

Introduction to SDK Development

A detailed illustration of a woman in traditional Indian attire, including a white sari with floral patterns and a white dupatta. She is holding a tulsi garland (a string of small flowers) in her right hand. The illustration is set against a dark red background on the left side of the cover.

iPhone and iPad IN ACTION

Revised edition of *iPhone in Action*

Brandon Trebitowski
Christopher Allen
Shannon Appelcline

SAMPLE CHAPTER

 **MANNING**



iPhone and iPad in Action

by Brandon Trebitowski

Christopher Allen

Shannon Appelcline

Chapter 5

brief contents

- 1 ■ Introducing the iPhone and iPad 1
- 2 ■ Learning Objective-C and the iPhone OS SDK 13
- 3 ■ Using Xcode 36
- 4 ■ Using Interface Builder 53
- 5 ■ Creating basic view controllers 68
- 6 ■ Monitoring events and actions 87
- 7 ■ Creating advanced view controllers 111
- 8 ■ Data: actions, preferences, and files 139
- 9 ■ Data: advanced techniques 159
- 10 ■ Positioning: accelerometers, location, and the compass 188
- 11 ■ Media: images and the camera 210
- 12 ■ Media: audio and recording 224
- 13 ■ Graphics: Quartz, Core Animation, and OpenGL 243
- 14 ■ The web: web views and internet protocols 271
- 15 ■ Peer-to-peer connections using Game Kit 295
- 16 ■ Push notification services 316
- 17 ■ The Map Kit framework 328
- 18 ■ In-app purchasing using Store Kit 342
- 19 ■ iPhone SDK enhancements 357

5

Creating basic view controllers

This chapter covers

- Understanding the importance of controllers
- Programming bare view controllers
- Utilizing table view controllers

In the last two chapters, we've offered a hands-on look at the two core tools used to program using the SDK: Xcode and Interface Builder. In the process, we haven't strayed far from the most fundamental building block of the SDK: the view, whether a UILabel, a UIWebView, or a UIImageView.

Ultimately, the view is only part of the story. As we mentioned when we looked at the iPhone OS, views are usually connected to view controllers, which manage events and otherwise take the controller role in the MVC model. We're now ready to begin a three-part exploration of what that all means.

In this chapter, we look at basic view controllers that manage a single page of text. With that basis, we can examine events and actions in chapter 6, correctly integrating them into the MVC model. Finally, in chapter 7, we'll return to the topic of view controllers to look at advanced classes that can be used to connect up several pages of text.

Over the course of our two view controller chapters (5 and 7), we'll offer code samples that are a bit more skeletal than usual. That's because we want to provide you with the fundamental, reusable code that you'll need to use the controllers on your own. Consider chapters 5 and 7 more of a reference—although a critical one. You'll make real-world use of the controllers in the rest of this book, including when we look at events and actions in chapter 6. Right now, though, let's examine the available view controllers.

5.1 The view controller family

When we first talked about view controllers in chapter 2, we mentioned that they come in several flavors. These run from the bare-bones `UIViewController`, which is primarily useful for managing autorotation and for taking the appropriate role in the MVC model, to the more organized `UITableViewController`, on to a few different controllers that allow navigation across multiple pages.

All of these view controllers—and their related views—are listed in table 5.1.

Table 5.1 There are a variety of view controllers, giving you considerable control over how navigation occurs in your program.

| Object | Type | Summary |
|-------------------------------------|-----------------|--|
| <code>UIViewController</code> | View controller | A default controller, which controls a view. Also the basis for the flipside controller, which appears only as an Xcode template, not as a UIKit object. |
| <code>UIView</code> | View | Either your full screen or some part thereof. This is what a view controller controls, typically through some child of <code>UIView</code> , not this object itself. |
| <code>UITableViewController</code> | View controller | A controller that uses <code>UITableView</code> to organize data listings. |
| <code>UITableView</code> | View | A view that works with the <code>UITableViewController</code> to create a table UI. It contains <code>UITableCells</code> . |
| <code>UITabBarController</code> | View controller | A controller that works with a <code>UITabBar</code> to control multiple <code>UITableViewController</code> s. |
| <code>UITabBar</code> | View | A view that works with the <code>UITabBarController</code> to create the tab bar UI. It contains <code>UITabBarItem</code> s. |
| <code>UINavigationController</code> | View controller | A controller used with a <code>UINavigationController</code> to control multiple <code>UITableViewController</code> s. |

Table 5.1 There are a variety of view controllers, giving you considerable control over how navigation occurs in your program. (continued)

| Object | Type | Summary |
|---|-----------------|--|
| UINavigationController | View | A view that works with UINavigationController to create the navigation UI. |
| Flipside controller | View controller | A special template that supports a two-sided UIViewController. |
| ABPeoplePickerNavigationController ABNewPersonViewController ABPersonViewController ABUnknownPersonViewController UIImagePickerController | View controller | Modal view controllers that allow interaction with sophisticated user interfaces for the Address Book and the photos roll. |

As we've already noted, we'll be discussing these view controllers in two different chapters. Here, we'll look at the single-page view controllers: UIViewController and UITableViewController. In chapter 7, we'll examine the multipage view controllers: UITabBarController, UINavigationController, and the flipside controller. This is a clear functional split: the single-page controllers exist primarily to support the controller role of the MVC model, whereas the multipage controllers exist primarily to support navigation and may even delegate MVC work to a simpler view controller lying below them. (As for the modal controllers, we'll get to them when we cover the appropriate topics in chapters 8 and 11.)

