John Ray

In **Full Color**

Figures and code appear as they do in Xcode

Covers iOS 3.2 and up

Sams **Teach Yourself**

iPad™ Application Development

in **24 Hours**

SAMS

John Ray

Sams Teach Yourself

iPad™ Application Development

in 24 Hours

SAMS 800 East 96th Street, Indianapolis, Indiana, 46240 USA

Sams Teach Yourself iPad™ Application Development in 24 Hours

ISBN-13: 978-0-672-33339-2
ISBN-10: 0-672-33339-2

Library of Congress Cataloging-in-Publication Data:

Ray, John, 1971-

Sams teach yourself iPad application development in 24 hours / John Ray.

p. cm.

Includes index.

ISBN 978-0-672-33339-2

1. iPad (Computer)—Programming. 2. Application software—Development. I. Title.

QA76.8.I863R392 2011

005.3—dc22

2010023693

Printed in the United States of America

First Printing July 2010

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales
international@pearson.com

Associate Publisher
Greg Wiegand

Acquisitions Editor
Laura Norman

Development Editor
Keith Cline

Managing Editor
Kristy Hart

Project Editor
Lori Lyons

Indexer
Angela Martin

Proofreader
Kathy Ruiz

Technical Editor
Matthew David

Publishing Coordinator
Cindy Teeters

Multimedia Developer
Dan Scherf

Book Designer
Gary Adair

Compositor
Gloria Schurick

Contents at a Glance

Table of Contents

About the Author

John Ray is currently serving as a Senior Business Analyst and Development Team Manager for the Ohio State University Research Foundation. He has written numerous books for Macmillan/Sams/Que, including *Using TCP/IP: Special Edition*, *Sams Teach Yourself Dreamweaver MX in 21 Days*, *Mac OS X Unleashed*, and *Sams Teach Yourself iPhone Development in 24 Hours*. As a Macintosh user since 1984, he strives to ensure that each project presents the Macintosh with the equality and depth it deserves. Even technical titles such as *Using TCP/IP* contain extensive information on the Macintosh and its applications—and have garnered numerous positive reviews for its straightforward approach and accessibility to beginning and intermediate users.

Dedication

This book is dedicated to everyone who can see beyond the count of USB ports, RAM slots, and technical jargon to recognize the beauty of a platform as a whole. I'm excited to see what you create.

Acknowledgments

Thank you to the group at Sams Publishing—Laura Norman, Kristy Hart, Lori Lyons, Keith Cline, Matthew David, Gloria Schurick, Kathy Ruiz—for recognizing the importance of the iPhone OS/iPad platform, and helping to create this book. Skilled editors make authors coherent.

As always, thanks to my family and friends for keeping me sane for the duration of the project. It wasn't that bad, was it?

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

E-mail: feedback@samspublishing.com

Mail: Greg Wiegand
Associate Publisher
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Introduction

"It's just a big iPod Touch."

Few words have puzzled me more during the weeks leading up to the iPad launch. Let's break down exactly what it means to be a "big iPod Touch."

First, it means a large, bright, colorful display, coupled with an amazingly thin enclosure and amazing battery life. Second, it means a user experience based on the world's most popular portable Internet device.

Perhaps the most important aspect of being a "big iPod touch" is that it is a device designed to be controlled by human fingers. Every aspect of development is centered on touch interactions. Quite simply, the iPad is a multitouch device that is 100% dedicated to running applications that you control with your fingers.

Terrible, isn't it?

Less than a month after the iPad launched, Apple has sold more than a million units. It's reassuring that people still recognize and embrace innovation. It also means that there is no end to the opportunity that the iPad affords to you, the developer.

The iPad is an open canvas. On the iPhone, there are plenty of apps, but less of an opportunity to experiment with user interfaces. On the iPad, apps take on new life. The display begs to be touched, and complex gestures are fun and easy to implement. Computing truly becomes a personal experience, similar to curling up with a good book.

Our hope is that this book will bring iPad development to a new generation of developers who want to create large-scale multitouch applications. *Sams Teach Yourself iPad Application Development in 24 Hours* provides a clear natural progression of skills development—from installing developer tools and registering with Apple, to submitting an application to the App Store. It's everything you need to get started in just 24 hour-long lessons.

Who Can Become an iPad Developer?

If you have an interest in learning, time to invest in exploring and practicing with Apple's developer tools, and an Intel Macintosh computer, you have everything you need to begin developing for the iPad.

Developing an application won't happen overnight, but with dedication and practice, you can be writing your first applications in a matter of days. The more time you spend working with the Apple developer tools, the more opportunities you'll discover for creating new and exciting projects.

You should approach iPad application development as creating software that *you* want to use, not what you think others want. If you're solely interested in getting rich quick, you're likely to be disappointed. (The App Store is a crowded marketplace—albeit one with a lot of room—and competition for top sales is fierce.) However, if you focus on building useful and unique apps, you're much more likely to find an appreciative audience.

Who Should Use This Book?

This book targets individuals who are new to development for the iPhone OS and have experience using the Macintosh platform. No previous experience with Objective-C, Cocoa, or the Apple developer tools is required. Of course, if you do have development experience, some of the tools and techniques may be easier to master, but this book does not assume that you've coded before.

That said, some things are expected from you, the reader. Specifically, you must be willing to invest in the learning process. If you just read each hour's lesson without working through the tutorials, you will likely miss some fundamental concepts. In addition, you need to spend time reading the Apple developer documentation and researching the topics covered in this book. A vast amount of information on iPhone development is available, but only limited space is available in this book. However, this book does cover what you need to forge your own path forward.

