

IOS PROGRAMMING

THE BIG NERD RANCH GUIDE

CHRISTIAN KEUR, AARON HILLEGASS & JOE CONWAY



iOS Programming: The Big Nerd Ranch Guide

by Christian Keur, Aaron Hillegass and Joe Conway

Copyright © 2014 Big Nerd Ranch, LLC

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, contact

Big Nerd Ranch, LLC
1989 College Ave NE
Atlanta, GA 30317
(404) 478-9005
<http://www.bignerdranch.com/>
book-comments@bignerdranch.com

The 10-gallon hat with propeller logo is a trademark of Big Nerd Ranch, LLC

Exclusive worldwide distribution of the English edition of this book by

Pearson Technology Group
800 East 96th Street
Indianapolis, IN 46240 USA
<http://www.informit.com>

The authors and publisher have taken care in writing and printing this book but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

App Store, Apple, Cocoa, Cocoa Touch, Finder, Instruments, iCloud, iPad, iPhone, iPod, iPod touch, iTunes, Keychain, Mac, Mac OS, Multi-Touch, Objective-C, OS X, Quartz, Retina, Safari, and Xcode are trademarks of Apple, Inc., registered in the U.S. and other countries.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

ISBN-10 0321942051
ISBN-13 978-0321942050

Fourth edition, second printing, March 2014

Acknowledgments

While our names appear on the cover, many people helped make this book a reality. We would like to take this chance to thank them.

- The other instructors who teach the iOS Bootcamp fed us with a never-ending stream of suggestions and corrections. They are Brian Hardy, Mikey Ward, Owen Mathews, Juan Pablo Claude, Rod Strougo, Jonathan Blocksom, Fernando Rodriguez, Jay Campbell, Matt Mathias, Scott Ritchie, Pouria Almassi, Step Christopher, TJ Usiyan, Bolot Kerimbaev, and Mark Dalrymple. These instructors were often aided by their students in finding book errata, so many thanks are due to all the students who attend the iOS Bootcamp.
- Our technical reviewers, Chris Morris, Jawwad Ahmad, and Véronique Brossier, helped us find and fix flaws.
- Our tireless editor, Susan Loper, took our distracted mumblings and made them into readable prose.
- Elizabeth Holaday jumped in to provide copy-editing and proofing.
- Ellie Volckhausen designed the cover. (The photo is of the bottom bracket of a bicycle frame.)
- Chris Loper at IntelligentEnglish.com designed and produced the print book and the EPUB and Kindle versions.
- The amazing team at Pearson Technology Group patiently guided us through the business end of book publishing.

The final and most important thanks goes to our students whose questions inspired us to write this book and whose frustrations inspired us to make it clear and comprehensible.

Table of Contents

Introduction	xi
Prerequisites	xi
What has Changed in the Fourth Edition?	xi
Our Teaching Philosophy	xi
How to Use This Book	xii
How This Book is Organized	xiii
Style Choices	xiv
Typographical Conventions	xiv
Necessary Hardware and Software	xiv
1. A Simple iOS Application	1
Creating an Xcode Project	1
Model-View-Controller	4
Designing Quiz	4
Building an Interface	5
Creating view objects	6
Configuring view objects	8
Making connections	10
Creating Model Objects	14
Implementing action methods	14
Running on the Simulator	15
Deploying an Application	15
Application Icons	16
Launch Images	18
2. Views and the View Hierarchy	19
View Basics	20
The View Hierarchy	20
Subclassing UIView	22
Views and frames	23
Custom Drawing in drawRect(_:)	27
Drawing a single circle	28
UIBezierPath	28
Using the developer documentation	29
Drawing concentric circles	33
Redrawing Views	35
The Run Loop	36
More Developer Documentation	37
Bronze Challenge: Draw an Image	37
3. UIScrollView and Delegation	39
Using UIScrollView	39
Delegation	43
Protocols	45
Zooming	46
For the More Curious: Panning and paging	47
4. View Controllers	49
Subclassing UIViewController	49
The view of a view controller	50
Creating a view programmatically	50
Setting the root view controller	51
Another UIViewController	52
Creating a view in Interface Builder	54
UITabBarController	59
Tab bar items	61
UIViewController Initializers	63
Adding a Local Notification	63
Loaded and Appearing Views	64

