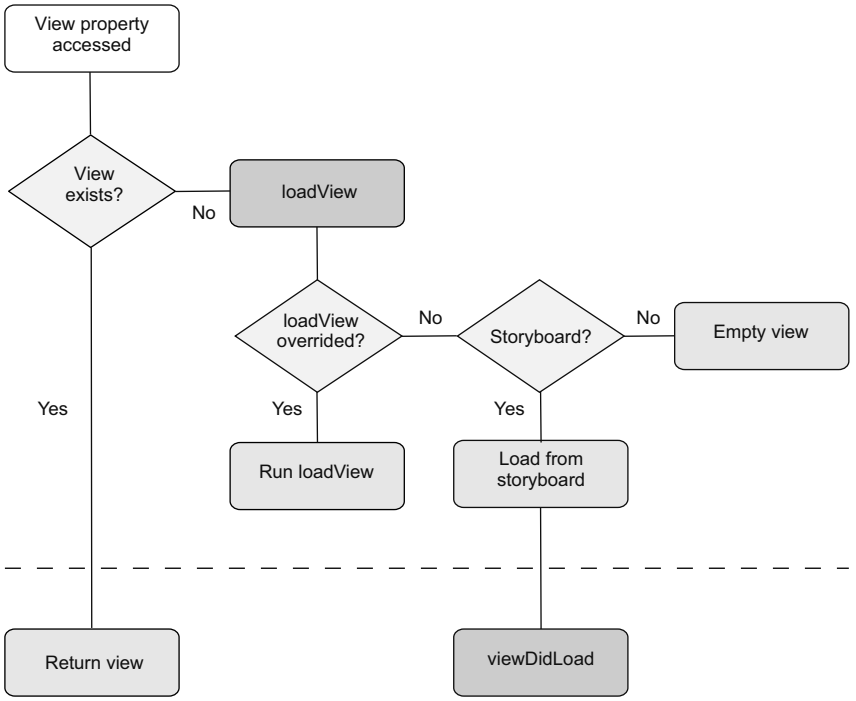

Figure 2.15 The flow for retrieving and setting up a view in a view controller

Because of the verbosity of Cocoa's method names, it's rather simple to understand when each of these methods is called. Within `IAViewController` in your Hello Time project, you added a call to `checkTime:` at the end of the `viewDidLoad` method. This was added here because you want to trigger the time check *after* your view is ready to be used.

Let's go one step further and add something else to the bottom of `viewDidLoad`. You'll add a log statement so you can see when it's being triggered. Add the following code to the bottom, underneath the call you added to check the time:

```
NSLog(@"viewDidLoad called");
```

Now override `viewWillAppear:` and `viewDidAppear:` by adding the following code:

```
- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
    NSLog(@"viewDidAppear: called");
}

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    NSLog(@"viewWillAppear: called");
}
```

Run the application and look at the console within Xcode. You can activate the console by pressing Shift-Command-C or by choosing View > Debug Area > Activate Console. You'll then see the following output:

```
Hello Time [12139:c07] viewDidLoad called
Hello Time [12139:c07] viewWillAppear: called
Hello Time [12139:c07] viewDidAppear: called
```

From this you can see that the first method that's called is `viewDidLoad`, followed by `viewWillAppear:`, and then finally `viewDidAppear:`. Before `viewDidLoad` is triggered, your view controller checks to see if `loadView` has been overridden. In this case it's not because it's being loaded from the storyboard. You could also go one step further and add in calls to see when your view disappears by adding log statements to `viewWillDisapear:` and `viewDidDisappear:`. These would normally be triggered when you decided to load another view controller in your application.

You've used only a basic view controller, but there are many different types to choose from. Apple provides many for you to use for different types of applications.

2.3.3 Different types of view controllers

There are different types of view controllers that you can use without having to write your own from scratch. Some of these view controllers help you manage data by laying it out in a table or grid format. Others manage other view controllers that allow you to display them in a tab-based layout or in a hierarchical structure. All of these view controllers are listed in table 2.2.

Table 2.2 Different out-of-the-box subclasses of `UIViewController` in iOS

UIViewController subclass	Description
`UINavigationController`	Manages navigation of hierarchical view controllers
`UITabBarController`	Represents and manages multiple view controllers as tabs
`UITableViewController`	Presents and manages data represented as a table
`UICollectionViewController`	Presents and manages data represented as a collection

You've encountered most of these while using other iOS apps. Let's take a quick high-level look at each of these to see how they differ, starting with `UINavigationControllers`.