What Is (and Isn't) in This Book?

This book targets the initial release of the iPad OS 3.2. Much of what you'll be learning is common to all iPhone OS releases (the iPad is built on top of the iPhone OS), but we also cover several important advances, such as popovers, modal views, and more!

Unfortunately, this is not a complete reference for the iPhone OS application programming interfaces. Some topics require much more space than the format of this book allows. Thankfully, the Apple developer documentation is available directly within the free tools you'll be downloading in Hour 1, "Preparing Your System and iPad for Development." Many lessons include a section titled "Further Exploration"

that will guide you toward additional related topics of interest. Again, a willingness to explore is an important quality in becoming a successful developer!

Each coding lesson is accompanied by project files that include everything you need to compile and test an example or, preferably, follow along and build the application yourself. Be sure to download the project files from the book's website at http://teachyourselfipad.com.

In addition to the support website, you can follow along on Twitter! Search for #iPadIn24 on Twitter to receive official updates and tweets from other readers. Use the hashtag #iPadIn24 in your tweets to join the conversation. To send me messages via Twitter, begin each tweet with @johnemeryray.

This page intentionally left blank

HOUR 11

Presenting Options with Popovers and Toolbars

What You'll Learn in This Hour:

- How to add toolbars and toolbar buttons to your projects
- The role of popovers in the iPhone OS
- How to generate custom popover views in your projects
- Tricks for checking to see whether a popover is already displayed

On the iPhone, what you see is typically what you get. The user interface elements either show the options that are available to you, offer the ability to scroll to additional options, or swap out the current screen for another view that displays more information. The multiple-window model used in Mac OS X is gone. Although you might encounter an occasional alert dialog, windowing is not a standard in iPhone interfaces. On the iPad, things have changed. Apple has introduced the popover: a user interface element that can present views on top of other views.

In this hour, we explore how to prepare views for use in popovers, including adding toolbars and toolbar buttons (the most frequent UI element used to invoke a popover). You'll also configure the different display attributes associated with popovers, and communicate information between popover views and your main application view. Popovers are such a prevalent and important UI element that we'll be focusing on them for the next few hours, so be sure to work through this lesson carefully.

Understanding Popovers and Toolbars

Popovers are everywhere in the iPad interface, from Mail to Safari, as demonstrated in Figure 11.1. Using a popover enables you to display new information to your users without leaving the screen you are on, and to hide the information when the user is done with it. There are few desktop counterparts to popovers, but they are roughly analogous to tool palettes, inspector panels, and configuration dialogs. In other words, they provide user interfaces for interacting with content on the iPad screen, but without eating up permanent space in your UI.

FIGURE 11.1
Popovers are unique to the iPad UI.

Popovers, although capable of being displayed when a user interacts with any onscreen object, are most often shown when the user presses a toolbar button (`UIBarButton`) from within a toolbar object (`UIToolbar`). This is exactly the scenario shown in Figure 11.1. Because of this relationship, we will be presenting both of these objects within this hour's lesson. Let's quickly review what we need for each before we get started coding.

Popovers

Unlike other UI elements, popovers aren't something you just drag into a view from the Interface Builder Library. They are, in fact, entirely independent views, designed just like your main application view. The display of the views is governed by a popover controller (`UIPopoverController`). The controller displays the popover

when a user event is triggered, such as touching a toolbar button. When the user is done with the popover, touching outside of its visible rectangle automatically closes the view.

To create a popover, we'll need to cover three different requirements. First, we need to make a view and view controller specially designed for the popover's contents. Second, when the proper event occurs in the user interface, we need to allocate and initialize an instance of popover controller. Third, when the user is done with the popover, we want to make sure that any changes made in the popover are reflected in the main application.

Popover Views

You've been developing views and view controllers for the past several hours, so you'll feel right at home working with a popover view. It uses the same `UIViewController` that we've been using all along, but with the addition of one unique property: `contentSizeForViewInPopover`.

This property should be set to the width and height of the popover to be displayed. Apple allows popovers up to 600 pixels wide and the height of the iPad screen, but recommends that they be kept to 320 pixels wide, or less. For example, to set the content size of 320 pixels by 200 pixels for a view controller that will be displaying a popover, we might add the following to the `viewDidLoad` method:

```
self.contentSizeForViewInPopover=CGSizeMake(320.0,200.0);
```

In fact, that's exactly what we're going to be doing in the tutorial shortly.

Popover Controller

Like views need view controllers, popovers need popover controllers (`UIPopoverController`). Popover controllers take care of all the hard work of rendering popovers on the screen in the right place. We'll focus on two methods of the popover controller:

> `initWithContentViewController`—Initializes the popover with the contents of a view controller. When the popover is displayed, whatever the view controller's view is, is displayed.
>
> `presentPopoverFromBarButtonItem:permittedArrowDirections:animated`—Invokes the display of the popover so that it appears to emerge from (and point to) a toolbar button. The parameters for this method allow fine-tuning of the arrow from the popover to the UI element it is appearing from, and whether its display is animated.

The popover controller will also need the `delegate` property set to an object that will take care of all the "cleanup" when the popover is dismissed by the user. This

includes releasing the popover controller and updating the contents of the main application to reflect the user's actions in the popover. This leads us to the final popover requirement: the `UIPopoverControllerDelegate` protocol.

