This is your code cookbook reference
for iOS 7 apps development

iOS 7 Development Recipes

A Problem-Solution Approach

Joseph Hoffman | Hans-Eric Grönlund | Colin Francis | Shawn Grimes

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

About the Authors xix

About the Technical Reviewer xxi

Acknowledgments xxiii

Introduction xxv

■ Chapter 1: Application Recipes 1

■ Chapter 2: Storyboard Recipes 59

■ Chapter 3: Layout Recipes 93

■ Chapter 4: Table and Collection View Recipes 135

■ Chapter 5: Location Recipes 199

■ Chapter 6: Motion Recipes 241

■ Chapter 7: Map Recipes 273

■ Chapter 8: Social Network Recipes 343

■ Chapter 9: Camera Recipes 387

■ Chapter 10: Multimedia Recipes 439

■ Chapter 11: Image Recipes 477

■ Chapter 12: Graphics Recipes 511

■ Chapter 13: Animation and Physics Recipes 537

Chapter 14: User Data Recipes 571
Chapter 15: Data Storage Recipes 623
Chapter 16: Data Transmission Recipes 679
Chapter 17: Game Kit Recipes 711

Index 767

Introduction

The easy part of software development is knowing how to write code in the programming language at hand. The tougher part is mastering the programming interfaces of the platform and getting to the level where you can effectively turn ideas into working features with real values. iOS 7, although extremely powerful and easy to use, is no exception to this. Objective-C, considered by many to be a somewhat "funky" programming language, is a language you can get your head around quickly and even learn to appreciate. However, you're likely to spend a lot of time learning the various APIs and frameworks.

We believe the best way to acquire the necessary knowledge and reach a plateau of high productivity is through hands-on experience. We think the best way to learn is to follow step-by-step procedures, creating small projects in which you can test and tweak the features and get a feel for them before you implement them in your real projects.

With this idea in mind, we created ***iOS 7 Development Recipes***. This book contains over 600 pages of sample code accompanied by instructions about how to create small test apps that allow you to run the code on your iOS 7 device or in the iOS simulator.

We have tried to cover as many topics as possible based on the features of iOS 7. We hope this book provides the fundamentals you need to start converting your great ideas into fantastic apps.

Who This Book Is For

When you read this book, it will help if you have a basic knowledge of Objective-C, have taken your first steps in Xcode, and have written a couple of Hello World apps. If you haven't, don't worry; simply pay extra attention to the first eight recipes of Chapter 1. They should provide most of the basics you will need in order to follow along with the recipes in the rest of the book.

How This Book Is Structured

For the most part, the example-based chapters of this book do not build on one another; that is, you do not have to read the chapters sequentially. You should be able to read the chapters and build your apps in any order you want, depending on your specific interests. Whenever we do build on an example from a different chapter, we will let you know.

However, we recommend that you at least skim Chapter 1, “Application Recipes,” then Chapter 2, “Storyboard Recipes,” and finally Chapter 3, “Layout Recipes,” before moving on.

The first chapter contains recipes for common tasks, such as creating outlets and actions, which are referenced throughout the book and should be fully understood before you move ahead.

The second chapter covers storyboards. Storyboards are an alternative way of building user interfaces. Storyboards show both the individual screens (single view controllers) and the connections (segues) between them. Before the introduction of storyboards in iOS 5, .xib files were typically used to build the interface. The .xib files were individual scenes; that is, one .xib file per scene. The connections between .xib files were handled in code. While you still have the ability to use .xib files, it is clear that Apple is pushing developers more and more to use storyboards instead.

The third chapter provides basic knowledge of the layout tools for iOS 7. Reading that chapter might be helpful when you create the user interfaces of the recipes later, especially if you want your scenes to look good on multiple devices.

This book uses a mix of programmatic interface building, storyboards, and .xib files. For the most part, you will start every application with a single view application template for the iPhone that will include a storyboard. We will instruct you otherwise if this is not the case.

Throughout this book, we assume you are developing in the latest versions of iOS and Xcode, which at the time of writing are iOS 7.0 and Xcode 5.0.

Many of the recipes in this book cannot be fully tested on the iOS simulator, and as such will require both an iOS device and a provisioning profile, which you can acquire when you subscribe to the iOS Developer Program. At the time of this writing, the developer program costs $99.00 USD. We’ve pointed out each recipe that cannot be tested in the iOS simulator.

> **Note** Since the introduction of iPhone 5, there are two screen sizes to account for: the 3.5" screen and the 4" screen. The width in both screens is the same, but the 4" screen is taller. Most of the interfaces in this book are laid out to look good on a 3.5" screen and will have unused space if run on a 4" screen. Feel free to modify the interface to better fit a 4" screen if that is what you want. Chapter 3 discusses Auto Layout, which will give you the tools to make an interface look good on either size screen.

What's New in This Edition

By now you undoubtedly have seen the fresh look of iOS 7. Because of the new software, we've made numerous changes to the images and code we provided in the previous edition of this book, ***iOS 6 Recipes: A Problem-Solution Approach***. While we couldn't address everything iOS 7 has to offer in this new edition, we tried to add some topics we found very useful to the average developer.

Because iOS 7 is a major overhaul in design, all the images and figures have been updated to reflect the new design. Apple has made numerous minor changes to existing APIs for iOS 7. We updated the code in this book to reflect those changes.

Due to Apple's added emphasis on storyboards, we have changed the examples in this book to take advantage of storyboards rather than .xib files. Previous editions of this book were largely based on .xib files. In addition, we created a new chapter for storyboards (Chapter 2), which now includes instructions for creating tab view controllers.

Auto Layout has changed with iOS 7, so we have updated Chapter 3 to reflect these changes. Auto Layout constraints no longer exist for the developer. This greatly increases developer freedom and flexibility.

Apple added nice features to the Map Kit framework, which allow you to get directions from Apple as well as take full advantage of 3-D map support. We have included recipes that show you how to use both of these new features in Chapter 7.

iOS 7 gives us the ability to detect and read QR codes with the camera natively. We added a new recipe to Chapter 9, "Camera Recipes," that shows how to read QR codes.

We also added Chapter 12, which focuses on Core Graphics. While the previous edition touched on Core Graphics, we have expanded the content in its own chapter.

The most exciting contribution to this new edition is a new chapter on animation (Chapter 13). This chapter introduces some UIView animation basics and then moves on to the new UIKit Dynamics framework. This framework allows for realistic, physics-inspired animation.

Downloading the Code

The code for the examples included in this book is available on the Apress web site at www.apress.com. You can find a link on the book's information page under the Source Code/Downloads tab. This tab is located under the Related Titles section of the page.

Contacting the Author

If you have any questions or comments regarding this book, I would be happy to hear them. Contact me at NSCookbook@gmail.com and include "iOS 7 Recipes" in the subject line, or write a comment on my blog at http://www.NSCookbook.com.

Chapter 1

Application Recipes

This chapter serves as both a refresher for creating apps for iOS and a foundation for completing the basic tasks that are repeated throughout this book. The first eight recipes walk you through fundamental tasks such as setting up an application, connecting and referencing user interface elements in your code, and adding images and sound files to your project. The knowledge you will acquire from the recipes in this chapter is necessary to complete many of the tasks in the other chapters of this book.

The last four recipes in this chapter deal with select, useful tasks such as setting up simple APIs for default error and exception handling, including a "lite" version of your app in your projects, and making an app launch seem quick and easy in the eyes of the user. For the most part, these tasks won't be repeated in the remainder of this book; however, we feel they are essential knowledge for any developer.

Recipe 1-1: Setting Up a Single View Application

Many of the recipes in this book are implemented in a test application with a single view. Such a project is easy to set up in Xcode using the Single View Application template. This template enables you to use storyboards.

A storyboard is a way to build interfaces consisting of multiple scenes and their connections. Before the introduction of storyboards in iOS 5, .xib files were typically used to build the interface for individual scenes; in other words, there was one .xib file per scene. The connections between .xib files were handled in code. While you still have the ability to use .xib files, Apple is pushing developers to use storyboards instead. As such, you will be using storyboards for most of this book.

To create a new single view application, go to the main menu and select File ➤ New ➤ Project. This brings up the dialog box with available project templates (see Figure 1-1). The template you want is located on the Application page under the iOS section. Choose "Single View Application" and click "Next."