Accessing subviews	65
Interacting with View Controllers and Their Views	66
Bronze Challenge: Another Tab	66
Silver Challenge: Controller Logic	66
For the More Curious: Retina Display	66
5. UITableView and UITableViewController	69
Beginning the Homeowner Application	69
UITableViewController	71
Subclassing UITableViewController	71
UITableView's Data Source	73
Creating ItemStore	73
Giving the controller access to the store	74
Implementing data source methods	74
UITableViewCell	75
Creating and retrieving UITableViewCells	76
Reusing UITableViewCells	77
Subclassing UITableViewCell	79
Creating ItemCell	80
Exposing the properties of ItemCell	81
Using ItemCell	81
Bronze Challenge: Sections	82
Silver Challenge: Constant Rows	82
Gold Challenge: Customizing the Table	82
6. Editing UITableView	83
Editing Mode	83
Adding Rows	88
Deleting Rows	89
Moving Rows	90
Bronze Challenge: Renaming the Delete Button	91
Silver Challenge: Preventing Reordering	91
Gold Challenge: Really Preventing Reordering	91
7. UINavigationController	93
UINavigationController	94
An Additional UIViewController	98
Navigating with UINavigationController	102
Pushing view controllers	102
Passing data between view controllers	103
Appearing and disappearing views	104
UINavigationBar	105
Bronze Challenge: Displaying a Number Pad	108
Silver Challenge: Dismissing a Number Pad	108
Gold Challenge: Pushing More View Controllers	108
8. Camera	109
Displaying Images and UIImageView	109
Adding a camera button	111
Taking Pictures and UIImagePickerController	113
Setting the image picker's sourceType	113
Setting the image picker's delegate	114
Presenting the image picker modally	115
Saving the image	116
Creating ImageStore	117
Dictionary	117
Another Dependency	119
Creating and Using Keys	120
Wrapping up ImageStore	121
Dismissing the Keyboard	122
Bronze Challenge: Editing an Image	123
Silver Challenge: Removing an Image	123

Gold Challenge: Camera Overlay	123
For the More Curious: Navigating Implementation Files	124
// MARK:	125
9. Saving, Loading, and Application States	127
Archiving	127
Application Sandbox	129
Constructing a file path	130
NSKeyedArchiver and NSKeyedUnarchiver	130
NSNotificationCenter	131
Application States and Transitions	133
Writing to the Filesystem with NSData	135
Low-Memory Warnings	137
More on NSNotificationCenter	138
Model-View-Controller-Store Design Pattern	138
Bronze Challenge: PNG	139
For the More Curious: Application State Transitions	139
For the More Curious: Reading and Writing to the Filesystem	140
For the More Curious: The Application Bundle	141
10. Introduction to Auto Layout	145
Universalizing Homepwner	145
The Auto Layout System	147
Alignment rectangle and layout attributes	147
Constraints	148
Adding Constraints in Interface Builder	150
Adding more constraints	153
Adding even more constraints	157
Priorities	158
Debugging Constraints	158
Ambiguous layout	159
Unsatisfiable constraints	163
Misplaced views	163
Bronze Challenge: Practice Makes Perfect	165
Silver Challenge: Universalize Quiz	165
11. Auto Layout: Programmatic Constraints	167
Visual Format Language	168
Creating Constraints	168
Activating Constraints	169
Intrinsic Content Size	170
The Other Way	172
For the More Curious: NSAutoresizingMaskLayoutConstraint	173
12. Size Classes	175
Updating the Image View	175
What is a Size Class	177
Updating the Interface	178
UIAlertController	182
13. Autorotation, Modal View Controllers, and Popover Controllers	185
Autorotation	185
Modal View Controllers	187
Dismissing modal view controllers	188
Modal view controller styles	190
Modal view controller transitions	191
UIPopoverController	191
Gold Challenge: Popover Appearance	193
For the More Curious: View Controller Relationships	193
Parent-child relationships	193
Presenting-presenter relationships	194
Inter-family relationships	194
14. Core Data	197