Popover Controller Delegate Protocol

To make the full use of a popover, we'll need an additional protocol method added to one of our classes. In our sample application, we'll be using our main application's view controller class for this purpose. This means we need to add a line to our main view controller's interface (.h) file to state that we're conforming to the `UIPopverControllerDelegate` protocol. Second, we'll be adding the protocol method `popoverControllerDidDismissPopover` to our application's view controller implementation file. That's it.

When the popover is dismissed by the user touching outside of its display, the `popoverControllerDidDismissPopover` method is invoked and we can react appropriately.

Toolbars

Toolbars (`UIToolbar`) are, comparatively speaking, one of the simpler UI elements that you have at your disposal. A toolbar is implemented as a solid bar, either at the top or bottom of the display, with buttons (`UIBarButtonItem`) that correspond to actions that can be performed in the current view. The buttons provide a single selector action, which works nearly identically to the typical Touch Up Inside event that you've encountered before.

Toolbars, as their name implies, are used for providing a set of static choices to the user—interface options that should be visible regardless of whether the application's primary content is changing. As you'll see, they can be implemented almost entirely visually and are the de facto standard for triggering the display of a popover on the iPad.

Although implementing popovers might be sounding a bit convoluted at this point, hang in there. After you've created one, the process will seem incredibly simple, and you'll want to use them everywhere!

Using Popovers with Toolbars

Popovers are used to display interface elements that configure how your application behaves but that don't need to be visible all the time. Our sample implementation will display a toolbar, complete with a Configure button, that invokes a popover. The popover will display configuration four switches (`UISwitch`) for a hypothetical time-based application: Weekends, Weekdays, AM, and PM.

The user will be able to update these switches in the popover, and then touch outside the popover to dismiss it. Upon dismissal, four labels in the main application view will be update to show the user's selections. The final application will resemble Figure 11.2.

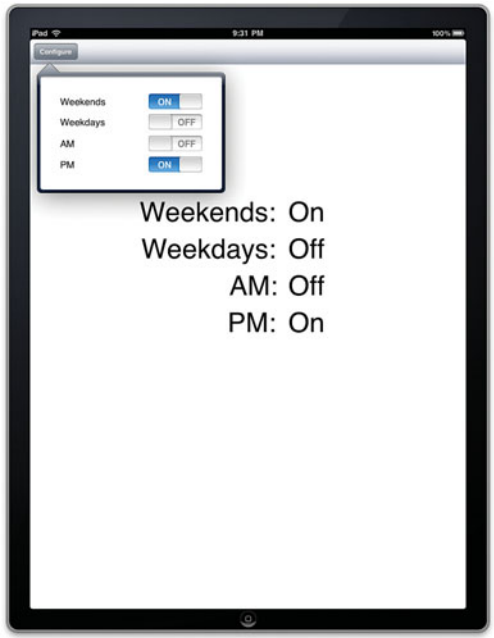

FIGURE 11.2
This application will display a popover and update the main application view to reflect a user's actions in the popover.

Implementation Overview

The implementation of this project is simpler than it may seem at the onset. You'll be creating a View-based iPad application that includes a toolbar with the Configure button and four labels that will display what a user has chosen in the popover. The popover will require its own view controller and view. We'll add these to the main project, but they'll be set up almost entirely independently from the main application view.

Building the connection between the main view and the popover will require surprisingly few lines of code. We need to be careful that touching the Configure button doesn't continue to add popovers to the display if one is already shown, but you'll learn a trick that keeps it all under control.

Setting Up the Project

This project will start with the View-Based Application template; we'll be adding in another view and view controller to handle the popover. Let's begin. Launch Xcode

(Developer/Applications), and then create a new View-based iPad project called **PopoverConfig**.

Xcode will create the basics structure for your project, including the `PopoverConfigViewController` classes. We'll refer to this as the *main application view controller* (the class the implements the view that the user sees when the application runs). For the popover content itself, we need to add a new view controller and XIB file to the PopoverConfig project.

Adding an Additional View Controller Class

With the Classes group selected in your Xcode project, choose File, New File, from the menu bar. Within the New File dialog box, choose the Cocoa Touch Class within the iPhone OS category, and then the `UIViewController` subclass icon, as shown in Figure 11.3.

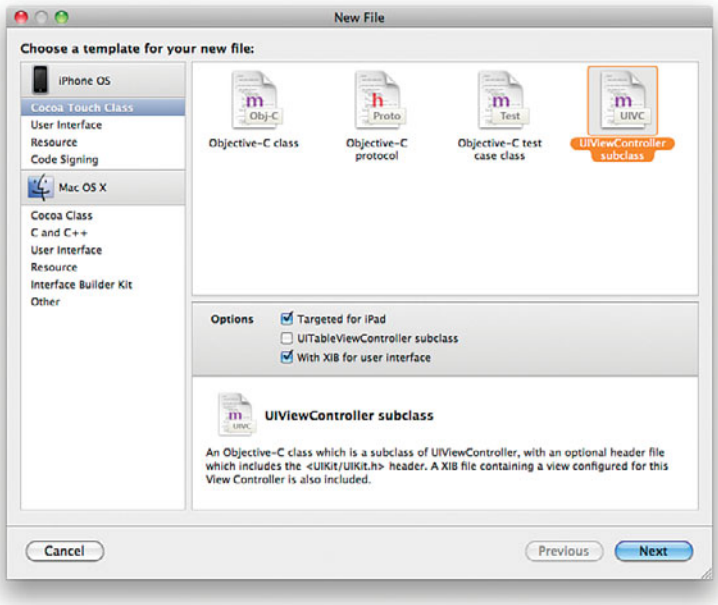

FIGURE 11.3 Create the popover's content view controller and XIB file.