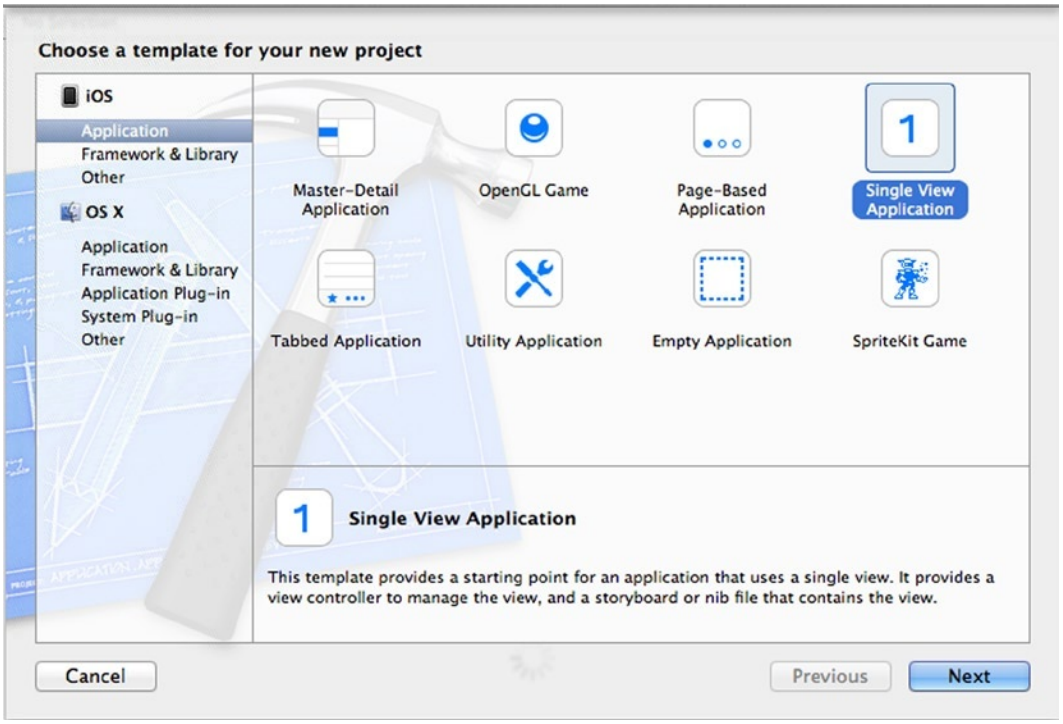

Figure 1-1. *The single view application template in the iOS application section*

Enter a few properties for your Application recipes:

- A *Product Name*, such as **My Test App**
- An *Organization Name*, which can be your name, unless you already have an organization name
- A *Company Identifier*, which preferably is your Internet domain, if you have one

If you like, you can also enter a class prefix that will be applied to all classes you create using the Objective-C file template. This can be a good idea if you want to avoid future name conflicts with third-party code; however, if this app is meant only for testing a feature, you can leave the class prefix item blank.

You also need to specify which device type your application is for: iPad, iPhone, or both (Universal). Choose iPhone or iPad if you are testing. You can also pick Universal, but then the template will generate more code, which you probably don't need if your only purpose is trying a new feature. Figure 1-2 shows the properties you need to fill out in the project options window.

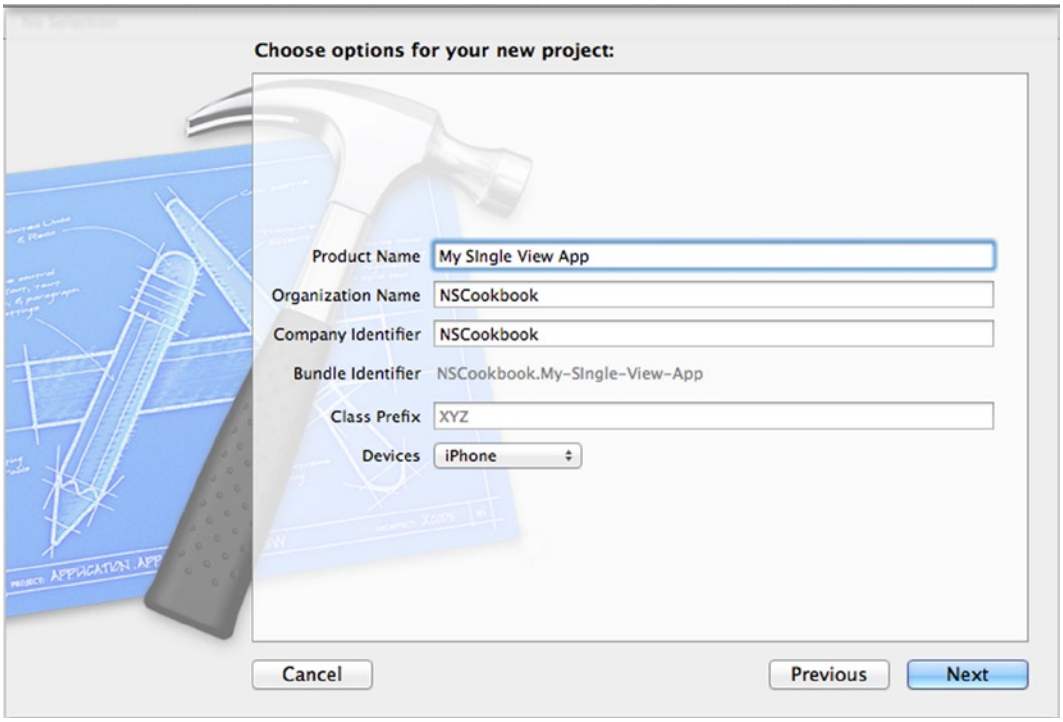

Figure 1-2. *Configuring the project*

Click the “Next” button and then select a folder where the project will be stored. Bear in mind that Xcode creates a new folder for the project within the folder you pick, so select the root folder for your projects.

There’s often a good reason to place the project under *version control*. It allows you to check changes to the code so you can go back to a previous version if something goes wrong or if you simply want to see the history of changes to the application. Xcode comes with Git, a feature-rich, open-source version-control system that allows multiple developers to work on a project simultaneously with ease. To initialize it for your project, check the “Create local git repository for this project” checkbox, as in Figure 1-3. As of Xcode 5, you can specify a server as well as your Mac for this repository.

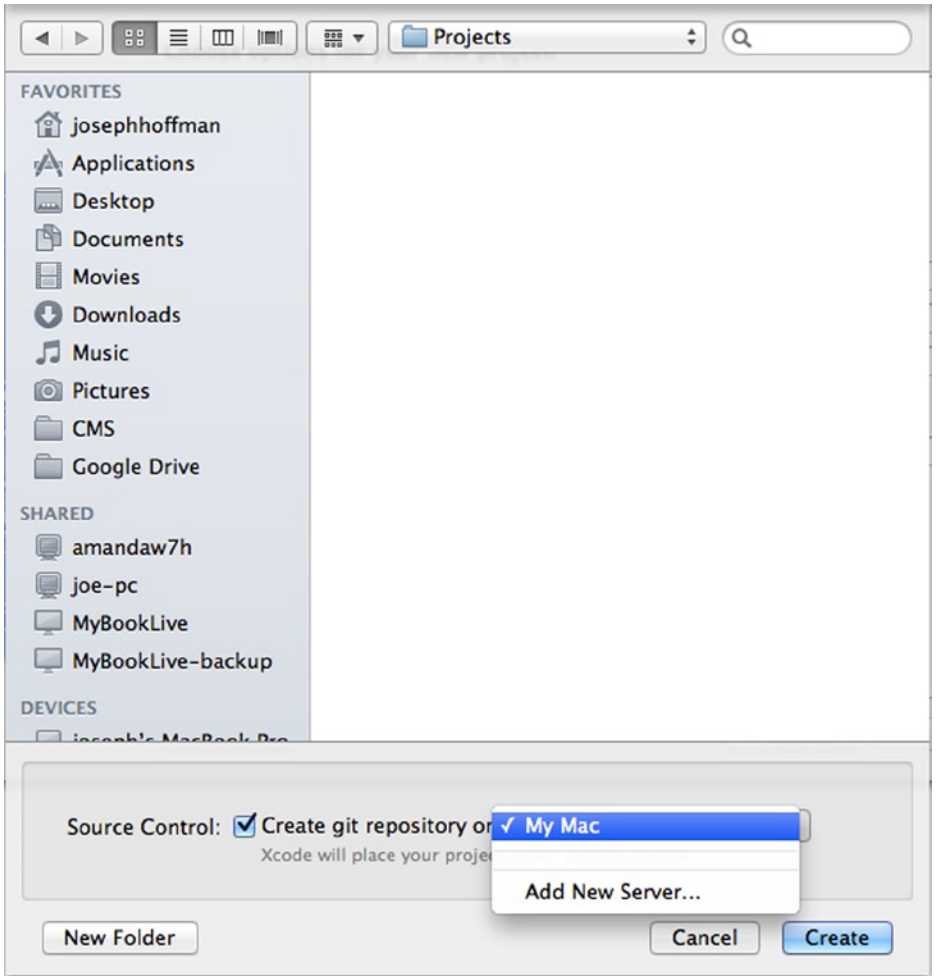

Figure 1-3. *Selecting the parent folder for the project*

Now click the “Create” button. An application with an app delegate, a storyboard, and a view controller class will be generated for you (see Figure 1-4).