Object-Relational Mapping	197
Moving Homepwner to Core Data	197
The model file	197
NSManagedObject and subclasses	200
Updating ItemStore	202
Adding AssetTypes to Homepwner	206
Faults	210
Trade-offs of Persistence Mechanisms	211
Bronze Challenge: Assets on the iPad	211
Silver Challenge: New Asset Types	212
Gold Challenge: Showing Assets of a Type	212
15. Localization	213
Internationalization	213
Formatters	214
Base Internationalization	216
Pseudolanguages	217
Localization	220
NSLocalizedString and Strings Tables	222
Manually Updating the Strings Table	225
Bronze Challenge: Another Localization	226
For the More Curious: NSBundle's Role in Internationalization	226
For the More Curious: Importing and Exporting as XLIFF	226
16. Controlling Animations	229
Basic Animations	229
Timing functions	231
Keyframe Animations	232
Animation Completion	234
Bronze Challenge: Spring Animations	234
Silver Challenge: Improved Quiz	234
17. UIStoryboard	235
Creating a Storyboard	235
UITableViewController Controllers in Storyboards	238
Segues	239
Enabling Color Changes	243
Passing Data Around	244
18. Web Services and WKWebView	249
Web Services	250
Starting the Nerdfeed application	250
NSURL, NSURLRequest, NSURLSession, and NSURLSessionTask	251
Formatting URLs and requests	252
Working with NSURLSession	252
JSON data	254
Parsing JSON data	254
The main thread	256
WKWebView	257
Credentials	258
Silver Challenge: More WKWebView	259
For the More Curious: The Request Body	260
19. UISplitViewController	263
Splitting Up Nerdfeed	263
Displaying the Master View Controller in Portrait Mode	267
20. Touch Events and UIResponder	269
Touch Events	269
Creating the TouchTracker Application	270
Drawing with DrawView	271
Turning Touches into Lines	272
Handling multiple touches	273
Bronze Challenge: Saving and Loading	276

Silver Challenge: Colors	276
Gold Challenge: Circles	276
For the More Curious: The Responder Chain	276
For the More Curious: UIControl	277
21. UIGestureRecognizer and UIMenuController	279
UIGestureRecognizer Subclasses	279
Detecting Taps with UITapGestureRecognizer	280
Multiple Gesture Recognizers	281
UIMenuController	283
UILongPressGestureRecognizer	284
UIPanGestureRecognizer and Simultaneous Recognizers	285
For the More Curious: UIMenuController and UIResponderStandardEditActions	287
For the More Curious: More on UIGestureRecognizer	287
Silver Challenge: Mysterious Lines	288
Gold Challenge: Speed and Size	288
Mega-Gold Challenge: Colors	288
22. Debugging Tools	289
Gauges	289
Instruments	291
Allocations instrument	292
Time Profiler instrument	296
Leaks instrument	298
Projects, Targets, and Build Settings	299
Build configurations	301
Changing a build setting	302
23. Afterword	305
What to do Next	305
Shameless Plugs	305
Index	307

Introduction

As an aspiring iOS developer, you face three basic hurdles:

- *You must learn the Objective-C language.* Objective-C is a small and simple extension to the C language. After the first four chapters of this book, you will have a working knowledge of Objective-C.
- *You must master the big ideas.* These include things like memory management techniques, delegation, archiving, and the proper use of view controllers. The big ideas take a few days to understand. When you reach the halfway point of this book, you will understand these big ideas.
- *You must master the frameworks.* The eventual goal is to know how to use every method of every class in every framework in iOS. This is a project for a lifetime: there are over 3000 methods and more than 200 classes available in iOS. To make things even worse, Apple adds new classes and new methods with every release of iOS. In this book, you will be introduced to each of the subsystems that make up the iOS SDK, but we will not study each one deeply. Instead, our goal is to get you to the point where you can search and understand Apple's reference documentation.

We have used this material many times at our iOS Development Bootcamp at Big Nerd Ranch. It is well-tested and has helped hundreds of people become iOS application developers. We sincerely hope that it proves useful to you.

Prerequisites

This book assumes that you are already motivated to learn to write iOS apps. We will not spend any time convincing you that the iPhone, the iPad, and the iPod touch are compelling pieces of technology.

We also assume that you know the C programming language and something about object-oriented programming. If this is not true, you should probably start with an introductory book on C and Objective-C, such as *Objective-C Programming: The Big Nerd Ranch Guide*.

What has Changed in the Fourth Edition?

This edition assumes that the reader is using Xcode 5 and running applications on an iOS 7 device or simulator.

We have adopted a more modern style of Objective-C in this edition. We use properties, dot notation, auto-synthesized instance variables, the new literals, and subscripting extensively. We also use blocks more.