Be sure that Targeted for iPad and With XIB for user interface are selected, and then choose Next. When prompted, name the new class **PopoverContentViewController** and click Finish.

The `PopoverContentViewController` implementation and interface files are added to the Classes group.

Depending on your version of Xcode, the XIB file may also be added to the folder you had selected when creating the class files. If this is the case, drag it to the Resources group.

Did you Know?

Preparing the Popover Content

This hour's project is unique in that most of your interface work takes place in a view that is only onscreen occasionally when the application is running—the popover's content view. The view will have four switches (`UISwitch`), which we'll need to account for.

We only need to be able to read values from the popup view, not invoke any actions, so we'll just add four `IBOutlets`.

Adding Outlets

Open the PopoverContentViewController.h interface file and add outlets for four `UISwitch` elements: `weekendSwitch`, `weekdaySwitch`, `amSwitch`, `pmSwitch`. Be sure to also at `@property` directives for each switch. The resulting interface file is shown in Listing 11.1.

LISTING 11.1

```
#import <UIKit/UIKit.h>

@interface PopoverContentViewController : UIViewController {
    IBOutlet UISwitch *weekendSwitch;
    IBOutlet UISwitch *weekdaySwitch;
    IBOutlet UISwitch *amSwitch;
    IBOutlet UISwitch *pmSwitch;
}

@property (nonatomic,retain) UISwitch *weekendSwitch;
@property (nonatomic,retain) UISwitch *weekdaySwitch;
@property (nonatomic,retain) UISwitch *amSwitch;
@property (nonatomic,retain) UISwitch *pmSwitch;

@end
```

For each of the properties we've declared, we need to add a `@synthesize` directive in the implementation (popoverContentViewController.m) file. Open this file and make your additions following the `@implementation` line:

```
@synthesize weekdaySwitch;
@synthesize weekendSwitch;
@synthesize amSwitch;
@synthesize pmSwitch;
```

Setting the Popover Content Size

Our next step is easy to overlook, but amazingly important to the final application. For an application to present an appropriately sized popover, you must manually define the popover's content size. The easiest (and most logical) place to do this is within the popover's view controller.

Continue editing the popoverContentViewController.m file to uncomment its `viewDidLoad` method and add a size definition:

```
- (void)viewDidLoad {
    self.contentSizeForViewInPopover=CGSizeMake(320.0,200.0);
    [super viewDidLoad];
}
```

For this tutorial project, our popover will be 320 pixels wide and 200 pixels tall. Remember that Apple supports values up to 600 pixels wide and a height as tall as the iPad's screen area allows.

Releasing Objects

Even though this view controller sits outside of our main application, we still need to clean up memory properly. Finish up the implementation of the `popoverContentViewController` class by releasing the four switch instance variables in the `dealloc` method:

```
- (void)dealloc {
    [weekdaySwitch release];
    [weekendSwitch release];
    [amSwitch release];
    [pmSwitch release];
    [super dealloc];
}
```

That finishes the `popoverContentViewController` logic! Although we still have a little bit of work to do in Interface Builder, the rest of the programming efforts will take place in the main popoverConfig view controller class.

Preparing the View

Building a popover's view is *identical* to building any other view with one small difference: You can only use the portion of the view that fits within the size of the popover you're creating. Open the popoverContentViewController XIB file in Interface Builder and add four labels (Weekends, Weekdays, AM, and PM) and four corresponding switches (`UISwitch`) from the library.

Position these in the upper-left corner of the view to fit within the 320x200 dimensions we've defined, as shown in Figure 11.4.

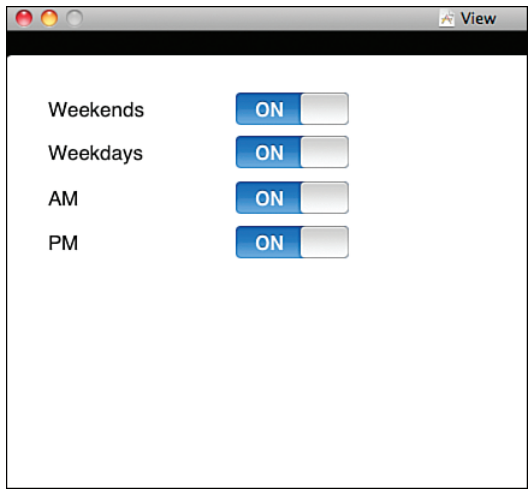

FIGURE 11.4
Add four configuration switches and corresponding labels to the popover content view.

Connecting the Outlets

After creating the view, connect the switches to the `IBOutlets`. Control-drag from the File's Owner icon in the Document window to the first switch in the view (the Weekends switch in my implementation) and choose the `weekendSwitch` outlet when prompted, as shown in Figure 11.5.