So far, you've been programming without using view controllers, which are an important part of SDK programming. You *could* write an SDK program without them, but every SDK program *should* include them, even if you use a bare-bones view controller to manage the rotation of the screen.

5.2 The standard view controller

The plain view controller is simple to embed inside your program. By why would you want to use a view controller? That's going to be one of the topics we'll cover here. Now, we'll look at how view controllers fit into the view hierarchy, how you create them, how you expand them, and how you make active use of them. Let's get started with the most basic anatomical look at the view controller.

5.2.1 The anatomy of a view controller

A view controller is a UIViewController object that sits immediately above a view (of any sort). It, in turn, sits below some other object as part of the tree that ultimately goes back to an application's main window. This is shown in figure 5.1.

When we move on to advanced view controllers in chapter 7, you'll see that the use of a bare

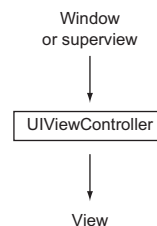


Figure 5.1 A bare view controller shows view controlling at its simplest: it sits below one object and above another.

view controller can grow more complex. Bare view controllers often sit beneath advanced view controllers, to take care of the individual pages that advanced view controllers let you navigate among.

Looking at the iPhone OS's class hierarchy, you can see that the `UIViewController` is a direct descendent of `NSObject`. That means it doesn't get any of the functionality of `UIResponder` or `UIView`, which you find in most other UIKit objects. It's also the parent object of all the other view controllers we'll discuss. Practically, this means that the lessons learned here also apply to all the other controllers.

But learning about how a view controller works leaves out one vital component: how do you create it?

5.2.2 **Creating a view controller**

The easiest way to incorporate a plain view controller into your project is to select a different template when you create it. The View-Based Application template should probably be your default template for programming from here on out, because it comes with a view controller built in.

As usual, the template's work is primarily done through Interface Builder. When you create a new project (which we've called `viewex` for the purpose of this example), you can verify this by looking up the view controller's `IBOutlet` command in the program's app delegate header file:

```
ViewexViewController *viewController;
```

The app delegate's source code file further shows that the view controller's view has already been hooked up to the main window:

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

This view is a standard `UIView` that's created as part of the template. Although a view controller has only one view, that view may have a variety of subviews, spreading out into a hierarchy. We'll show you how to add a single object beneath the view in a moment, and you'll make more complete use of it in the next chapter. But before we get there, we want to step back and look at how you can create a view controller by hand if you need to.

5.2.3 **Creating another view controller**

Creating another view controller is simple. First, in Interface Builder, drag a view controller from the Library to your xib document window. Alternatively, in Xcode, you can `alloc` and `init` an object from the `UIViewController` class.

NOTE Increasingly, we'll assume that you're doing work through IB and using appropriate templates. But the same methods for object creation that you learned in the last couple of chapters remain available for all objects.

Second, note that the previous `IBOutlet` command shows that the controller isn't instantiated directly from the `UIViewController` class. Rather, it's instantiated from

its own subclass, which has its own set of files (`viewexViewController.{h|m}`), named after the example project's name. This is standard operating procedure.

Because you want a view controller to do event management, you'll often need to modify some of the controller's standard event methods, so you require your own subclass. To start, the view controller class files are mostly blank, but Xcode helpfully highlights a number of standard view controller methods that you may want to modify.

After you've finished creating a bare view controller, you're mostly ready to go. But you have a slight opportunity to modify the view controller for your specific program, and that's what we'll cover next.

5.2.4 *Building up a view controller interface*

In order to correctly use a view controller, you need to build your view objects as subviews of the view controller, rather than subviews of your main window or whatever else lies above it. This is easy in both Xcode and Interface Builder.


THE XCODE SOLUTION

The view controller class file gives you access to a pair of methods that can be used to set up your view controller's views. If the view controller's view is linked to a .xib file, you should use `viewDidLoad`, which will do additional work after the .xib is done loading; if it isn't created from inside Interface Builder, you should instead use `loadView`.

Before you do any of this, your view controller will always start off with a standard `UIView` as its one subview. But by using these methods, you can instead create the view controller's view as you see fit, even creating a whole hierarchy of subviews if you desire.