Apple continues to evolve iOS, and we have eagerly added coverage of block-based animations, Auto Layout, and `NSURLSession` to the book.

Besides these obvious changes, we made thousands of tiny improvements that were inspired by questions from our readers and our students. Every chapter of this book is just a little better than the corresponding chapter from the third edition.

Our Teaching Philosophy

This book will teach you the essential concepts of iOS programming. At the same time, you will type in a lot of code and build a bunch of applications. By the end of the book, you will have knowledge *and* experience. However, all the knowledge should not (and, in this book, will not) come first. That is sort of the traditional way we have all come to know and hate. Instead, we take a learn-while-doing approach. Development concepts and actual coding go together.

Here is what we have learned over the years of teaching iOS programming:

- We have learned what ideas people must grasp to get started programming, and we focus on that subset.
- We have learned that people learn best when these concepts are introduced *as they are needed*.
- We have learned that programming knowledge and experience grow best when they grow together.

- We have learned that “going through the motions” is much more important than it sounds. Many times we will ask you to start typing in code before you understand it. We realize that you may feel like a trained monkey typing in a bunch of code that you do not fully grasp. But the best way to learn coding is to find and fix your typos. Far from being a drag, this basic debugging is where you really learn the ins and outs of the code. That is why we encourage you to type in the code yourself. You could just download it, but copying and pasting is not programming. We want better for you and your skills.

What does this mean for you, the reader? To learn this way takes some trust. And we appreciate yours. It also takes patience. As we lead you through these chapters, we will try to keep you comfortable and tell you what is happening. However, there will be times when you will have to take our word for it. (If you think this will bug you, keep reading – we have some ideas that might help.) Do not get discouraged if you run across a concept that you do not understand right away. Remember that we are intentionally *not* providing all the knowledge you will ever need all at once. If a concept seems unclear, we will likely discuss it in more detail later when it becomes necessary. And some things that are not clear at the beginning will suddenly make sense when you implement them the first (or the twelfth) time.

People learn differently. It is possible that you will love how we hand out concepts on an as-needed basis. It is also possible that you will find it frustrating. In case of the latter, here are some options:

- Take a deep breath and wait it out. We will get there, and so will you.
- Check the index. We will let it slide if you look ahead and read through a more advanced discussion that occurs later in the book.
- Check the online Apple documentation. This is an essential developer tool, and you will want plenty of practice using it. Consult it early and often.
- If Objective-C or object-oriented programming concepts are giving you a hard time (or if you think they will), you might consider backing up and reading our *Objective-C Programming: The Big Nerd Ranch Guide*.

How to Use This Book

This book is based on the class we teach at Big Nerd Ranch. As such, it was designed to be consumed in a certain manner.

Set yourself a reasonable goal, like “I will do one chapter every day.” When you sit down to attack a chapter, find a quiet place where you will not be interrupted for at least an hour. Shut down your email, your Twitter client, and your chat program. This is not a time for multi-tasking; you will need to concentrate.

Do the actual programming. You can read through a chapter first, if you like. But the real learning comes when you sit down and code as you go. You will not really understand the idea until you have written a program that uses it and, perhaps more importantly, debugged that program.

A couple of the exercises require supporting files. For example, in the first chapter you will need an icon for your Quiz application, and we have one for you. You can download the resources and solutions to the exercises from <http://www.bignerdranch.com/solutions/iOSProgramming4ed.zip>.

There are two types of learning. When you learn about the Peloponnesian War, you are simply adding details to a scaffolding of ideas that you already understand. This is what we will call “Easy Learning.” Yes, learning about the Peloponnesian War can take a long time, but you are seldom flummoxed by it. Learning iOS programming, on the other hand, is “Hard Learning,” and you may find yourself quite baffled at times, especially in the first few days. In writing this book, we have tried to create an experience that will ease you over the bumps in the learning curve. Here are two things you can do to make the journey easier:

- Find someone who already knows how to write iOS applications and will answer your questions. In particular, getting your application onto the device the first time is usually very frustrating if you are doing it without the help of an experienced developer.
- Get enough sleep. Sleepy people do not remember what they have learned.

How This Book is Organized

In this book, each chapter addresses one or more ideas of iOS development through discussion and hands-on practice. For more coding practice, most chapters include challenge exercises. We encourage you to take on at least some of these. They are excellent for firming up the concepts introduced in the chapter and making you a more confident iOS programmer. Finally, most chapters conclude with one or two “For the More Curious” sections that explain certain consequences of the concepts that were introduced earlier.