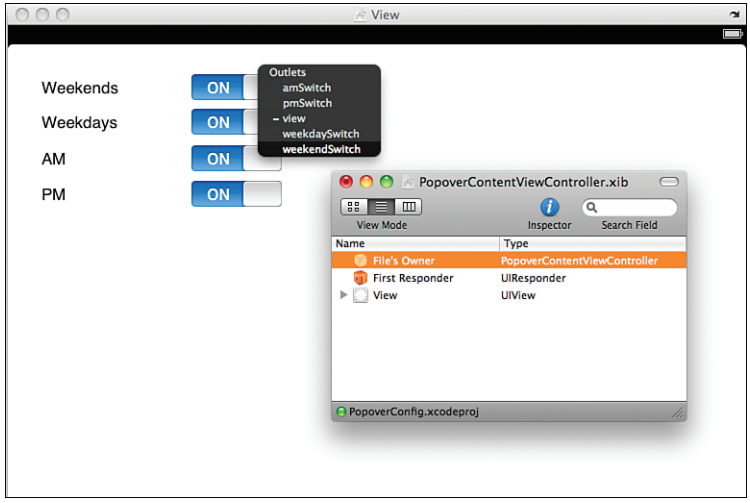

FIGURE 11.5
Connect the switches to their outlets.

Repeat these steps for the other three switches, connecting them to the `weekdaySwitch`, `amSwitch`, and `pmSwitch` outlets.

The popover content is now complete. Let's move to the main application.

Preparing the Application View

With the popover content under control, we'll build out the main application view/view controller. There are only a few "gotchas" here, such as declaring that we're going to conform to the `UIPopoverControllerDelegate` protocol, and making sure that we create an instance of the popover content view.

Conforming to a Protocol

To conform to the popover controller delegate, open the popoverConfigViewController.h interface file, and modify the `@interface` line to include the name of the protocol, enclosed in <>. The line should read as follows:

```
@interface PopoverConfigViewController : UIViewController
<UIPopoverControllerDelegate> {
```

We still need to implement a method for the protocol within the view controller, but more on that a bit later.

Adding Outlets, Actions, and Instance Variables

We need to keep track of quite a few things within the main application's view controller. We're going to need an instance variable for the popover's controller (`popoverController`). This will be used to display the popover, and to check whether the popover is already onscreen. We'll also need an `IBAction` defined (`showPopover`) for displaying the popover.

In addition, five `IBOutlets` are required—four for `UILabels` that will display the values the user enters in the popover (`weekdayOutput`, `weekendOutput`, `amOutput`, `pmOutput`), and the last for the popover's view controller (`popoverContent`).

Sound like enough? Not quite! Because we're going to be using the `popoverContentViewController` class within the main application, we need to import its interface file, too.

Did you Know?

What About the Configure Button?

If you're following closely, you might wonder whether we need an instance variable for the Configure button. When we initialize the popover controller, we need to tell it what onscreen object it should point to (that is, the Configure button). Thankfully, the button passes a reference of itself to the action it calls when pressed, so we can use that reference rather than keeping track of the button separately.

Edit the interface file so that it matches Listing 11.2.

LISTING 11.2

```
#import <UIKit/UIKit.h>
#import "PopoverContentViewController.h"

@interface PopoverConfigViewController : UIViewController
           <UIPopoverControllerDelegate> {
    UIPopoverController *popoverController;
    IBOutlet    UILabel *weekdayOutput;
    IBOutlet    UILabel *weekendOutput;
    IBOutlet    UILabel *amOutput;
    IBOutlet    UILabel *pmOutput;
    IBOutlet    popoverContentViewController *popoverContent;
}

@property (retain,nonatomic) UILabel *weekdayOutput;
@property (retain,nonatomic) UILabel *weekendOutput;
@property (retain,nonatomic) UILabel *amOutput;
@property (retain,nonatomic) UILabel *pmOutput;
@property (retain,nonatomic) PopoverContentViewController *popoverContent;

-(IBAction)showPopover:(id)sender;

@end
```

For each `@property` directive, there needs to be a corresponding `@synthesize` in the popoverConfigViewController.m file. Edit the file now, adding these lines following the `@implementation` line:

```
@synthesize popoverContent;
@synthesize weekdayOutput;
@synthesize weekendOutput;
@synthesize amOutput;
@synthesize pmOutput;
```

This gives us everything we need to build and connect the main application interface elements, but before we do, let's make sure that everything we're added here is properly released.

Releasing Objects

Edit popoverConfigViewController.m's `dealloc` method to release the `UILabels`, and the instance of the popover content view controller (`popoverContent`):

```
- (void)dealloc {
    [weekdayOutput release];
    [weekendOutput release];
    [amOutput release];
    [pmOutput release];
    [popoverContent release];
    [super dealloc];
}
```

Nicely done! All that's left now is to edit the popoverConfigViewController XIB file to create the main application interface and write the methods for showing and handling the subsequent dismissal of the popover.

Creating the View

Open the popoverConfigViewController XIB file in Interface Builder. We need to add a toolbar, a toolbar button, and some labels to display our application's output. Let's start with the labels, because we've got plenty of experience with them. Drag a total of eight `UILabel` objects to the screen. Four will hold the application's output, and four will just be labels (fancy that!).

Arrange the labels near the center of the screen, forming a column with Weekends:, Weekdays:, AM:, and PM: on the left, and On, On, On, On aligned with them on the right. The On labels are the labels that will map to the `IBOutlet` output variables; they've been set to a default value of `On` because the switches in the popover content view default to the On position.

If desired, use the Attributes Inspector (Command+1) to resize the labels to something a bit larger than the default. I've used a 48pt font in my interface, as shown in Figure 11.6.