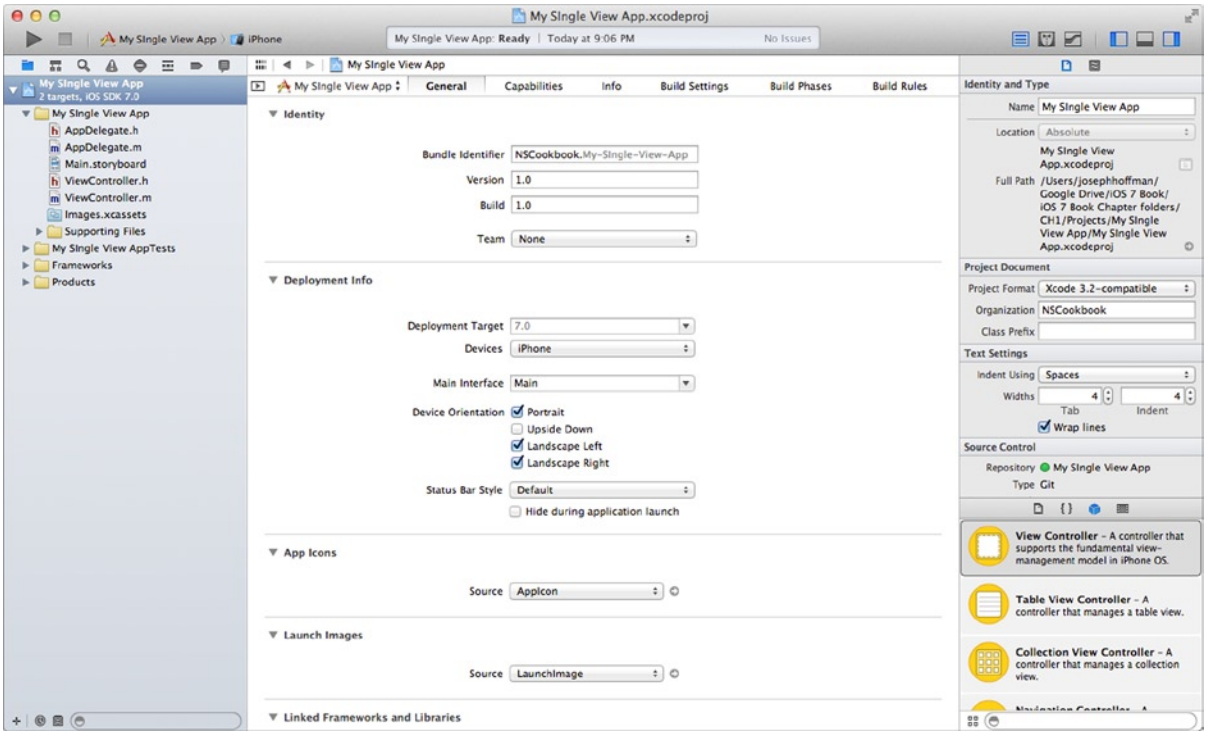

Figure 1-4. *A basic application with an app delegate and a view controller*

The setup is now complete, and you can build and run the application (which at this point shows only a blank screen).

At this point, you have a good foundation on which you can build for the next seven recipes.

An Alternate Way

Whether or not to use storyboards is a topic of debate among developers. Some developers love storyboards, while others dislike them. As a developer, it is likely you will work on projects that don't use storyboards or that use a mix of .xib files and storyboards. We won't argue for or against storyboards, but many developers tend to agree that storyboards can present difficulties when using version control and when multiple developers need to work on the same storyboard. With this in mind, it might be beneficial to learn the .xib approach. Learning this approach is entirely optional, so you can move on to Recipe 1-2 if you choose.

To create an empty application and add a ViewController class with an accompanying .xib file, you first need to create a new project. Choose "Empty Application" instead of "Single View Application" (refer to Figure 1-1) and then click "Next."

The next screen will look almost the same as Figure 1-2. This time, there will be a new check box called "Use Core Data." Leave it cleared and click "Next." Again, you'll be prompted for a save location. Find a suitable location to save and click "Create."

Upon creation of your new, empty application, you'll see there are no ViewController.m or ViewController.h files; you will need to create those. Click the "+" sign in the lower-left corner of the Xcode window and select "new file" (see Figure 1-5). You can also press Cmd + N.

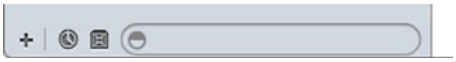

***Figure 1-5.** Creating a new file in Xcode*

Next, you'll be prompted to choose a template for your new file. Choose "Objective-C class," as shown in Figure 1-6.

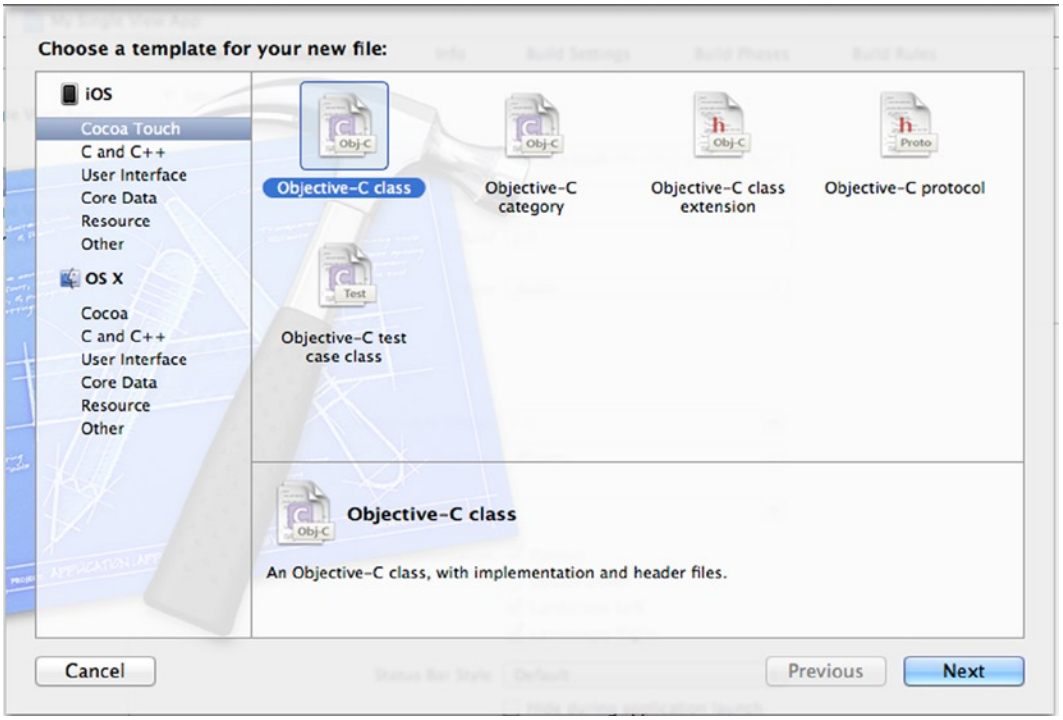

***Figure 1-6.** Creating a new file in Xcode*

You will be prompted for file options. Give your new class the name of "ViewController" and choose "UIViewController" from the drop-down box (see Figure 1-7). Make sure you select the "With XIB for user interface" check box. Click "Next."