NAVIGATION CONTROLLERS

Navigation controllers (`UINavigationController`) handle the display of data hierarchically using multiple view controllers within a stack. The first view controller within a navigation controller is referred to as the root view controller. When displaying a view, you can push another view onto the stack. The navigation controller allows you to go back by popping the current view off the stack until you reach the

Figure 2.16 A navigation controller in action within the default Settings app

root view. Figure 2.16 shows a navigation controller in action within the default Settings app.

As you can see in the figure, the Settings app uses a navigation controller. By tapping on Twitter, a new view is pushed onto the navigation controller and brought into display. A back button with the label Settings is also added to this newly pushed view. From there you can go one level deeper by clicking a Twitter account, which pushes yet another view.

TAB BAR VIEW CONTROLLERS

Tab bar view controllers (`UITabBarViewController`) offer a simple way to segment different view controllers for your users. The identifying element of a tab bar controller is, of course, the tab bar. The tab bar view controller is used in Apple's App Store application, as shown in figure 2.17.

This tab bar (`UITabBar`) can contain several tab bar items (`UITabBarItem`). These tab bar items contain an icon and a title used to represent a specific view that it will display once activated. Each tab contains a single view controller, which can have its own hierarchy. For example, a view controller contained within a tab can be a `UINavigationController`.

TABLE VIEW CONTROLLERS

Table view controllers (`UITableViewController`) contain a single table view. Table views allow you to display data as a list of rows represented by table view cells. These rows of data can also be visually separated and sectioned into groups.

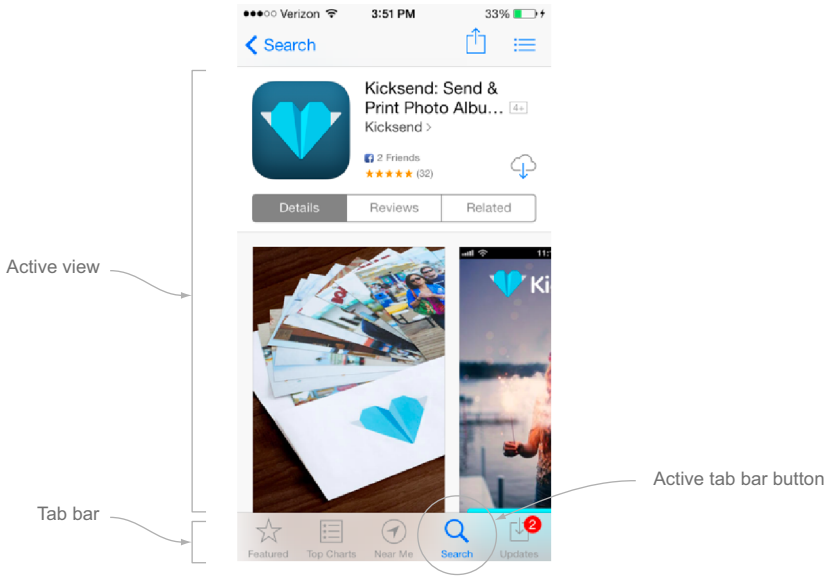

Figure 2.17 `UITabBarViewController` used in Apple's App Store application

A table view controller does much of the table view setup for you and doesn't make you worry about manually plugging in the right methods to feed your table view with data. It's common to use a regular view controller and manually add a table view to it without relying on a table view controller to do it for you. Depending on the kind of data you're displaying or the complexity of your app, you may want more control over how your data is set up.

Figure 2.18 shows how a table view controller would look if it contained a grouped table view with multiple sections.

It's virtually impossible to tell the difference between a table view controller and a regular view controller with a single table view attached because they both display a table view.

So far you've dealt with both portrait and landscape orientations in the Hello Time app. Now you'll learn more about supporting different types of orientations.

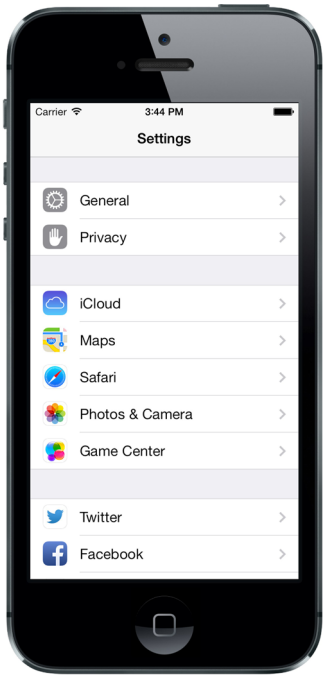

Figure 2.18 Table view shown within a table view controller as seen in the Settings application