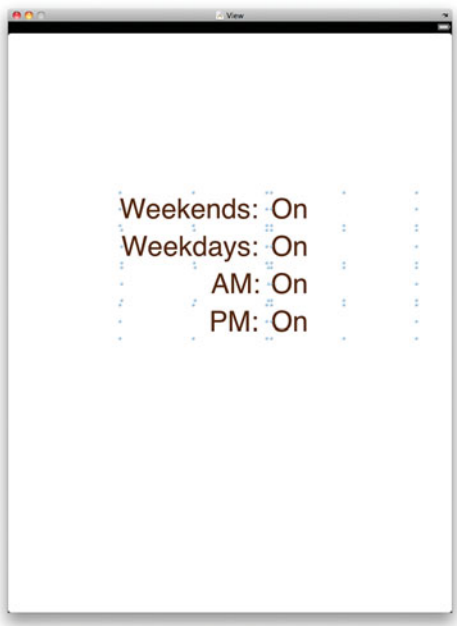

FIGURE 11.6
Add a total of eight labels to the view.

Adding a Toolbar and Toolbar Button

Using the Interface Builder Library, drag an instance of a toolbar (`UIToolbar`) to top of the view. The toolbar object includes, by default, a single button called Item. Double-click the button to change its title to Configure; the button will automatically resize itself to fit the label.

In this application, the single button is all that is needed. If your project needs more, you can drag Bar Button Items from the library into the toolbar. The buttons are shown as subviews of the toolbar within the Interface Builder Document window.

Figure 11.7 shows the final interface and the Document window showing the interface hierarchy.

FIGURE 11.7
Labels, and toolbar, and a toolbar button complete the interface.

Connecting the Outlets and Actions

It's time to connect the interface we've built to the outlets and actions we defined in the view controller. Control-drag from the File's Owner icon in the IB Document window to the first On label, connecting to the `weekendOutput` outlet, as shown in Figure 11.8. Repeat for the other three labels, connecting to `weekdayOutput`, `amOutput`, and `pmOutput`.

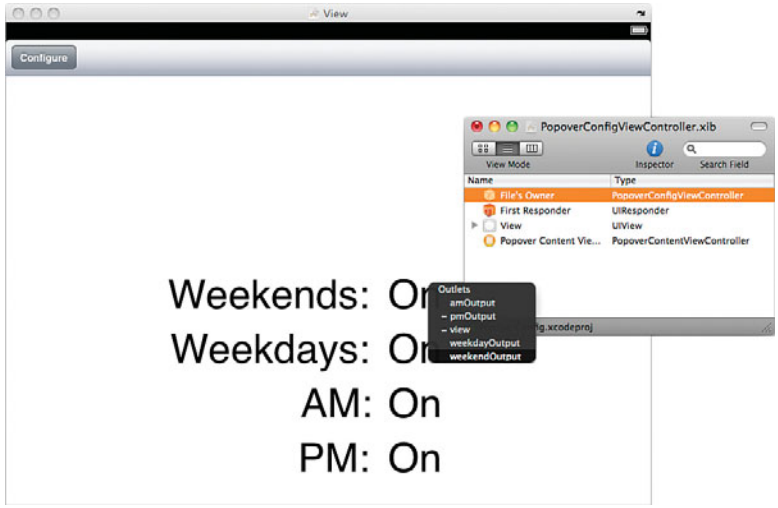

FIGURE 11.8. Connect each output label to its outlet.

Next, Control-drag from the Configure toolbar button to the File's owner icon. Choose `showPopover` when prompted, as shown in Figure 11.9. Note that we didn't have to worry about connecting from a Touch Up Inside event because toolbar buttons have only one event that can be used.

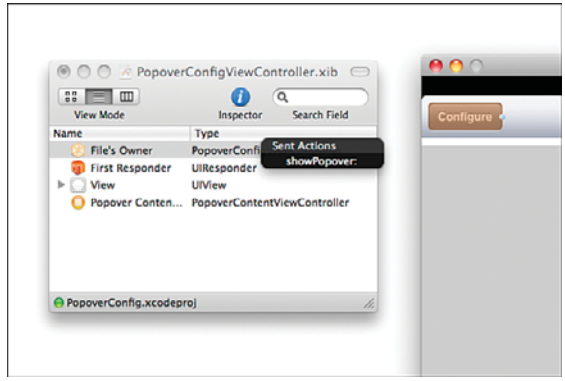

FIGURE 11.9 Connect the configure button to the `showPopover` action.

Only one step remains to be completed in interface builder: instantiating the popover content view controller.

Instantiating the Popover Content View Controller

Earlier in the tutorial, we developed the popover content view controller and view (`popoverContentViewController`). What we haven't done, however, is actually use it anywhere. We can take two approaches to creating an instance of the controller so that we can use it in the application:

1. The content view controller is instantiated whenever the popover is invoked, and released when the popover is dismissed.
2. The content view controller is instantiated when main application view loads and is released when the application is finished.

I've chosen to go with approach number 2. By instantiating the popover's view controller when the main application view loads, we can use it repeatedly without reloading the view. This means that if the user displays the popover and updates the switches, those changes will be visible no matter how many times the user dismisses or opens the popover.

> If you are creating an application with *many* popovers, go with method 1; otherwise, all the views will be kept in memory simultaneously.

Did you Know?

Without adding any code, we can instantiate `popoverContentViewController` when the popoverConfigViewController.xib file is loaded:

1. Open the popoverConfigViewController.xib file's document window in Interface Builder.
2. Drag a View Controller object from the Library into the document window.
3. Select the view controller in the Document window, and press Command+4 to open the Identity Inspector.
4. Set the class to the `popoverContentViewController` rather than the generic `UIViewController` class set by default. This can be seen in Figure 11.10.