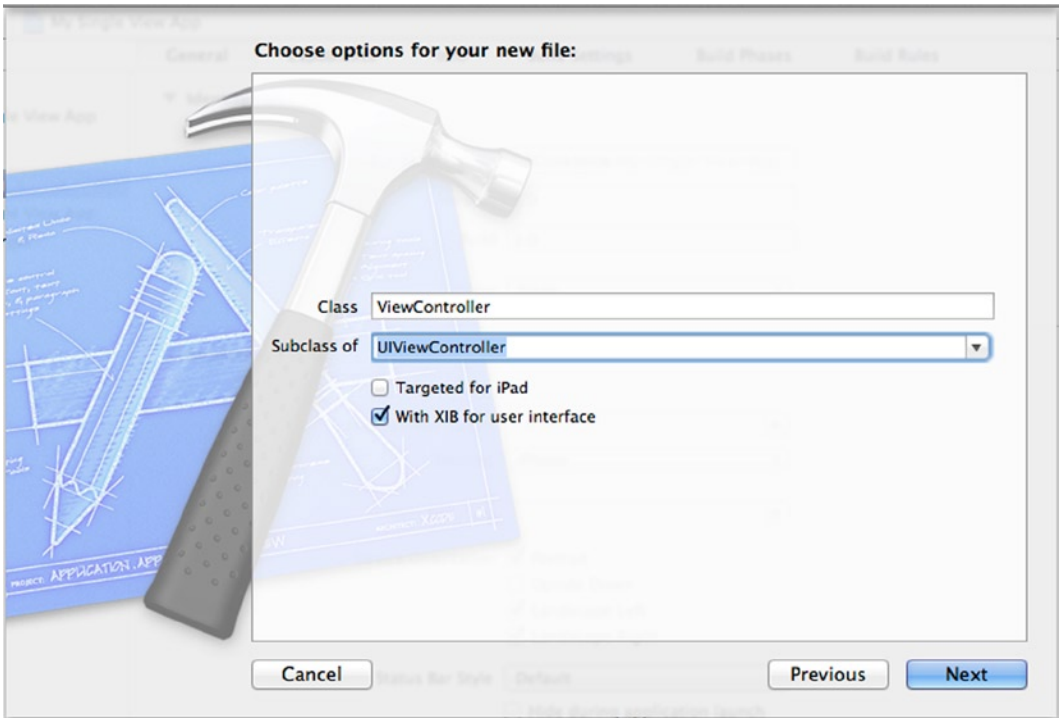

Figure 1-7. *Choosing options for a new file*

Now that you have created the class, you will need to select the AppDelegate.m and AppDelegate.h files from the Project Navigator on the left and then modify the code, as shown in Listing 1-1.

Listing 1-1. Adding properties to the single view AppDelegate.h file

```
AppDelegate.h

#import <UIKit/UIKit.h>

@class ViewController;
#import "ViewController.h"

@interface AppDelegate : UIResponder<UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;
@property (strong, nonatomic) ViewController *viewController;

@end
```

```
AppDelegate.m

#import "AppDelegate.h"

@implementation AppDelegate

- (BOOL)Application recipes:(UIApplication *)application didFinishLaunchingWithOptions:
(NSDictionary *)launchOptions
{
self.window = [[UIWindowalloc] initWithFrame:[[UIScreenmainScreen] bounds]];
// Override point for customization after application launch.
self.viewController = [[ViewControlleralloc] initWithNibName:@"ViewController" bundle:nil];
self.window.rootViewController = self.viewController;
    [self.windowmakeKeyAndVisible];
returnYES;
}
```

Once you are finished, you should have a single view application with a single .xib file, as shown in Figure 1-8.

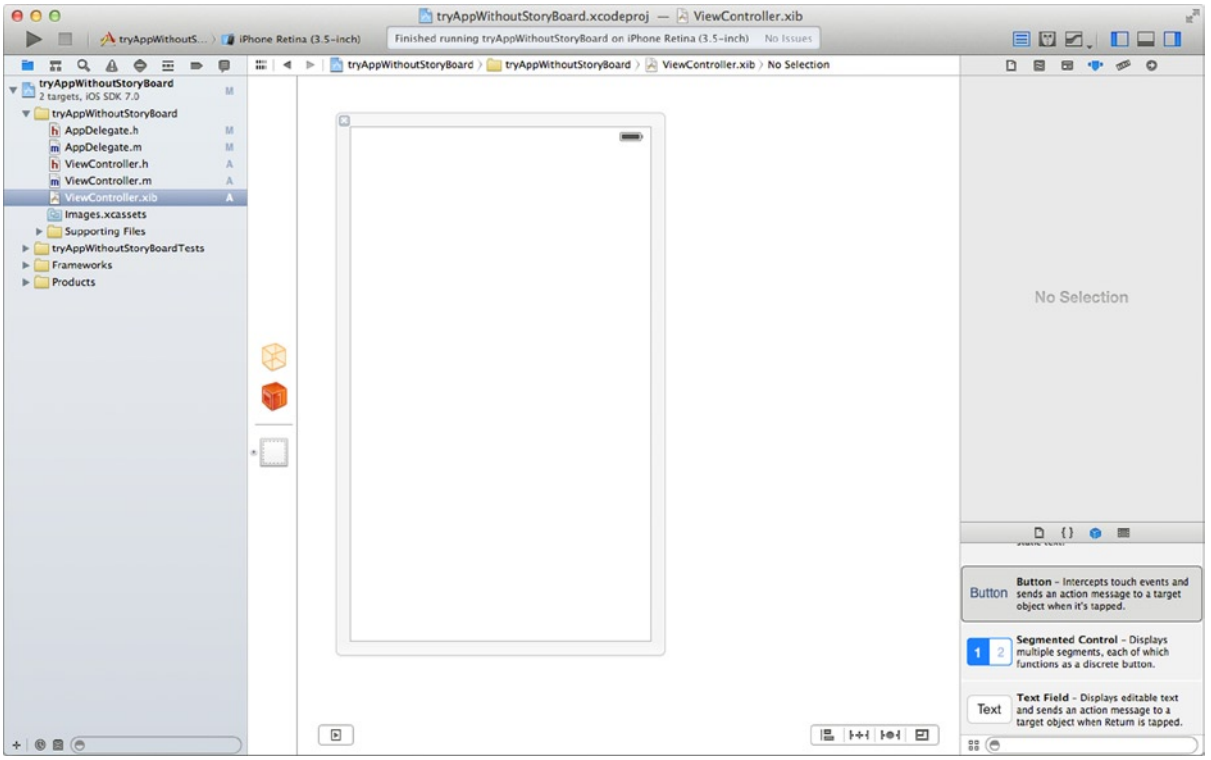

Figure 1-8. A single view application with the .xib approach

Recipe 1-2: Linking a Framework

The iOS operating system is organized into *frameworks*. A framework is a directory of code libraries and resources that are needed to support the library. To use the functionalities of a framework, you need to link the corresponding binary to your project. For `UIKit`, `Foundation`, and `CoreGraphics` frameworks, Xcode does this automatically when you create a new project. However, many important features and functions reside in frameworks such as `CoreMotion`, `CoreData`, `MapKit`, and so on. For these types of frameworks, you need to follow the following steps to add them:

1. Select the project node (the root node) in the project navigator panel on the left of the Xcode project window (Figure 1-4). This brings up the project editor panel.
2. Select the target in the Targets dock, as shown on the left of Figure 1-9. If you have more than one target, such as a unit test target, you need to perform these steps for each of them.

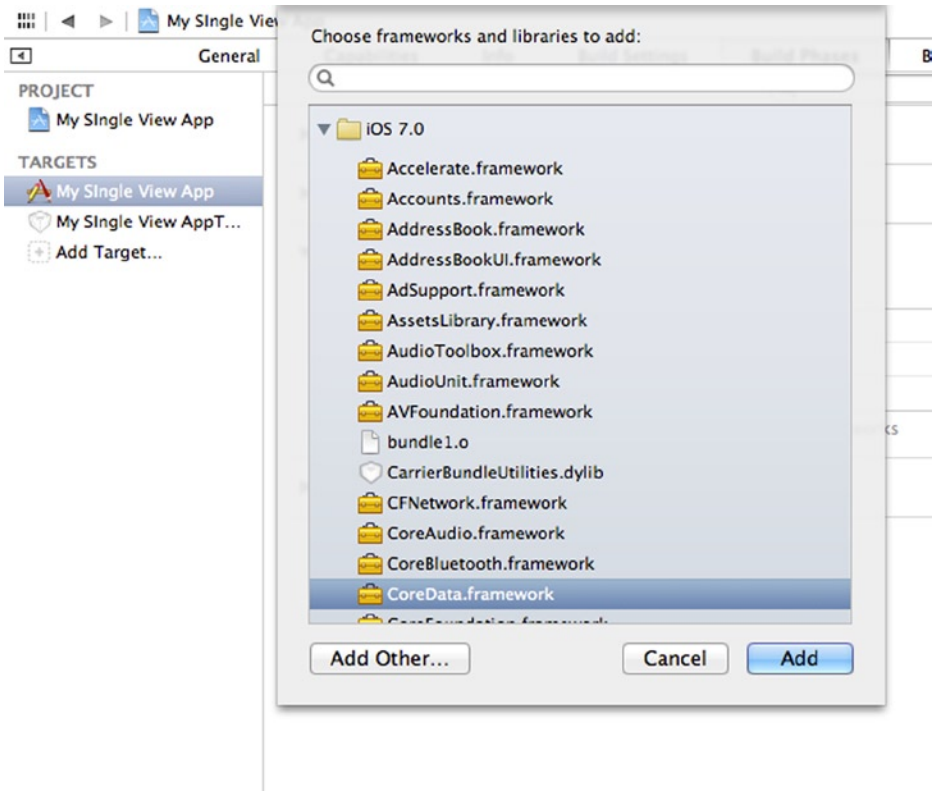

***Figure 1-9.** Adding the Core Data framework*

3. Navigate to the Build Phases tab and expand the Link Binary with Libraries section. There you will see a list of the currently linked frameworks. Alternatively, you can scroll to the bottom of the page under the General tab.