2.3.4 Different status bar styles

The status bar includes the time, battery charge, network, and Wi-Fi information. In iOS 6 the status bar

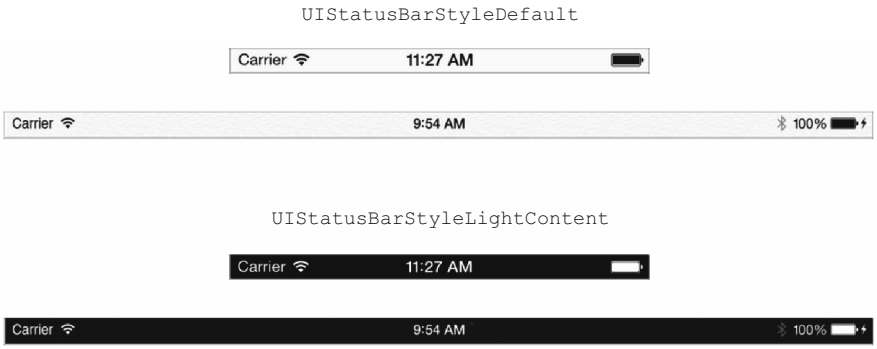

Figure 2.19 Different status bar styles that should be used depending on the type of content shown

had a specific tint that was separate from the view directly underneath it. In iOS 7 this changed because the status bar is transparent and shows the view behind it. Different status bar styles are available that you can specify using a `UIStatusBarStyle` constant. One specifies whether the *content* should be dark (`UIStatusBarStyleDefault`) or light (`UIStatusBarStyleLightContent`). This is shown in figure 2.19.

The status bar style can be globally set in the project's settings in the General tab within Xcode. There are times where you might not want to have your status bar be the same style throughout each view controller in your application. For example, you could have one view where you're showing light content and another where you're showing dark content. To do this you'd first have to set the `UIViewControllerBasedStatusBarAppearance` key to `YES` in your application's `plist` file.

Next, you'd need to implement the `preferredStatusBarStyle` method in your view controller. This method expects to return a `UIStatusBarStyle` constant, which means if you wanted to use a status bar for a screen with light content, you'd return `UIStatusBarStyleLightContent`, as shown here:

```
- (UIStatusBarStyle)preferredStatusBarStyle
{
    return UIStatusBarStyleLightContent;
}
```

To trigger this method, you'll have to call `[self setNeedsStatusBarAppearanceUpdate]` within your view controller. You can do this within `viewDidLoad` or anywhere else that would warrant a new status bar style. Next, we'll explore how to support different types of orientations.

Extended layout support in iOS 7

In iOS 7 content extends from edge to edge of the screen as opposed to iOS 6. Content is also displayed underneath navigation and tab bars so that it fills the screen. It's possible to not have your content go from edge to edge by updating the `edgesForExtendedLayout` property on a `UIViewController`. You could set this to `UIRectEdgeNone` if you don't wish to support extended layout or to `UIRectEdgeAll` if you do.

2.4 Supporting different orientations

An iOS app can be used in either portrait or landscape mode. Portrait mode is the default mode for any new application you create for Xcode. Landscape mode is more popularly used for viewing photos or videos in full screen. Depending on the type of application you're building and the type of device (iPhone or iPad), you could choose to support either orientation or just one. When supporting both orientations you have to add in functionality so that your views react accordingly to the change in width and height of the new orientation.

2.4.1 Enabling support for portrait and landscape

There are four different types of orientations you can choose to support: portrait, upside-down portrait, landscape left, and landscape right. In Hello Time you're supporting three different types of orientations by default. You can see this by going to the project navigator in Xcode and viewing your target's general settings, as shown in figure 2.20.

By default, upside-down portrait is not supported, but you can easily enable it by clicking it inside your target's Summary tab in Xcode. By doing this, all of your views won't automatically resize when the device's orientation changes. You also still need to specify in each specific view controller what orientations are supported. This is because you may have certain view controllers where you would like to support only portrait and another where you may want to play video and support portrait *and* landscape. In Hello Time you specified which orientations you support by adding the following code using the `supportedInterfaceOrientations` method:

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

You could have used the `UIInterfaceOrientation` enumerable type to specify each specific orientation. All of these are listed in table 2.3.