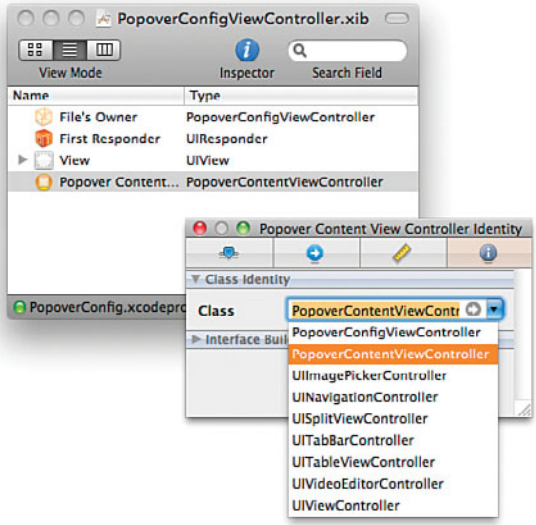

FIGURE 11.10
Set the object to be an instance of `popoverContentViewController`.

5. Switch to the Attributes Inspector (Command+1) and set the NIB name field to **popoverContentViewController** so that the view controller knows where its view is stored.
6. Close the Inspector window.
7. Control drag from the File's Owner icon to the popover content view controller icon within the Document window. Choose `popoverContent` when prompted, as shown in Figure 11.11.
8. Save the XIB file.

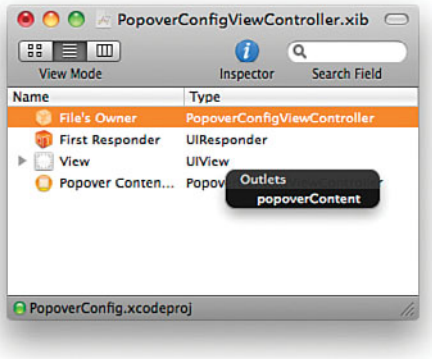

FIGURE 11.11
Connect the popover content view controller to the `popoverContent` outlet.

Our views and view controllers are completed. All that remains is writing the code that handles the application logic.

Implementing the Application Logic

We need to implement two methods to complete this tutorial. First, we need to implement `showPopover` to display the popover and allow the user to interact with it. Second, the popover controller delegate method `popoverControllerDidDismissPopover` must be built to take care of cleaning up the popover when the user is done with it, and to update the application's view with any changes the user made within the popover.

Displaying the Popover

Open the popoverConfigViewController.m file and add the `showPopover` method, shown in Listing 11.3, immediately following the `@synthesize` directives.

LISTING 11.3

```
1: -(IBAction)showPopover:(id)sender {
2:     if (popoverController==nil) {
3:         popoverController=[[UIPopoverController alloc]
4:             initWithContentViewController:popoverContent];
5:         [popoverController presentPopoverFromBarButtonItem:sender
6:          permittedArrowDirections:UIPopoverArrowDirectionAny animated:YES];
7:         popoverController.delegate=self;
8:     }
9: }
```

There are three steps to displaying and configuring the popover.

First, in lines 3–4, the popover controller, `popoverController`, is allocated and initialized with the popover's content view, `popoverContent`.

Second, lines 5 and 6 display the popover using the (very verbose) method `presentPopoverFromBarButtonItem:permittedArrowDirections:animated`. The bar button item (our toolbar button) can be referenced through the `sender` variable, which is passed to `showPopover` when the button is pressed. The `permittedArrowDirections` parameter is passed the constant `UIPopoverArrowDirectionAny`, meaning the popover can be drawn with an arrow that points in any direction (as long as it points to the specified interface element). The `animated` parameter gives the iPad the go-ahead to animate the appearance of the popover (currently a nice fade-in effect).

Third, line 7 sets the popover controller's delegate to the same object that is executing the code (`self`)—in other words, the `popoverConfigViewController`. By doing this, the popover controller will automatically call the method `popoverControllerDidDismissPopover` within popoverConfigViewController.m when the user is done with it.

Nothing too scary, right? Right. But what about lines 2 and 8? The entire display of the popover is wrapped in an `if-then` statement. The reason for this can be easily demonstrated by removing the `if-then` and running the application. Without the conditional, multiple copies of the popover will be displayed (one on top of the other) each time the Configure button is pressed. This is a large memory leak and would make the application behave very strangely for the user. To get around the problem, we perform a simple comparison: `popoverController==nil`. When the popover controller hasn't been initialized, it will have a value of `nil` (that is, no value at all). In this case, the statements to initialize the controller and show the popover are executed. Once the popover is displayed, however, the `popoverController` has a value and will no longer equal `nil`, keeping any further instances of it from being displayed.

Of course, we want the user to be able to dismiss and redisplay the popover, so we need to release the `popoverController` and set it back to `nil` when we hide the popover again. Let's look at that implementation now.

Did you Know?

What Constants Can I Provide for a Popover's Arrow Direction?

You can force the popover's arrow (and subsequent onscreen positioning) by using one of five different constants:

`UIPopoverArrowDirectionUp`—The popover points up toward the interface element.

`UIPopoverArrowDirectionDown`—The popover points down toward the interface element.

`UIPopoverArrowDirectionLeft`—The popover points left toward the interface element.

`UIPopoverArrowDirectionRight`—The popover points right toward the interface element.

`UIPopoverArrowDirectionAny`—The popover can be oriented in whatever position the iPhone OS finds most appropriate.

Apple recommends using the "Any" option whenever possible in your applications.