4. Click the “Add items (+)” button at the bottom of the list. This brings up a list of available frameworks.
5. Select the framework you want to link and use the “Add” button to include it (see Figure 1-9).

> **Tip** To make it easier to find a particular framework, you can use the search field to filter the list.

When you add a framework to your project, a corresponding framework reference node is placed in your project tree (see Figure 1-10).

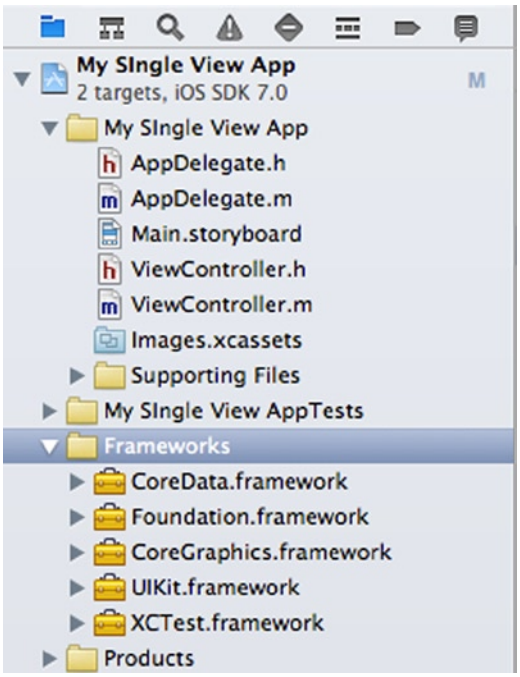

***Figure 1-10.** When adding a framework, a reference node is created inside the framework group of your process tree*

Now, to use the functions and classes from within your code, you only need to import the framework. This is normally done in a header file (.h) within your project, as shown in Listing 1-2, where we import `CoreData` in the `ViewController.h` file.

Listing 1-2. Importing the CoreData framework

```
//
//  ViewController.h
//  My Single View App
//
```

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

@interface ViewController : UIViewController

@end
```

> **Note** If you don't know the header file for a framework, don't worry. All framework APIs follow the same pattern, namely `#import <FrameworkName/FrameworkName.h>`.

With the framework binary linked and the API imported, you can start using its functions and classes in your code.

Recipe 1-3: Adding a User Interface Control View

iOS provides a number of built-in control views, such as buttons, labels, text fields, and so on, that you can use to compose your user interface. Xcode makes designing user interfaces easy with a built-in editor, Interface Builder. Interface Builder is a graphical editor, which allows you to edit both .xib files and storyboards by dragging and dropping components. All you need to do is to drag the controls you want from the object library and position them the way you want in your view. The editor helps you make a pleasing user interface by snapping to standard spaces.

In this recipe, we'll show you how to add a system button to your view. A system button is a button with default iOS styling. We'll assume you've already created a single view application in which to try this.

To create a new button, select the `Main.storyboard` file to bring up storyboard. Be sure the Utilities View (the panel on the right) is visible. If it isn't, select the corresponding button in the toolbar (see Figure 1-11).

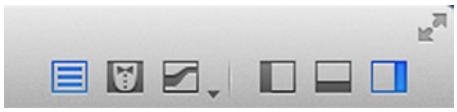

Figure 1-11. *The button to hide or show the Utilities View is located in the upper-right corner of Xcode*

Make sure the object library is visible in the Utilities View (lower-right corner of Xcode). Click the "Show the Object Library" button (see Figure 1-12) if it isn't.

Figure 1-12. The object library contains the built-in user interface controls

Now locate the button in the Object Library and drag it onto the view. Blue guidelines will help you center it, as shown in Figure 1-13.

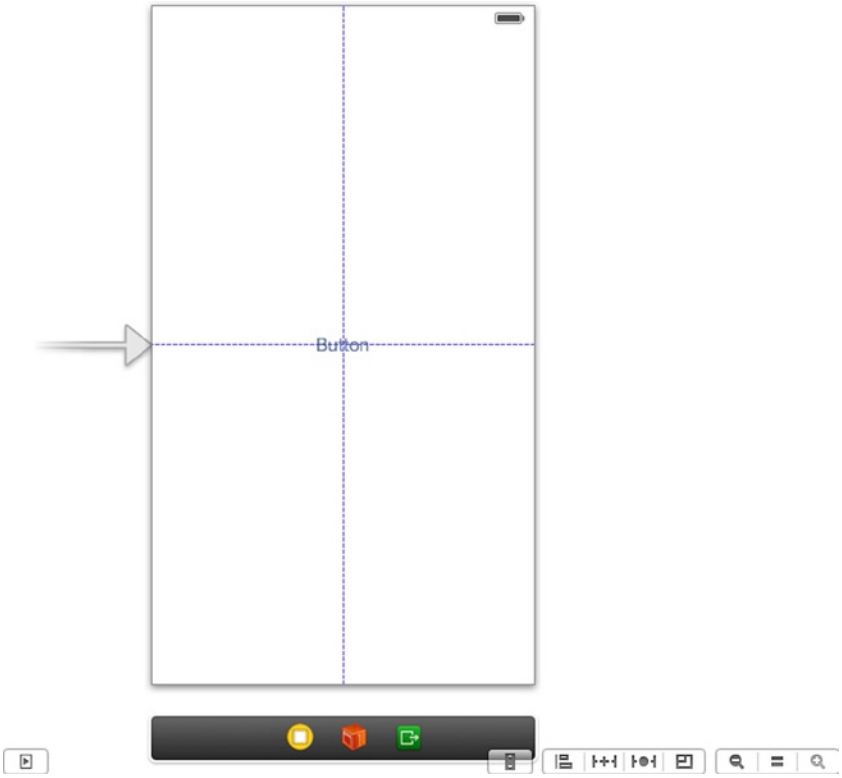

Figure 1-13. Dragging a system button from the Object Library

Change the text either by double-clicking the button or by setting the corresponding attribute in the attributes inspector, as shown in Figure 1-14. In the attributes inspector, you can also change other attributes, such as color or font. In previous versions of iOS, a button has a border. The new design philosophy set forth by Apple in iOS 7 embraces borderless buttons.