Chapter 1 introduces you to iOS programming as you build and deploy a tiny application. You will get your feet wet with Xcode and the iOS simulator along with all the steps for creating projects and files. The chapter includes a discussion of Model-View-Controller and how it relates to iOS development.

Chapters 2 and 3 provide an overview of Objective-C and memory management. Although you will not create an iOS application in these two chapters, you will build and debug a tool called `RandomItems` to ground you in these concepts.

In Chapters 4 and 5, you will begin focusing on the iOS user interface as you learn about views and the view hierarchy and create an application called `Hypnosister`.

Chapters 6 and 7 introduce view controllers for managing user interfaces with the `HypoNerd` application. You will get practice working with views and view controllers as well as navigating between screens using a tab bar. You will also get plenty of experience with the important design pattern of delegation as well as working with protocols, the debugger, and setting up local notifications.

Chapter 8 introduces the largest application in the book – `Homepwner`. (By the way, “`Homepwner`” is not a typo; you can find the definition of “`pwn`” at www.urbandictionary.com.) This application keeps a record of your possessions in case of fire or other catastrophe. `Homepwner` will take fourteen chapters to complete.

In Chapters 8, 9, and 19, you will build experience with tables. You will learn about table views, their view controllers, and their data sources. You will learn how to display data in a table, how to allow the user to edit the table, and how to improve the interface.

Chapter 10 builds on the navigation experience gained in Chapter 6. You will learn how to use `UINavigationController` and you will give `Homepwner` a drill-down interface and a navigation bar.

In Chapter 11, you will learn how to take pictures with the camera and how to display and store images in `Homepwner`. You will use `NSDictionary` and `UIImagePickerController`.

In Chapters 12 and 13, you will set `Homepwner` aside for a bit to create a drawing application named `TouchTracker` to learn about touch events. You will see how to add multi-touch capability and how to use `UIGestureRecognizer` to respond to particular gestures. You will also get experience with the first responder and responder chain concepts and more practice with `NSDictionary`.

In Chapter 14, you will learn how to use debug gauges, Instruments, and the static analyzer to optimize the performance of `TouchTracker`.

In Chapters 15 and 16, you will make `Homepwner` a universal application – an application that runs natively on both the iPhone and the iPad. You will also work with Auto Layout to build an interface that will appear correctly on any screen size.

In Chapter 17, you will learn about handling rotation and using `UIPopoverController` for the iPad and modal view controllers.

Chapter 18 delves into ways to save and load data. In particular, you will archive data in the `Homepwner` application.

In Chapter 20, you will update `Homepwner` to use Dynamic Type to support different font sizes that a user may prefer.

Chapter 21 takes another break from `Homepwner` and introduces web services as you create the `Nerdfeed` application. This application fetches and parses an RSS feed from a server using `NSURLConnection` and `NSXMLParser`. `Nerdfeed` will also display a web page in a `UIWebView`.

In Chapter 22, you will learn about `UISplitViewController` and add a split view user interface to `Nerdfeed` to take advantage of the iPad’s larger screen size.

Chapter 23 returns to the Homepwner application with an introduction to Core Data. You will change Homepwner to store and load its data using an **NSManagedObjectContext**.

In Chapter 24, you will add state restoration to Homepwner to let users return to the application right where they left off – no matter how long they are away.

Chapter 25 introduces the concepts and techniques of internationalization and localization. You will learn about **NSLocale**, strings tables, and **NSBundle** as you localize parts of Homepwner.

In Chapter 26, you will use **NSUserDefaults** to save user preferences in a persistent manner. This chapter will complete the Homepwner application.

Chapter 27 introduces the Core Animation framework with a brief return to the HypnoNerd application to implement animations.

Chapter 28 introduces building applications using storyboards. You will piece together an application using **UIStoryboard** and learn more about the pros and cons of using storyboards.

Style Choices

This book contains a lot of code. We have attempted to make that code and the designs behind it exemplary. We have done our best to follow the idioms of the community, but at times we have wandered from what you might see in Apple’s sample code or code you might find in other books. You may not understand these points now, but it is best that we spell them out before you commit to reading this book:

- We typically create view controllers programmatically. Some programmers will instantiate view controllers inside XIB files or in a storyboard. We will discuss storyboards and demonstrate their use a little, but, in reality, we seldom use them in our projects at Big Nerd Ranch.
- We will nearly always start a project with the simplest template project: the empty application. When your app works, you will know it is because of your efforts – not because that behavior was built into the template.