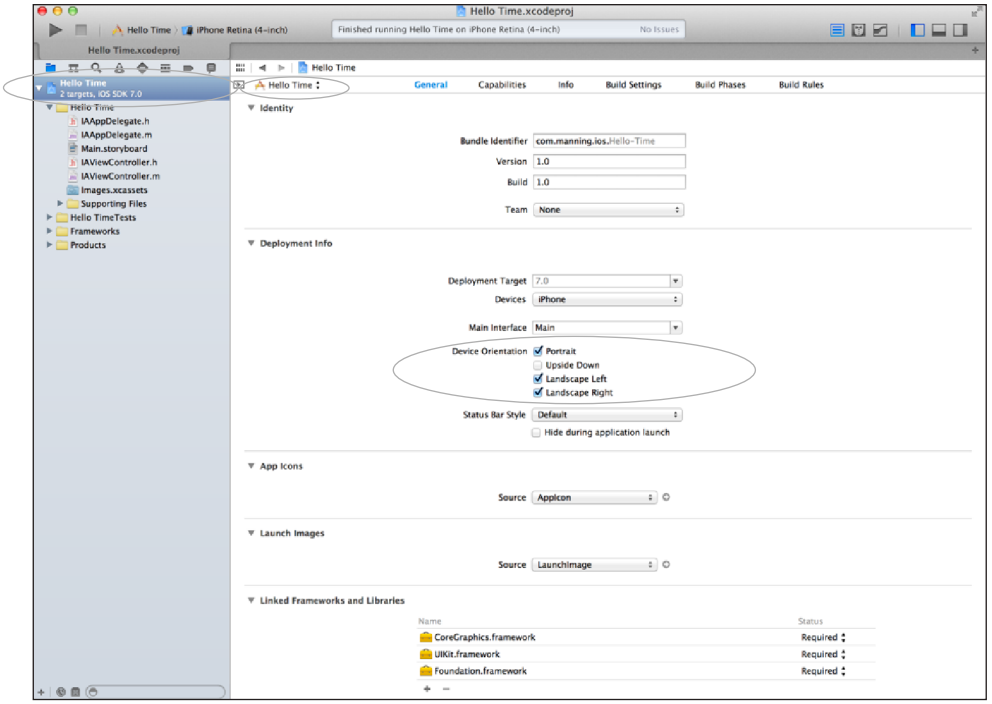

Figure 2.20 The three different orientations supported by Hello Time

Table 2.3 `UIInterfaceOrientation` enumerable for representing different orientations

UIInterfaceOrientation	Orientation description
`UIInterfaceOrientationPortrait`	Portrait
`UIInterfaceOrientationPortraitUpsideDown`	Upside-down portrait
`UIInterfaceOrientationLandscapeLeft`	Left landscape
`UIInterfaceOrientationLandscapeRight`	Right landscape

In Hello Time, you used a mask to represent the supported orientations. This allowed you to combine all portrait and landscape orientations instead of writing them all out. These orientation bit masks are used to represent multiple orientation combinations, as shown in table 2.4.

Table 2.4 `UIInterfaceOrientationMask` enumerable for representing different orientation combinations

UIInterfaceOrientationMask	UIInterfaceOrientations supported
`UIInterfaceOrientationMaskPortrait`	Portrait
`UIInterfaceOrientationMaskPortraitUpsideDown`	Upside-down portrait

Table 2.4 `UIInterfaceOrientationMask` enumerable for representing different orientation combinations *(continued)*

UIInterfaceOrientationMask	UIInterfaceOrientations supported
`UIInterfaceOrientationMaskLandscapeLeft`	Left landscape
`UIInterfaceOrientationMaskLandscapeRight`	Right landscape
`UIInterfaceOrientationMaskPortraitAll`	Portrait, upside-down portrait
`UIInterfaceOrientationMaskLandscapeAll`	Left landscape, right landscape
`UIInterfaceOrientationMaskAll`	Portrait, upside-down portrait, left landscape, right landscape

If you wanted to support just portrait (and not also upside-down portrait) within your view controller, you'd add the following:

```
-(NSUInteger) supportedInterfaceOrientations
{
    return UIInterfaceOrientationPortrait;
}
```

If you wanted to support both portrait and landscape orientations, you could change the `supportedInterfaceOrientations` method to the following:

```
-(NSUInteger) supportedInterfaceOrientations
{
    return UIInterfaceOrientationMaskAll;
}
```

2.4.2 Updating your views for different orientations

When you added support for landscape, your views didn't support the new orientation automatically. When you first positioned the label and button for your views, you did it for the portrait orientation. After adding support for the landscape orientation, you saw how they looked when the device was rotated, as shown in figure 2.21.