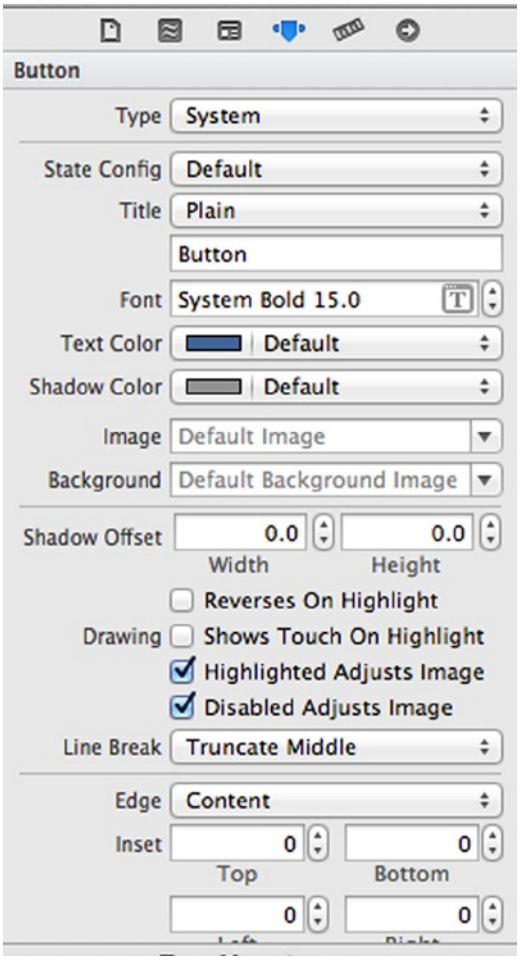

Figure 1-14. *Setting the button text in the attributes inspector*

You can now build and run your application. Your button shows, but it won't respond if you tap it. For that, you need to connect it to your code by means of outlets and actions, which are the topics of the next two recipes.

Recipe 1-4: Creating an Outlet

iOS is built on the Model-View-Controller design pattern. One effect of this is that the views are completely separated from the code that operates on the views (the so-called controllers). To reference a view from a view controller, you need to create an *outlet* in your controller and hook it up with the view. An outlet is an annotated property that connects a storyboard object to a class so you can reference it in code. Connecting an outlet can be accomplished in many different ways, but the simplest is to use Xcode's assistant editor.

We'll build on what you did in Recipe 1-3 and create an outlet for the button. Although the referenced view in this example is a button, the steps are the same for any other type of view, be it a label, text field, table view, and so on.

To create an outlet, open the main.storyboard file, select the view controller that contains the button, and click the "Assistant Editor" button in the upper-right corner of Xcode (see Figure 1-15.)

Figure 1-15. *The center button in the editor group activates the assistant editor*

With the assistant editor active, the edit area is split in two, showing Interface Builder on the left and the view controller's header file on the right. Press and hold the "Ctrl" key while dragging a blue line from the button to the code window. A hint with the text "`Insert Outlet, Action, or Outlet Collection`" should appear, as shown in Figure 1-16.

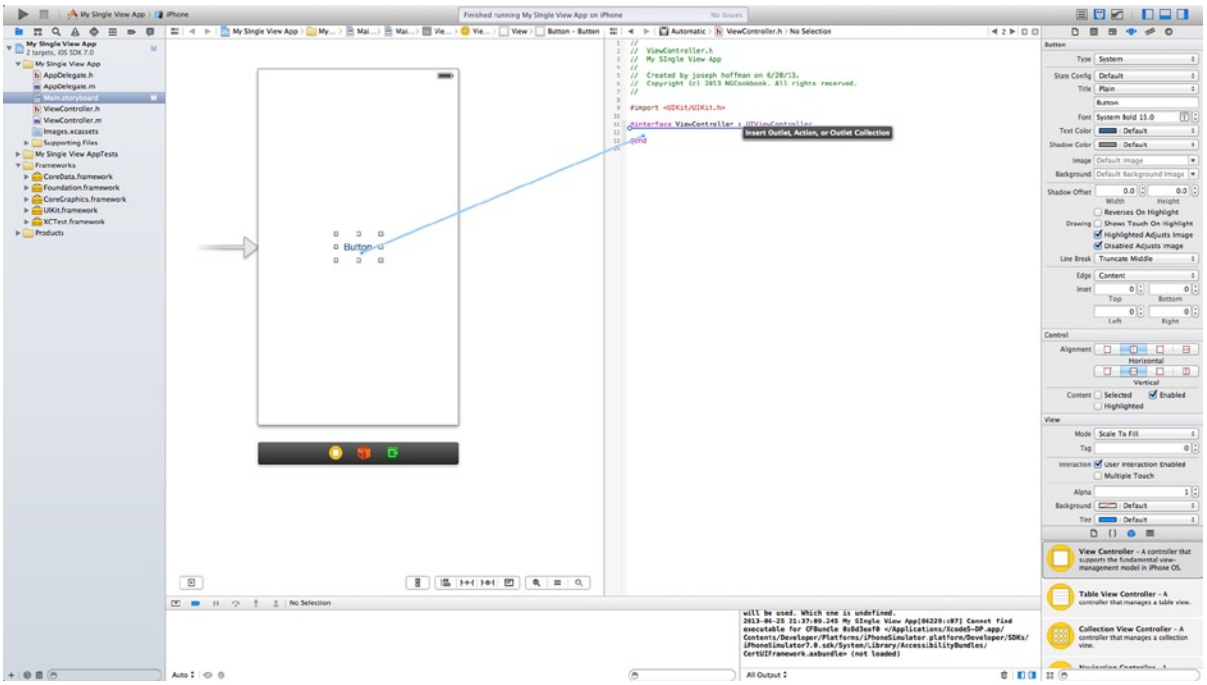

Figure 1-16. *Creating an outlet in the assistant editor using Ctrl-drag*

> **Note** Because an outlet is really only a special kind of an Objective-C property, you need to drag the blue line to somewhere it can be declared in code; that is, somewhere between the `@interface` and `@end` declarations.

In the dialog box that appears (shown in Figure 1-17), give the outlet a name. This will be the name of the property that you'll use to reference the button later from your code, so name it accordingly. Be sure that Connection is set to "Outlet" and that the type is correct (it should be `UIButton` for system buttons). Also, because you are using ARC (Automatic Reference Counting for memory management) by default, outlets should always use the `Weak` storage type.

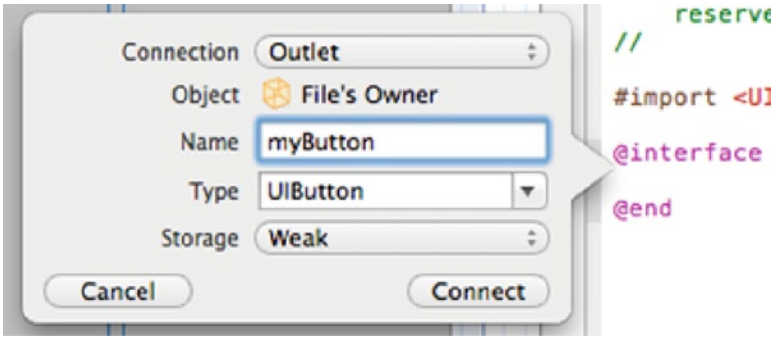

Figure 1-17. Configuring an outlet

> **Note** Although Objective-C properties should generally use the `Strong` storage type, outlets are an exception. The details are beyond the scope of this book, but briefly the reason has to do with internal memory management; using `Weak` spares you from writing certain cleanup code that you would otherwise have to write. Throughout this book, we assume that you're creating your outlets using `Weak` storage.

Click the "Connect" button. By doing this, Xcode creates a property and hooks it up with the button. Your view controller's header file should now look similar to Figure 1-18; the little dot next to the property indicates that it is connected to a view in the `storyboard` or .xib file.