Reacting to the Popover Dismissal

When the user gets rid of the popover by touching outside of its content area, we want our application to react and display any changes the user made within the popover view. We also want to prepare the popover's controller to show the popover again. Enter the popover controller delegate method `popoverControllerDidDismissPopover` as shown in Listing 11.4.

LISTING 11.4

```
1: -(void)popoverControllerDidDismissPopover:
2:               (UIPopoverController *)controller {
3:     weekdayOutput.text=@"On";
4:     weekendOutput.text=@"On";
5:     amOutput.text=@"On";
6:     pmOutput.text=@"On";
7:
8:     if (!popoverContent.weekdaySwitch.on) {
9:         weekdayOutput.text=@"Off";
10:    }
11:    if (!popoverContent.weekendSwitch.on) {
12:        weekendOutput.text=@"Off";
13:    }
14:    if (!popoverContent.amSwitch.on) {
15:        amOutput.text=@"Off";
16:    }
```

```
17:     if (!popoverContent.pmSwitch.on) {
18:         pmOutput.text=@"Off";
19:     }
20:     [popoverController release];
21:     popoverController=nil;
22: }
```

Most of the display logic used in this method should be familiar to you by now. Lines 3–6 set the four output labels to `On`, because this is the default state of our switches. Lines 8–19 are simple `if-then` statements which check to see whether a switch is *not* set to on, and, if so, sets the corresponding output label to `Off`.

Because we have an instance variable for the popover's view controller (`popoverContent`) and have defined the `UISwitches` as properties, we can access the individual state of a given switches using its `on` property in a single line: `popoverContent.<switch instance variable>.on`.

In the final two lines, 20 and 23, the popover controller is released and its instance variable (`popoverController`) set to `nil`. This prepares it for the next time the user presses the Configure button.

Did you Know?

It might surprise you to learn that releasing an object does not automatically set its instance variable to `nil`. In fact, the instance variable is *not* changed at release and will reference a nonexistent object, potentially causing major problems if you attempt to use it.

Watch Out!

You might be wondering why we didn't just use the `controller` reference rather than `popoverController` instance variable. The answer is that we need to be able to set the `popoverController` variable to `nil`. If we use `controller`, we reference the same object as `popoverController`, but setting `controller` to `nil` doesn't change the value of `popoverController`.

The application is now complete. Use Build and Run to test the popover's display on your iPad. You've just implemented one of the most important and flexible UI features available on the iPad platform!

Further Exploration

In this hour's sample project, you attached a toolbar to a "bar button" (toolbar button) using the `presentPopoverFromBarButtonItem:permittedArrowDirections:animated` method. This, granted, is a very popular approach, but you can create popovers

anywhere within your view by using the `UIPopoverController` method `presentPopoverFromRect:inView:permittedArrowDirections:animated`. With this method you can present the popover so that it appears from any rectangular areas, within any view. In addition, popover content does not need to be static! If you'd like, your popover's view controller can update its content on-the-fly, and the popover will update dynamically to display the changes. You'll need to manually update the `popoverContentSize` property of the controller so that all of your content fits, but size changes are animated smoothly for the end user.

To learn more about popovers, be sure to review Apple's `UIPopoverController` class reference within the developer documentation to get a complete picture of this important class and UI element.

Summary

Popovers provide a canvas for creating a range of unique interface elements that can be displayed virtually anywhere in your application. The approach that we took in this hour's lesson (creating a popover that is displayed when a toolbar button is pressed) is the most common implementation that you'll encounter.

You've learned not only how a popover is designed and displayed, but how to access data from within its view, and ways of keeping the popover controller from getting out of hand. In the next hour's lesson, you'll learn about several UI elements that Apple will allow *only* if they are displayed from within a popover. So even if you can't think of any uses for them yet, chances are, you will!

Q&A

Q. ***Can I have multiple popover's within a single application view?***

A. Yes, you can, but keep in mind that the example here uses a single delegate for handling the dismissal of a popover. There are a number of ways to get around this, including structuring your code so that changes within a popover are immediately reflected in the application, or you can segment your application so that each popover has a different delegate.

Q. ***You told me to drag the toolbar to the top of the window. The developer docs say to drag it to the bottom. What gives?***

A. At the time of this writing, Apple has not yet updated all the descriptions of the toolbar UI element to state that it can be used at the top and bottom of the iPad screen.

Workshop

Quiz

1. What class is a toolbar button?
2. How do you set where a popover appears?
3. Why do we need to compare the popover controller to `nil` before initializing it?

Answers

1. A toolbar button is an instance of the bizarrely named class `UIBarButtonItem`.
2. The iPhone OS determines where a popover appears onscreen. Setting the `permittedArrowDirections` parameter when displaying the popover, however, limits where the OS may position the popover so that it can be drawn with an arrow pointing to the UI element invoking it.
3. If the popover controller is *not* `nil`, that means the popover is visible onscreen and a new copy of it should not be created.

Activities

1. Explore the possibilities of popovers outside of toolbars. Implement an additional button (`UIButton`) within the popoverConfig application that displays the same popover, but located in the center of the screen.
2. Implement a second toolbar-based popover within the popoverConfig application. If you choose to use a single delegate for each, you can check to see which popover is being dismissed by comparing each controller instance variable to the controller variable passed to the `popoverControllerDidDismissPopover` method.

This page intentionally left blank

Index

Symbols

A

B

C

D

E

F

G

H

I

J–K

L

M

N

O

P

Q–R

S

T

U

V

W

X–Y–Z