The following code adds a simple `UILabel` to your view controller using `viewDidLoad`. We've chosen a humongous font that is automatically sized down so that later we can show off how rotation and resizing work:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    UILabel *myLabel = [[UILabel alloc]
        initWithFrame:[UIScreen mainScreen] bounds]];
    myLabel.adjustsFontSizeToFitWidth = YES;
    myLabel.font = [UIFont fontWithName:@"Arial" size:60];
    myLabel.textAlignment = UITextAlignmentCenter;
    myLabel.text = @"View Controllers!";
    myLabel.backgroundColor = [UIColor grayColor];
    [self.view addSubview:myLabel];
    [myLabel release];
}
```



1 Connects label as subview

The `self.view` line is the only one of particular note **1**. It connects your label object as a subview of the view controller's `UIView`.

This example is also noteworthy because it's the first time you've definitively moved outside of your app delegate for object creation. You could have done this object creation in the app delegate, but that's often sloppy programming because this needs to be done in the view controller. Now that you have view controllers, you'll increasingly do your work in those class files. This not only better abstracts your object

creation but also kicks off your support of the MVC model, because you now have controllers instantiating the views they manage. Watch for a lot more of this in the future. We'll also briefly return to the `viewDidLoad` and `loadView` methods when we talk about the bigger picture of the view controller lifecycle, shortly.

THE INTERFACE BUILDER SOLUTION

In the last chapter, we noted that view controllers often have their own `.xib` files, allowing you to have one `.xib` file for each page of content. That's what's going on in the program you created from the View-Based Application template. At creation, the template contains two `.xib` files: `MainWindow.xib` and `viewexViewController.xib`.

The `MainWindow.xib` file contains a view controller and a window. The all-important link to the second `.xib` file can be found here. If you click the view controller's Attribute tab, it helpfully shows you that the controller's content is drawn from `viewexViewController(.xib)`. This is shown in figure 5.2.

Now that you understand the hierarchy of `.xib` files that's been set up, how do you make use of them? In order to create an object as a subview of the view controller, you need to place it inside the `.xib` file that the view controller manages—in this case, `viewexViewController.xib`. To add a `UILabel` to your view controller, you call up the `viewexViewController.xib` file and then drag a label to the main display window, which should represent the existing view. Afterward, you can muck with the label's specifics in the inspector window, as usual.

Practically, there's nothing more you need to do to set up your basic view controller, but we still need to consider a few runtime fundamentals.

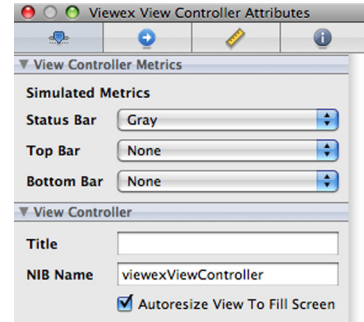


Figure 5.2 To hook up a new `.xib` file to a view controller, enter its name in the view controller's attributes under NIB Name.

5.2.5 Using your view controller

If you've chosen to use a standard view controller, it should be because you're only managing one page of content, not a hierarchy of pages. In this situation, you don't need your view controller to do a lot, but your view controller is still important for three things, all related to event management:

- It should act as the hub for controlling its view and subviews, following the MVC model. To do this, it needs easy access to object names from its hierarchy.
- It should control the rotation of its view, which will also require resizing the view in rational ways. Similarly, it should report back on the device's orientation if queried.
- It should deal with lifecycle events related to its view.

We've split these main requirements into six topics, which we'll cover in turn.

PUTTING THE MVC MODEL TO USE

Although we've talked about the Model-View-Controller (MVC) architectural pattern, you haven't yet put it to real use. Up to this point, it's been a sort of abstract methodology for writing programs. But now that you're ready to use view controllers, you can start using MVC as a real-world ideal for programming.

As you'll recall, under MVC, the *model* is your backend data and the *view* is your frontend user interface. The *controller* sits in between, accepting user input and modifying both of the other entities. The view controller should take the role of the controller in the MVC, as the name suggests. We'll get into this more in the next chapter, but we can say confidently that event and action control *will* happen through the view controller.

We can say this confidently because you'll pretty much be forced into using MVC. A view controller is automatically set up to access and modify various elements of views that sit under it. For example, the view controller has a `title` property that is intended to be a human-readable name for the page it runs. In chapter 7, you'll learn that tab bars and navigation bars automatically pick up that information for their own use. In addition, you'll often see view controllers automatically linked up to `delegate` and `datasource` properties, so that they can respond to the appropriate protocols for their subviews.

When you start seeing view controllers telling other objects what to do, look at it through the MVC lens. You should also think about MVC as you begin to program more complex projects using view controllers.

FINDING RELATED ITEMS

If a view controller is going to act as a controller, it needs easy access to the objects that lie both above and below it in the view hierarchy. For this purpose, the view controller contains a number of properties that can be used to find other items that are connected to it. They're listed in table 5.2.

Table 5.2 When you begin connecting a view controller to other things, you can use its properties to quickly access references to those other objects.

| Property | Summary |
|-----------------------------------|---|
| <code>modalViewController</code> | Reference to a temporary view controller, such as the Address Book and photo roll controllers that we'll discuss in chapter 8 and 11. |
| <code>navigationController</code> | Reference to a parent of the navigation controller type. |
| <code>parentViewController</code> | Reference to the immediate parent view controller, or nil if there is no view controller nesting. |
| <code>tabBarController</code> | Reference to a parent of the tab bar controller type. |
| <code>tabBarItem</code> | Reference to a tab bar item related to this particular view. |
| <code>view</code> | Reference to the controller's managed view. The view's <code>subviews</code> property may be used to dig further down in the hierarchy. |

These properties will be useful primarily when we move on to advanced view controllers, because they're more likely to link multiple view controllers together. We're mentioning them here because they're related to the idea of MVC and because they're `UIViewController` properties that will be inherited by all other types of controllers.

For now, we'll leave these MVC-related properties and get into some of the more practical things you can immediately do with a view controller, starting with managing view rotation.

ROTATING VIEWS

Telling your views to rotate is simple. In your view controller class file, you'll find a method called `shouldAutorotateToInterfaceOrientation:`. In order to make your application correctly rotate, all you need to do is set that function to return the Boolean `YES`, as shown here:

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

At this point, if you compile your program, you'll find that when you rotate your iPhone or iPad, the label shifts accordingly. Even better, because you set its font size to vary based on the amount of space it has, it gets larger when placed horizontally. This is a simple application of modifying your content based on the device's orientation.

You should consider one additional thing when rotating your views: whether they will resize to account for the different dimensions of the new screen.

RESIZING VIEWS

When you change your device's orientation from portrait to landscape, you change the amount of space for displaying content—for example, an iPhone goes from 320 x 480 to 480 x 320. As you just saw, when you rotated your label, it automatically resized, but this doesn't happen without some work.

A `UIView` (not the controller!) contains two properties that affect how resizing occurs. The `autoresizesSubviews` property is a Boolean that determines whether autoresizing occurs. By default, it's set to `YES`, which is why things worked correctly in the first view controller example. If you instead set it to `NO`, your view will stay the same size when a rotation occurs. In this case, your label will stay 320 pixels wide despite now being on a 480-pixel wide screen.

After you've set `autoresizesSubviews`, which says that resizing *will* occur, your view looks at its `autoresizingMask` property to decide *how* it should work. The `autoresizingMask` property is a bitmask that you can set with the different constants listed in table 5.3.

If you want to modify how your label resizes from within Xcode, you can do so by adding the following two lines to `viewDidLoad`:

```
myLabel.autoresizesSubviews = YES;
myLabel.autoresizingMask = UIViewAutoresizingFlexibleHeight |
    UIViewAutoresizingFlexibleWidth;
```

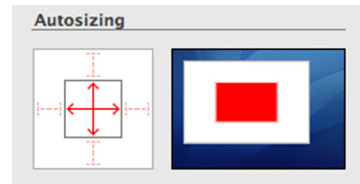
Table 5.3 `autoresizingMask` properties allow you to control how your views resize.

| Constant | Summary |
|---|-----------------------------------|
| <code>UIViewAutoresizingNone</code> | No resizing |
| <code>UIViewAutoresizingFlexibleHeight</code> | Height resizing allowed |
| <code>UIViewAutoresizingFlexibleWidth</code> | Width resizing allowed |
| <code>UIViewAutoresizingFlexibleLeftMargin</code> | Width resizing allowed to left |
| <code>UIViewAutoresizingFlexibleRightMargin</code> | Width resizing allowed to right |
| <code>UIViewAutoresizingFlexibleBottomMargin</code> | Height resizing allowed to bottom |
| <code>UIViewAutoresizingFlexibleTopMargin</code> | Height resizing allowed to top |

Note again that these resizing properties apply to a *view*, not to the view controller. You can apply them to any view you've seen so far. There has been little need for them before you started rotating things.

Modifying the way resizing works is even easier from within Interface Builder. If you recall, the `Resize` tab of the inspector window contains an `Autosizing` section, as shown in figure 5.3.

You can click six different arrows that correspond to the six resizing constants other than `None`. Highlighting an individual arrow turns on that type of resizing. The graphic to the right of these arrows serves as a nice guide to how resizing will work.

**Figure 5.3** IB graphically depicts exactly what autosizing looks like.

CHECKING ORIENTATION

Now that you have an application that can rotate at will, you may occasionally want to know what orientation a user's iPhone or iPad is sitting in. You do this by querying the `interfaceOrientation` view controller property. It's set to one of four constants, as shown in table 5.4.

You don't have to have a view controller to look up this information. A view controller's data is kept in tune with orientation values found in the `UIDevice` object—a useful object that also contains other device information, such as your system version. We'll talk about it in chapter 10.

Table 5.4 The view controller's `interfaceOrientation` property tells you the current orientation of an iPhone or iPad.

| Constant | Summary |
|---|------------------------------------|
| <code>UIInterfaceOrientationPortrait</code> | Device is vertical, right side up |
| <code>UIInterfaceOrientationPortraitUpsideDown</code> | Device is vertical, upside down |
| <code>UIInterfaceOrientationLandscapeLeft</code> | Device is horizontal, tilted left |
| <code>UIInterfaceOrientationLandscapeRight</code> | Device is horizontal, tilted right |

MONITORING THE LIFECYCLE

We've covered the major topics of loading, rotating, and resizing views within a view controller. With that under your belt, we can now look at the lifecycle events that may relate to these topics.

You saw lifecycle events in chapter 2, where we examined methods that alert you to the creation and destruction of the application, and some individual views. Given that one of the purposes of a controller is to manage events, it shouldn't be a surprise that the `UIViewController` has several lifecycle methods of its own, as shown in table 5.5.

Table 5.5 You can use the view controller's event-handler methods to monitor and manipulate the creation and destruction of its views.

| Method | Summary |
|---|--|
| <code>loadView:</code> | Creates the view controller's view if it isn't loaded from a <code>.xib</code> file. |
| <code>viewDidLoad:</code> | Alerts you that a view has finished loading. This is the place to put extra startup code if loading from a <code>.xib</code> file. |
| <code>viewWillAppear:</code> | Runs just before the view loads |
| <code>viewWillDisappear:</code> | Runs just before a view disappears—because it's dismissed or covered. |
| <code>willRotateToInterfaceOrientation:duration:</code> | Runs when rotation begins. |
| <code>didRotateToInterfaceOrientation:</code> | Runs when rotation ends. |

You've met `loadView` and `viewDidLoad`, which are run as part of the view controller's setup routine and which you used to add extra subviews. The `viewWillAppear:` message is sent afterward. The rest of the messages are sent at the appropriate times, as views disappear and rotation occurs.

Any of these methods can be overwritten to provide the specific functionality that you want when each message is sent.

OTHER VIEW METHODS AND PROPERTIES

The view controller object contains a number of additional methods that can be used to control exactly how rotation works, including controlling its animation and what header and footer bars slide in and out. These are beyond the scope of our introduction to view controllers, but you can find information about them in the `UIViewController` class reference.