Typographical Conventions

To make this book easier to read, certain items appear in certain fonts. Class names, method names, and function names appear in a bold, fixed-width font. Class names start with capital letters and method names start with lowercase letters. In this book, method and function names are formatted the same for simplicity’s sake. For example, “In the **loadView** method of the **BNRReveViewController** class, use the **NSLog** function to print the value to the console.”

Variables, constants, and types appear in a fixed-width font but are not bold. So you will see, “The variable **fido** will be of type **float**. Initialize it to **M_PI**.”

Applications and menu choices appear in the Mac system font. For example, “Open Xcode and select New Project... from the File menu.”

All code blocks are in a fixed-width font. Code that you need to type in is always bold. For example, in the following code, you would type in everything but the first and last lines. (Those lines are already in the code and appear here to let you know where to add the new stuff.)

```
@interface BNRQuizViewController ()  
  
@property (nonatomic, weak) IBOutlet UILabel *questionLabel;  
@property (nonatomic, weak) IBOutlet UILabel *answerLabel;  
  
@end
```

Necessary Hardware and Software

You can only develop iOS apps on an Intel Mac. You will need to download Apple’s iOS SDK, which includes Xcode (Apple’s Integrated Development Environment), the iOS simulator, and other development tools.

You should join Apple’s iOS Developer Program, which costs \$99/year, for three reasons:

- Downloading the latest developer tools is free for members.
- Only signed apps will run on a device, and only members can sign apps. If you want to test your app on your device, you will need to join.
- You cannot put an app in the store until you are a member.

If you are going to take the time to work through this entire book, membership in the iOS Developer Program is, without question, worth the cost. Go to <http://developer.apple.com/programs/ios/> to join.

What about iOS devices? Most of the applications you will develop in the first half of the book are for the iPhone, but you will be able to run them on an iPad. On the iPad screen, iPhone applications appear in an iPhone-sized window. Not a compelling use of the iPad, but that is okay when you are starting with iOS. In these first chapters, you will be focused on learning the fundamentals of the iOS SDK, and these are the same across iOS devices. Later in the book, we will look at some iPad-only options and how to make applications run natively on both iOS device families.

Excited yet? Good. Let's get started.

1

A Simple iOS Application

In this chapter, you are going to write an iOS application named Quiz. This application will show a question and then reveal the answer when the user taps a button. Tapping another button will show the user a new question (Figure 1.1).

Figure 1.1 Your first application: Quiz



When you are writing an iOS application, you must answer two basic questions:

- How do I get my objects created and configured properly? (Example: “I want a button here entitled Show Question.”)
- How do I deal with user interaction? (Example: “When the user taps the button, I want this piece of code to be executed.”)

Most of this book is dedicated to answering these questions.

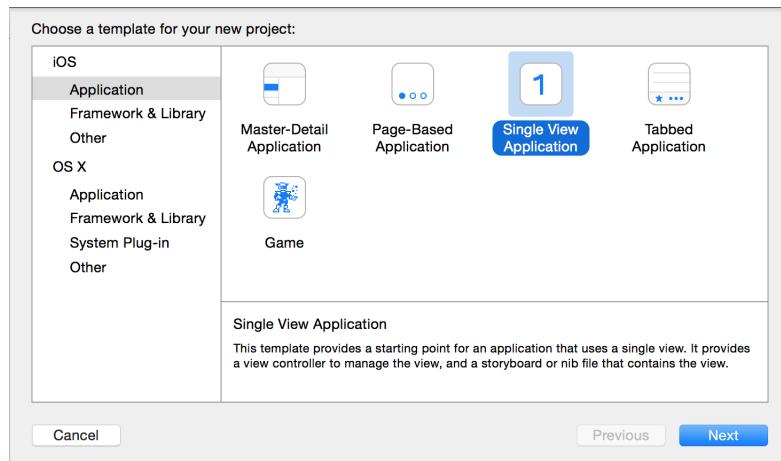
As you go through this first chapter, you will probably not understand everything that you are doing, and you may feel ridiculous just going through the motions. But going through the motions is enough for now. Mimicry is a powerful form of learning; it is how you learned to speak, and it is how you will start iOS programming. As you become more capable, you will experiment and challenge yourself to do creative things on the platform. For now, just do what we show you. The details will be explained in later chapters.