You then added a method to your view controller that allowed you to know when the orientation of your view controller changed. Adding the `willAnimateRotationToInterfaceOrientation:toInterfaceOrientation:duration:` method gave you the opportunity to make adjustments to the positioning of your views:

Figure 2.21 The views in the Hello Time application didn't automatically support the landscape orientation.

```
- (void) willAnimateRotationToInterfaceOrientation:
➥ (UIInterfaceOrientation)toInterfaceOrientation
➥ duration:(NSTimeInterval)duration
{
    CGRect timeFrame = self.timeLabel.frame;
    float viewHeight = self.view.frame.size.height;
    float viewWidth = self.view.frame.size.width;
    float fontSize = 30.0f;
    BOOL hideButton = YES;

    if (UIInterfaceOrientationIsLandscape(self.interfaceOrientation)) {
        fontSize = 40.0f;
        timeFrame.origin.y = (viewWidth / 2) - timeFrame.size.height;
    } else {
        hideButton = NO;
        timeFrame.origin.y = (viewHeight / 2) - timeFrame.size.height;
    }

    [self.modeButton setHidden:hideButton];
    [self.timeLabel setFont:[UIFont boldSystemFontOfSize:fontSize]];
    [self.timeLabel setFrame:timeFrame];
}
```

This method is automatically called in your view controller when the device is about to be rotated. In this method you checked to see what orientation your view was in and adjusted your label's frame accordingly. You also took the opportunity to change the background color of your main view and to hide the mode button. Something you could have done to avoid having to adjust the frame of each view as you did was to use auto layout.

Auto layout was introduced in iOS 6 and is helpful with situations like this. It allows you to specify certain constraints on your views so that they can reposition themselves depending on screen size or orientation. There are still times when you'll want to do more to your views when viewing them in a different orientation. You'll learn about auto layout and storyboarding in the next chapter.

2.5 Summary

We've only skimmed the surface of views and view controllers. As we progress, we'll be diving deeper into different ways of working with views and using advanced view controllers to help you make more immersive, rich applications.

- You saw how to position views within the view coordinate system.
- Views can be added and modified programmatically and even nested within one another.
- By knowing your view controller's lifecycle, you know how and when to properly manage the models and views it contains.
- Custom tint colors can be applied to a `UIView`, which will be applied to all of its subviews.
- Different types of view controllers are available to you out of the box, such as the `UITableViewController`, `UITabBarViewController`, and `UINavigationController`.

- You can create actions and connect them to respond to a specific event on a `UIControl` such as a button.
- You can specify different status bar styles per view controller.
- You can support two different types of portrait and landscape orientations.
- It's possible to alter your views depending on the orientation of the device.
- You updated your Hello Time application by adding a separate night mode as well as support for landscape orientation.

iOS DEVELOPMENT

iOS 7 IN ACTION

Lim • Mac Donell

To develop great apps you need a deep knowledge of iOS. You also need a finely tuned sense of what motivates 500 million loyal iPhone and iPad users. iOS 7 introduces many new visual changes, as well as better multitasking, dynamic motion effects, and much more. This book helps you use those features in apps that will delight your users.

iOS 7 in Action is a hands-on guide that teaches you to create amazing native iOS apps. In it, you'll explore thoroughly explained examples that you can expand and reuse. If this is your first foray into mobile development, you'll get the skills you need to go from idea to app store. If you're already creating iOS apps, you'll pick up new techniques to hone your craft, and learn how to capitalize on new iOS 7 features.

What's Inside

- Native iOS 7 design and development
- Learn Core Data, AirPlay, Motion Effects, and more
- Create real-world apps using each core topic
- Use and create your own custom views
- Introduction and overview of Objective-C

This book assumes you're familiar with a language like C, C++, or Java. Prior experience with Objective-C and iOS is helpful.

Brendan Lim is a Y Combinator alum, the cofounder of Kicksend, and the author of *MacRuby in Action*.
Martin Conte Mac Donell, aka fz, is a veteran of several startups and an avid open source contributor.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/iOS7inAction

"A practical journey through the iOS 7 SDK."
—Stephen Wakely
Thomson Reuters

"A kickstart for newbs and a deft guide for experts."
—Mayur S. Patil
Clearlogy Solutions

"Mobile developer: don't you dare not read this book!"
—Ecil Teodoro, IBM

"The code examples are excellent and the methodology used is clear and concise."
—Gavin Whyte
Verify Data Pty Ltd

"Everything you need to know to ship an app, and more."
—Daniel Zajork
API Healthcare Corporation

ISBN 13: 978-1-617291-42-5
ISBN 10: 1-617291-42-0
54499
9 781617 291425

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]