That's our look at the bare view controller. You now know not only how to create your first view controller, but also how to use the fundamental methods and properties that you'll find in *every* view controller. But the other types of view controller also have special possibilities all their own. We'll look at these, starting with the one other view controller that's intended to control a single page of data: the table view controller.

5.3 The table view controller

Like the plain view controller, the table view controller manages a single page. Unlike the plain view controller, it does so in a structured manner. It automatically organizes the data in a nicely formatted table.

Our discussion of the table view controller will be similar to the discussion we just completed of the bare view controller. We'll examine its place in the view hierarchy, and then you'll learn how to create it, modify it, and use it at runtime.

Let's get started by examining the new view controller's anatomy.

5.3.1 The anatomy of a table view controller

The table view controller's setup is slightly more complex than that of the bare view controller. A `UITableViewController` controls a `UITableView`, which is an object that contains some number of `UITableViewCell` objects arranged in a single column. This is shown in figure 5.4.

By default, the controller is both the delegate and the data source of the `UITableView`. As we've previously discussed, these properties help a view hand off events and actions to its controller. The responsibilities for each of these control types is defined by a specific protocol: `UITableViewDelegate` declares which messages the table view controller must respond to, and `UITableViewDataSource` details how it must provide the table view with content. You can look up these protocols in the same library that you've been using for class references.

Of all the view controllers, the table view controller is the trickiest to create on its own, for reasons that you'll see momentarily.

5.3.2 Creating a table view controller

The easiest way to create an application that uses a table view controller is to use the Navigation-Based template in Xcode. This provides you with a delegate and a view that contains a table view controller. It also creates some of the delegate methods required for interfacing with the table view.

Although you can quickly start an application using the Navigation-Based template, we'll discuss in detail how you can manually build a table view controller project. This will give you a better understanding of what's going on when you use the template. Table 5.6 shows the process.

The project-creation, object-creation, and object-linking steps pretty much follow the lessons you've already learned. You have to create the subclass for the table view controller because the class file is where you define what the table view contains; we'll cover this in more depth shortly.

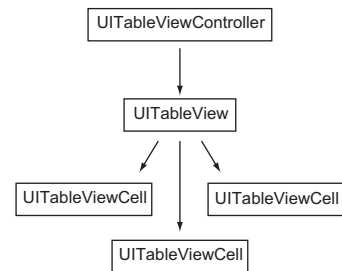


Figure 5.4 A table view controller controls a table view and its collection of cells.

Table 5.6 Creating a table view controller is simple, but it involves several steps.

| Step | Description |
|--|---|
| 1. Create a new project. | Open a Window-Based Application, and select iPhone from the Product drop-down menu. |
| 2. Create a table view controller. | In Xcode, create a new file containing a subclass of UIViewController. Then, select UITableViewController from the options. |
| 3. Link your Interface Builder object. | In Xcode, create an IBOutlet for your interface in the app delegate header file. In Interface Builder, link an outlet from your table view controller to the IBOutlet in the app delegate object, using the Connections tab of the inspector window. |
| 4. Connect your controller. | Link the controller's view to your main window. |

Note that you use two of the more advanced Interface Builder techniques that you learned in chapter 4: first linking in a new class (by changing the Identity tab) and then creating a new connection from it to your app delegate (via the Connections tab). As a result, you end up with two connections from Interface Builder to Xcode. On the one hand, the Interface Builder-created table view controller depends on your `RootViewController` files for its own methods; on the other hand, your app delegate file links to the controller (and eventually to the methods) via its outlet. This two-part connection to Interface Builder is common, and you should make sure you understand it before moving on.

As usual, you could elect to create this object solely in Xcode, by using an `alloc-init` command:

```
UITableViewController *myTable = [[RootViewController alloc]
    initWithStyle:UITableViewStylePlain];
```

The following simple code finishes the table-creation process by linking in the table's view in step 4 of the process:

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions{
    [window addSubview:myTable.view];
    [window makeKeyAndVisible];
}
```

Note that you link up your table view controller's view—not the controller itself—to your window. You've seen in the past that view controllers come with automatically created views. Here, the view is a table view.

If you want to see how that table view works, you can now go back into Interface Builder and click the table view to get its details. As shown in

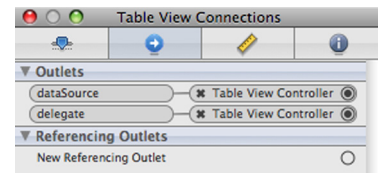


Figure 5.5 A look at the connections automatically created for a controller's table view

figure 5.5, it already has connections created for its `dataSource` and `delegate` properties.

Next, you need to fill the table with content.

5.3.3 Building up a table interface

As the data source, the controller needs to provide the view with its content. This is why you created a subclass for your table view controller and why every one of your table view controllers should have its own subclass: each will need to fill in its data in a different way.

We've mentioned that the `UITableViewDataSource` protocol declares the methods your table view controller should pay attention to in order to correctly act as the data source. The main work of filling in a table is done by the `tableView:cellForRowAtIndexPath:` method. When passed a row number, this method should return the `UITableViewCell` for that row of your table.

Before you can get to that method, though, you need to do some work. First, you must define the content that will fill your table. Then, you must define how large the table will be. Only then can you fill in the table using the `tableView:cellForRowAtIndexPath:` method.

In addition to these major table view elements, we'll also cover two optional variants that can change how a table looks: accessory views and sections.

CREATING THE CONTENT

You can use numerous SDK objects to create a list of data that your table should contain. In chapter 9, we'll talk about SQLite databases; and in chapter 14, we'll discuss pulling RSS data off the Internet. For now, we stay with the SDK's simpler objects. The most obvious are `NSArray`, which produces a static indexed array; `NSMutableArray`, which creates a dynamic indexed array; and `NSDictionary`, which defines an associative array.

For this example of table view content creation, you'll create an `NSArray` containing an `NSDictionary` that itself contains color names and `UIColor` values. As you can probably guess, you'll fill this skeletal table view example with something like the color selector that you wrote back when you were learning about views in chapter 4. The code required to create your content array is shown here:

```
- (void)viewDidLoad {
    colorList = [NSArray arrayWithObjects:
        [NSDictionary dictionaryWithObjectsAndKeys:
            @"brownColor",@"titleValue",
            [UIColor brownColor],@"colorValue",nil],
        [NSDictionary dictionaryWithObjectsAndKeys:
            @"orangeColor",@"titleValue",
            [UIColor orangeColor],@"colorValue",nil],
        [NSDictionary dictionaryWithObjectsAndKeys:
            @"purpleColor",@"titleValue",
            [UIColor purpleColor],@"colorValue",nil],
        [NSDictionary dictionaryWithObjectsAndKeys:
            @"redColor",@"titleValue",
            [UIColor redColor],@"colorValue",nil],
```

```

        nil];
    [colorList retain];
}

```

You should do this sort of setup before the view appears. Here, you do it in the `viewDidLoad` method. This method is called prior to the view appearing and is a good place to do your initialization.

The array and dictionary creations are simple. The Apple class references contain complete information about how to create and manipulate these objects; but, in short, you can create an `NSArray` as a listing of objects ending in a `nil`, and you can create an `NSDictionary` using pairs of values and keys, ending in a `nil`. Here, you're creating an array containing four dictionaries, each of which will fill one line of your table.

You also have to think about memory management here. Because your array was created with a class factory method, it'll be released when it goes out of scope. In order to use this array elsewhere in your class, you not only need to have defined it in your header file, but also need to send it a `retain` message to keep it around. You'll release it in your `dealloc` method, elsewhere in the class files.

BUILDING YOUR TABLE CELLS

When you've set up a data backend for your table, you need to edit three methods in your table view controller file: two that define the table and one that fills it, as shown in listing 5.1. We'll explain each of these in turn.

Listing 5.1 Three methods that control how your table is created and runs

```

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    return 1;
}

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


- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
            initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:CellIdentifier]
            autorelease];
    }

    cell.textLabel.textColor= [[colorList objectAtIndex:indexPath.row]
        objectForKey:@"colorValue"];
    cell.textLabel.text = [[colorList objectAtIndex:indexPath.row]
        objectForKey:@"titleValue"];

    return cell;
}

```



1
Sets cell's text
and text color

All these methods should appear by default in the table view controller subclass you create, but you may need to make changes to some of them to accommodate the specifics of your table.

The first method is `numberOfSectionsInTableView:`. Tables can optionally include multiple sections, each of which has its own index of rows, and each of which can have a header and a footer. For this example, you're creating a table with one section, but we'll look at multiple sections before we finish this chapter.

The second method, `tableView:numberOfRowsInSection:`, reports the number of rows in this section. Here, you return the size of the array you created. Note that you ignore the `section` variable because you have only one section.

The third method, `tableView:cellForRowAtIndexPath:`, takes the table set up by the previous two methods and fills its cells one at a time. Although this chunk of code looks intimidating, most of it will be sitting there waiting for you the first time you work with a table. In particular, the creation of `UITableViewCell` will be built in. All you need to do is set the values of the cell before it's returned. Here you use your `NSDictionary` to set the cell's text color and text content ❶.