```
5  //  Created by joseph hoffman on 6/20/13.
6  //  Copyright (c) 2013 NSCookbook. All rights reserved.
7  //
8
9  #import <UIKit/UIKit.h>
10
11 @interface ViewController : UIViewController
12
13 @property (weak, nonatomic) IBOutlet UIButton *myButton;
14
15 @end
16
```

Figure 1-18. An outlet property connected to a button in the storyboard

The outlet is now ready, and you can reference the button from your code using the property. To demonstrate that, add the code in Listing 1-3 to the `viewDidLoad` method in the `ViewController.m` file.

Listing 1-3. Demonstrating a referenced outlet

```
- (void)viewDidLoad
{
    [super viewDidLoad];
// Do any additional setup after loading the view, typically from a nib.
[self.myButton setTitle:@"Outlet1!" forState:UIControlStateNormal];
}
```

If you build and run your application, as Figure 1-19 shows, the button's title should now be "Outlet1!" instead of "Click Me!"

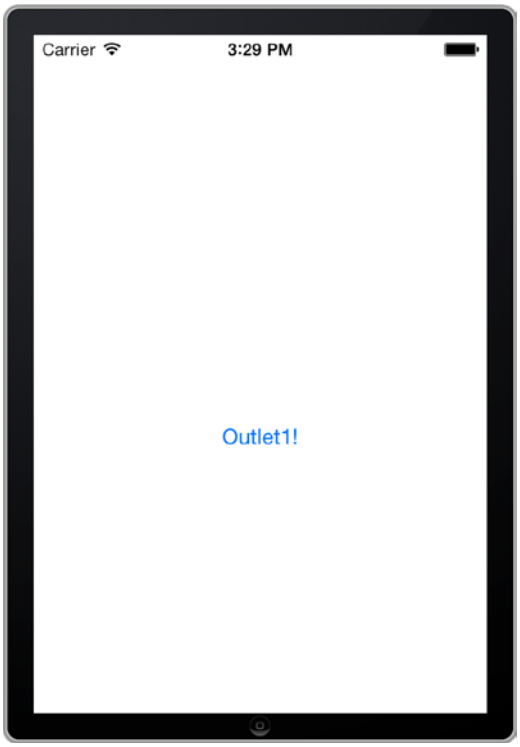

***Figure 1-19.** The button title changed from code using an outlet reference*

The next step is to make something happen when the button is tapped. This is what actions are for, which is the topic of the next recipe.

Recipe 1-5: Creating an Action

Actions are the way in which user interface controls notify your code (usually the view controller) that a user event has occurred; for example, when a button has been tapped or a value has been changed. The control responds to such an event by calling the action method you've provided.

In this recipe you will continue to build on what you've done in Recipes 1-3 and 1-4. The next few pages will help you create and connect an action method to receive "Touch Up Inside" events from the button. You then add code that displays an alert when the user taps the button.

To create an action, your Xcode should still be in assistant-editor mode with both the user interface and the header file showing. If not, press the button shown in Figure 1-15 to make it display.

Ctrl-click and drag the line from the button to the view controller's @interface section, exactly as you did when you created the outlet earlier. Only this time, change the connection type to `Action`, as in Figure 1-20.

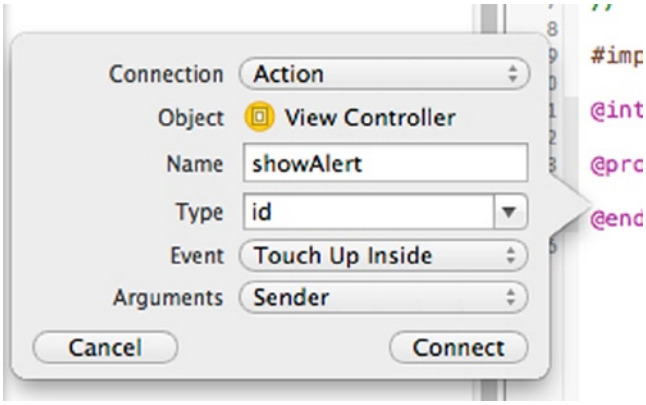

Figure 1-20. *Configuring an action method*

When you set the connection type to `Action`, you'll notice that the dialog box changes to show a different set of attributes. These attributes are different from the outlet connection type. (Compare Figure 1-20 to Figure 1-17.)The new attributes are `Type`, `Event`, and `Arguments`. Usually, the default values provided by Xcode are fine, but there might be situations where you would want to change them. The three attributes can be briefly described as follows:

- `Type`: The type of the sender argument; that is, the parameter input type for the action. This can be either the generic type ID or the specific type, which in this case is `UIButton`. It's usually a good idea to use the generic type so you can invoke the action method in other situations and not be forced to provide a `UIButton` (in this case).

- Event: This is the event type you want the action method to respond to. The most common events are touch events and events that indicate a value has changed. There are numerous kinds of touch events you can choose from.
- Arguments: This attribute dictates what arguments the action method will have. Possible values are the following:
  - None, which is no argument
  - Sender, which has the type you entered in the Type attribute
  - Sender and Event, which is an object holding additional information about the event that occurred

For the sake of this recipe, leave the attributes at id, Touch Up Inside, and Sender, respectively, but enter showAlert as the name.

> **Note** The convention in iOS is to name actions according to what will happen when an event gets triggered rather than a name that conveys the event type. Pick names such as showAlert, playCurrentTrack, and shareImage over names such as buttonClicked or textChanged.

Finalize the creation of the action by clicking the "Connect" button in the dialog box. Xcode then creates an action method in the view controller's class and hooks it up with the button. Your ViewController.h file should now look like Figure 1-21.

```
5  //  Created by joseph hoffman on 6/20/13.
6  //  Copyright (c) 2013 NSCookbook. All rights reserved.
7  //
8
9  #import <UIKit/UIKit.h>
10
11 @interface ViewController : UIViewController
12
13 @property (weak, nonatomic) IBOutlet UIButton *myButton;
14
15 - (IBAction)showAlert:(id)sender;
16 @end
17
```

Figure 1-21. *An outlet and an action connected to an object in the storyboard*

Now you're ready to implement the behavior you want when the user taps the button. In this case, you show an alert view that says "Hello Brother!" Add the code in Listing 1-4 to your ViewController.h file.

Listing1-4. Implementing the alert behavior

```
@implementation ViewController

// ...

- (IBAction)showAlert:(id)sender
{
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Testing Actions"
                                                    message:@"Hello Brother!"
                                                    delegate:nil
                                                    cancelButtonTitle:@"Dismiss"
                                                    otherButtonTitles:nil];
    [alert show];
}