Also note that this is your first use of the `NSIndexPath` data class. It encapsulates information on rows and sections. Cells have two views that you can access. The first is the `textLabel`. As you saw, this contains the text displayed in the cell. The other is `imageView`. It's basically an icon for the cell. You can set this to an image view. See section 11.2 for more information about using `UIImage`.

You may want to change more than text content and color. Table 5.7 lists all the cell label features that you may want to experiment with at this point.

Using these properties, you can make each table cell look unique, depending on the needs of your program.


ADDING ACCESSORY VIEWS

Although you didn't do so in the color-selector example, you can optionally set accessories on cells. *Accessories* are special elements that appear to the right of each list item.

Table 5.7 You can modify your table cells in a variety of ways.

| Property | Summary |
|--|---|
| <code>textLabel.font</code> | Sets the cell label's font using <code>UIFont</code> |
| <code>textLabel.lineBreakMode</code> | Sets how the cell label's text wraps using <code>UILineBreakMode</code> |
| <code>textLabel.text</code> | Sets the content of a cell label to an <code>NSString</code> |
| <code>textLabel.textAlignment</code> | Sets the alignment of a cell's label text using the <code>UITextAlignment</code> constant |
| <code>textLabel.textColor</code> | Sets the color of the cell's label text using <code>UIColor</code> |
| <code>textLabel.selectedTextColor</code> | Sets the color of selected text using <code>UIColor</code> |
| <code>imageView.image</code> | Sets the content of a cell's <code>imageView</code> to a <code>UIImage</code> |
| <code>imageView.selectedImage</code> | Sets the content of a selected cell to <code>UIImage</code> |

Table 5.8 A cell accessory gives additional information.

| Constant | Summary |
|---|---|
| <code>UITableViewCellAccessoryNone</code> | No accessory |
| <code>UITableViewCellAccessoryDisclosureIndicator</code> | A normal chevron: > |
| <code>UITableViewCellAccessoryDetailDisclosureButton</code> | A chevron in a blue button:  |
| <code>UITableViewCellAccessoryCheckmark</code> | A checkmark: ✓ |

Most frequently, you'll set accessories using an `accessoryType` constant that has four possible values, as shown in table 5.8.

An accessory can be set as a property of a cell:

```
cell.accessoryType = UITableViewCellAccessoryDetailDisclosureButton;
```

The normal chevron is usually used with a navigation controller, the blue chevron is typically used for configuration, and the checkmark indicates selection.

There is also an `accessoryView` property, which lets you undertake the more complex task of creating an entirely new view to the right of each list item. You create a view and then set `accessoryView` to that view:

```
cell.accessoryView = [[myView alloc] init];
```

There's an example of this in chapter 8, where you'll be working with preference tables.

ADDING SECTIONS

The example shows how to display a single section's worth of cells, but it would be trivial to rewrite the functions to offer different outputs for different sections within the table. Because of Objective-C's ease of accessing nested objects, you can prepare for this by nesting an array for each section inside a larger array:

```
masterColorList = [NSArray arrayWithObjects:colorList,otherColorList,nil];
```

Then, you return the count from this über-array for the `numberOfSections:` method:

```
return masterColorList.count;
```

You similarly return a subcount of one of the subarrays for the `tableView:numberOfRows:` method:

```
return [[masterColorList objectAtIndex:section] count];
```

Finally, you pull content from the appropriate subarray when filling in your cells using the same type of nested messaging.

When you're working with sections, you can also think about creating headers and footers for each section. Figure 5.6 shows what the revised application looks like so far, including two different sections, each of which has its own section header.

How do you create those section headers? As with all the methods you've seen that fill in table views, the section header messages and properties show up in the `UITableViewDataSource` protocol reference.

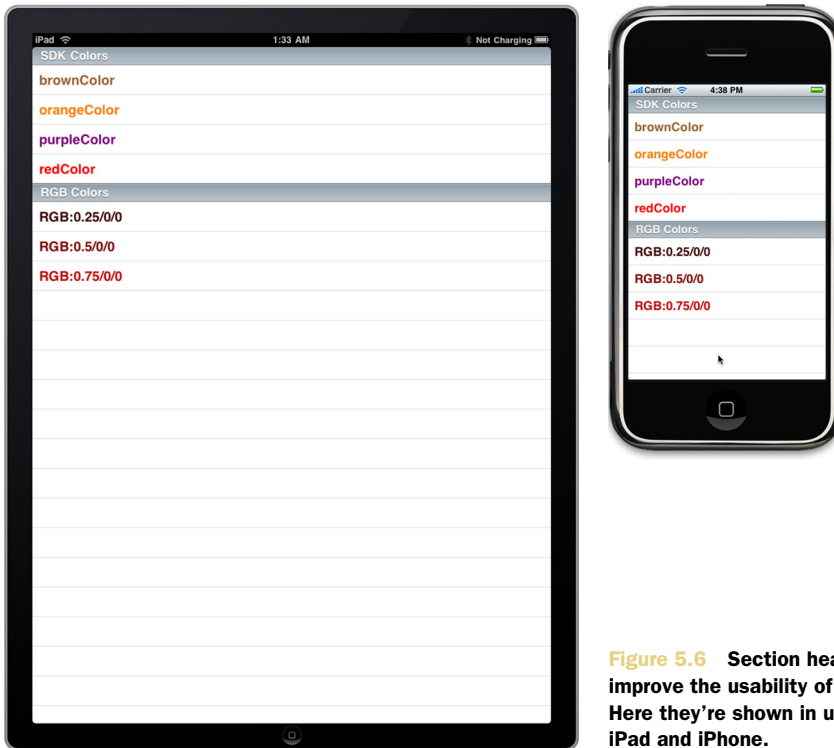


Figure 5.6 Section headers can improve the usability of table views. Here they're shown in use on both the iPad and iPhone.

To create section headers, you write a `tableView:titleForHeaderInSection:` method. As you'd expect, it renders a header for each individual section.

An example of its use is shown here. You could probably do something fancier instead, such as building the section names directly into your array:

```
- (NSString *)tableView:(UITableView *)tableView
    titleForHeaderInSection:(NSInteger)section {
    if (section == 0) {
        return @"SDK Colors";
    } else if (section == 1) {
        return @"RGB Colors";
    }
    return 0;
}
```

You can similarly set footers and otherwise manipulate sections according to the protocol reference.

There's still more to the table view controller. Not only do you have to work with data when you're setting it up, but you also have to do so when it's in active use, which usually occurs when the user selects individual cells.

5.3.4 Using your table view controller

We won't dwell too much on the more dynamic possibilities of the `UITableViewController` here. For the most part, you'll either use it to hold relatively static data (as you do here) or use it to interact with a navigation controller (as you'll see in chapter 7). But before we finish up with table view controllers, we'll look at one other fundamental: selection.

SELECTED CELLS

If you try the sample `tableex` application that you've been building throughout section 5.3, you'll see that individual elements in a table view can be selected.

In table 5.7, you saw that some properties apply explicitly to selected cells. For example, the following maintains the color of your text when it's selected, rather than changing it to white, as per the default:

```
cell.textLabel.textColor =
    [[[masterColorList objectAtIndex:indexPath.section]
     objectAtIndex:indexPath.row] objectForKey:@"colorValue"];
```

To set this value, you must add this line of code to your `tableView:didSelectRowAtIndexPath:` method. Also note that this is another example of using nested arrays to provide section- and row-specific information for a table list.

The `tableView:didSelectRowAtIndexPath:` method is the most important for dealing with selections. This method appears in the `UITableViewDelegate` protocol and tells you when a row has been selected. The message includes an index path, which, as you've already seen, contains both a row and a section number.

Here's a simple example of how you might use this method to checkmark items in your list:

```
- (void)tableView:(UITableView *)tableView
  didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    [[tableView cellForRowAtIndexPath:indexPath]
     setAccessoryType:UITableViewCellAccessoryCheckmark];
}
```

You can easily retrieve the selected cell by using the index path, and then you use that information to set the accessory value. You'll make more use of cell selection in chapter 7, when we talk about navigation controllers.

5.4 Summary

View controllers are the most important building blocks of the SDK that you hadn't seen up to this point. As we explained in this chapter, they sit atop views of all sorts and control how those views work. Even in this chapter's simple examples, you saw some real-world examples of this control, as view controllers managed rotation, filled tables, and reacted to selections.

You can think of a view controller as being like the glue of your application. It connects your view components to the underlying models. View controllers provide interaction with the interface through `IBOutlet`s and `IBAction`s.

Now that we're getting into user interaction, we're ready to examine how it works in more depth, and that's the focus of the next chapter. We'll examine the underpinnings of user interaction: events and actions.

iPhone and iPad IN ACTION

Trebitowski • Allen • Appelcline



This hands-on tutorial will help you master iPhone/iPad development using the native iPhone SDK. It guides you from setting up dev tools like Xcode and Interface Builder, through coding your first app, all the way to selling in the App Store.

Using many examples, the book covers core features like accelerometers, GPS, the Address Book, and much more. Along the way, you'll learn to leverage your iPhone skills to build attractive iPad apps. This is a revised and expanded edition of the original *iPhone in Action*.

What's Inside

- Full coverage of the iPhone SDK
- In-app purchasing with Store Kit
- Audio and recording
- Core Data, Core Location, Game Kit, Map Kit
- And much more!

No previous iPhone or iPad know-how needed. Familiarity with C, Cocoa, or Objective-C helps but is not required.

Brandon Trebitowski is a professional mobile developer with ELC Technologies and founder of iCodeBlog.com.

Christopher Allen is the host of iphonewebdev.com and an organizer of iPhoneDevCamp. **Shannon Appelcline** is a writer, technologist, and game developer. Christopher and Shannon wrote *iPhone in Action*.

For online access to the authors and a free ebook for owners of this book, go to manning.com/iPhoneandiPadinAction

“Everything you need to know about these devices of the future!”

—Berndt Hamboeck, pmOne

“Apple should make this its official iPhone and iPad development book.”

—Jason Jung, Rockwell

“Gets you up to speed and developing in a snap.”

—Clint Tredway, Developed It

“Don't launch Xcode without it.”

—Ted Neward
Neward & Associates

“Exactly what iNeed for iPhone development.”

—Christopher Haupt
Webvanta.com