@end
```

You can now build and run the application. When you tap the button, you should see your greeting alert, as in Figure 1-22.

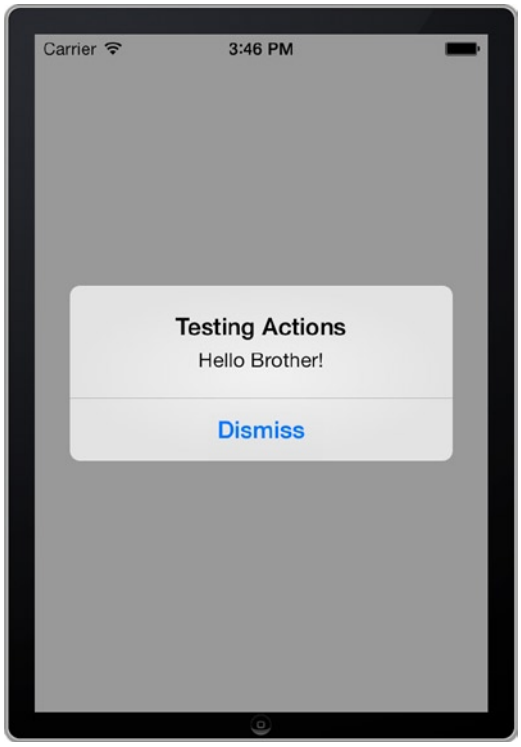

***Figure 1-22.** An action method showing an alert when the button is tapped*

Sometimes it happens that the code and the `storyboard` files get out of sync with connected outlets and actions. Usually this happens when you remove an action method or an outlet property in your code and replace it with a new one. In those cases, you get a runtime error. To fix this, remove the connection from Interface Builder in the connections inspector. The connections inspector can be found in the same pane as the attributes inspector under the circle with the arrow. Figure 1-23 shows two connected action methods for the same event inside the connections inspector. You can remove the lingering action method by clicking the "×" icon next to it.

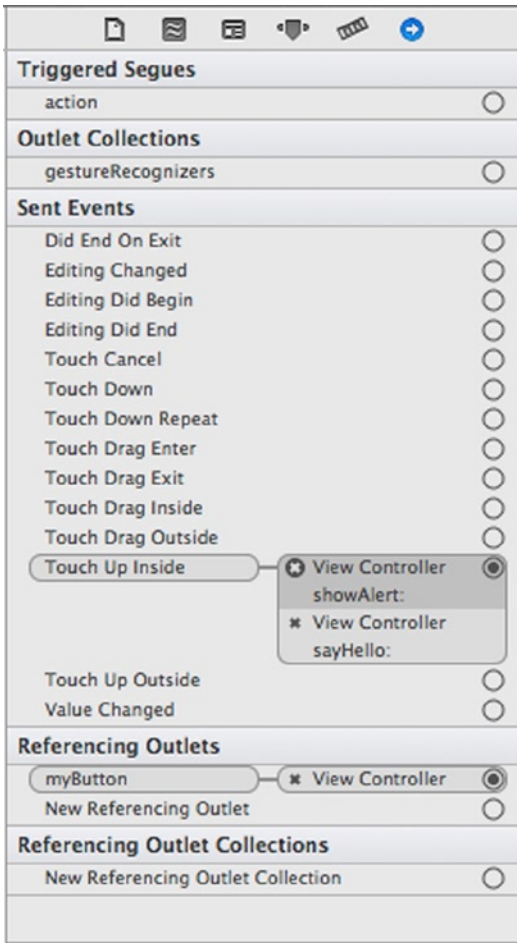

Figure 1-23. *A button with two different action methods (showAlert: and sayHello:) connected to the same event*

Recipe 1-6: Creating a Class

A common task in iOS programming is to create new classes. Whether your aim is to subclass an existing class or create a new domain-model class to hold your data, you can use the Objective-C class template to generate the necessary files.

In this recipe, we'll show you how to create and add a new class to your project. If you don't have a suitable project to try this in, create a new single view application.

For a new class, go to the project navigator and select the group folder in which you want to store the files for your new class. Normally, this is the group folder with the same name as your project, but as your application grows you may want to organize your files into subfolders.

Go to the main menu and select File ➤ New ➤ File (or simply use the keyboard shortcut command [⌘]+ N). Then select the Objective-C class template in the iOS Cocoa Touch section (see Figure 1-24).

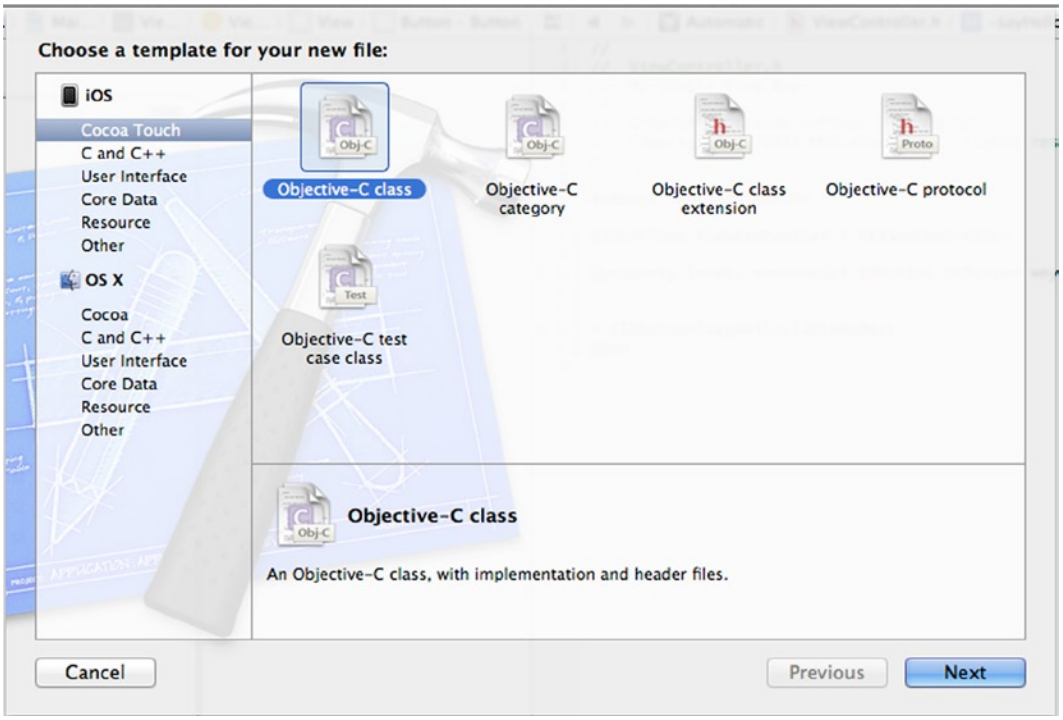

***Figure 1-24.** Using the Objective-C class template to create a new class*

On the following page, enter "MyClass" in the class field and choose "NSObject" from the subclass of the drop-down list. The convention in Objective-C is to name classes using the PascalCase style.

Here you are creating a new class called "MyClass," and you are making "NSObject" the parent class. An NSObject class is the best selection for a general class, but you might want to select a parent class of a different type, according to your needs. For example, if you want to create a new view controller, the parent class would be UIViewController.

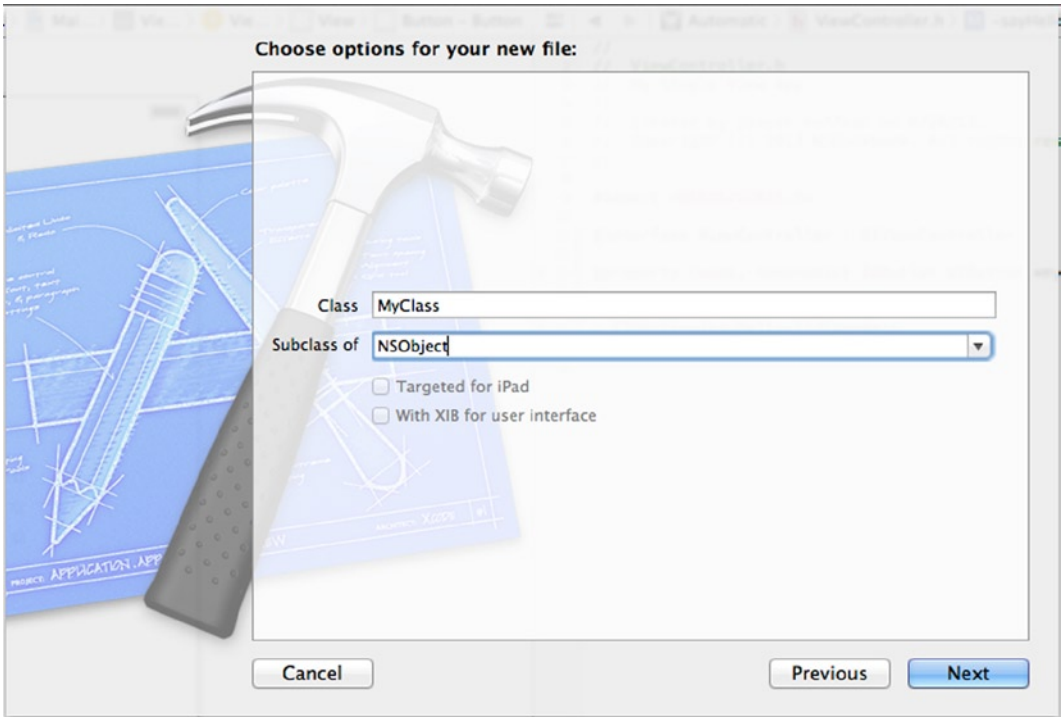

Figure 1-25. *Configuring a new class*

> **Note** Depending on which class you select as the parent, you might or might not set additional settings such as Targeted for iPad, or With XIB for user interface. These options are active if you subclass a view controller of some kind.

The next step is to select a physical location on the hard disk and a logical location within your project for your new class; that is, the file folder and the group folder. In this step (see Figure 1-26), you can also decide whether your class should be included in the target (the executable file). This is usually what you want, but there might be situations when you want to exclude files, such as when you have more than one target, like a unit-test target.

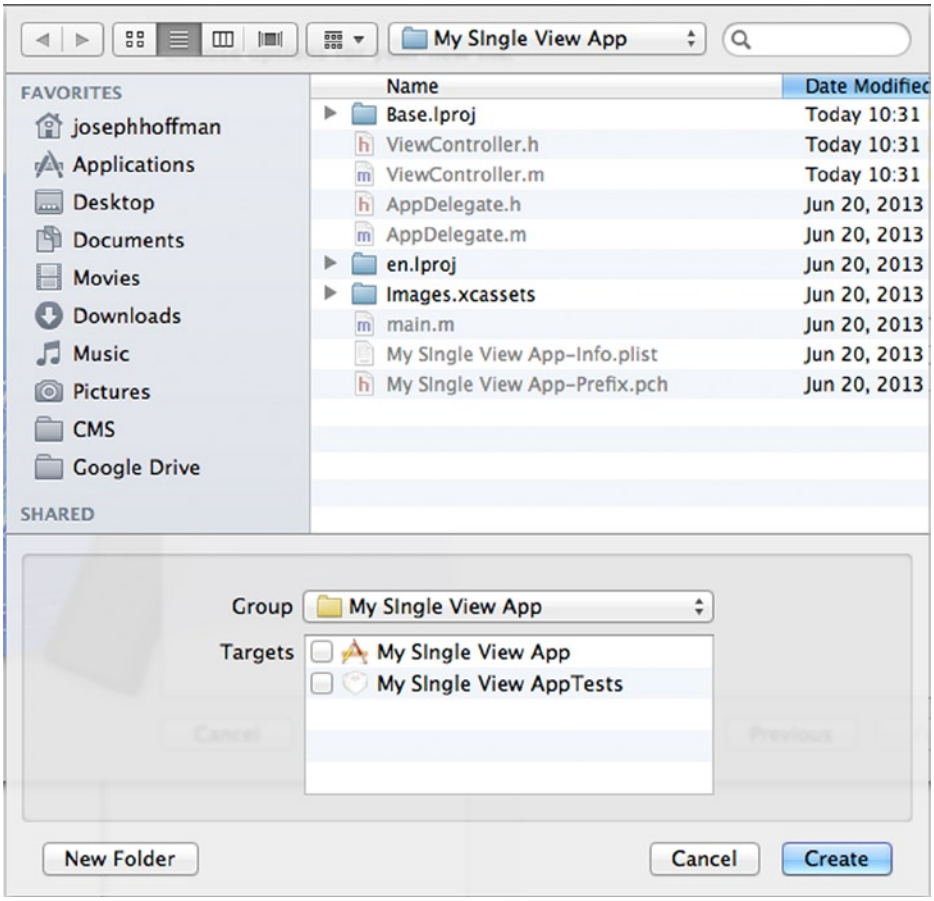

***Figure 1-26.** Selecting the physical (file folder) and logical (group folder) places for a class*

Most of the time you can just accept the default values for the locations, so go ahead and click "Create." Xcode then generates two new files for your project: `MyClass.h` and `MyClass.m`. They contain the code of an empty class, as in the header and implementation files shown in Listing 1-5 and Listing 1-6.

***Listing 1-5.** A new class header file*

```
//
//  MyClass.h
//  My App
//

#import <Foundation/Foundation.h>

@interface MyClass : NSObject

@end
```