Creating an Xcode Project

Open Xcode and, from the File menu, select New → Project...

A new workspace window will appear, and a sheet will slide down from its toolbar. On the lefthand side, find the iOS section and select Application (Figure 1.2). You will be offered several application templates to choose from. Select Single View Application.

Figure 1.2 Creating a project

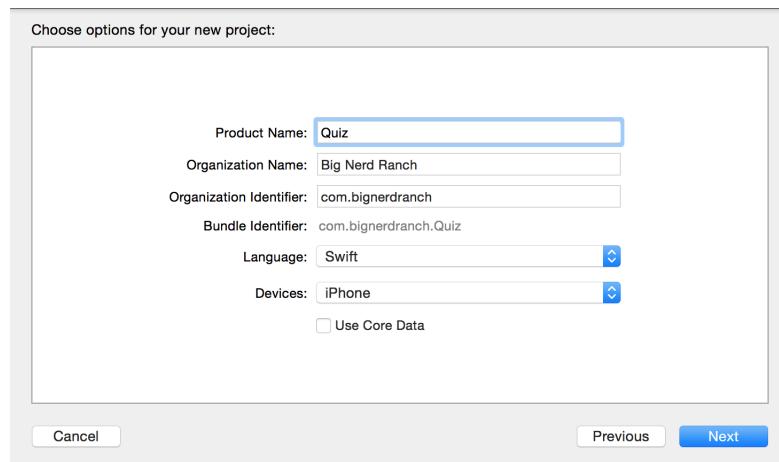


This book was created for Xcode 6.1. The names of these templates may change with new Xcode releases. If you do not see a Single View Application template, use the simplest-sounding template. Or visit the Big Nerd Ranch forum for this book at forums.bignerdranch.com for help working with newer versions of Xcode.

Click Next and, in the next sheet, enter Quiz for the Product Name (Figure 1.3). The organization name and company identifier are required to continue. You can use Big Nerd Ranch and com.bignerdranch. Or use your company name and com.yourcompanynamehere.

Choose Swift from the language pop-up menu and iPhone from the pop-up menu labeled Devices. Make sure that the Use Core Data checkbox is unchecked.

Figure 1.3 Configuring a new project

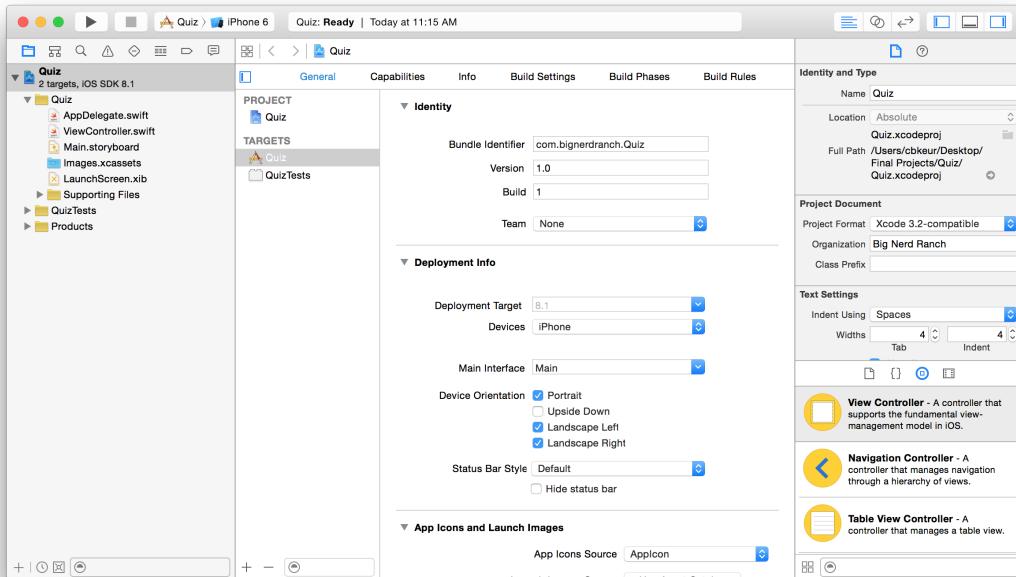


You are creating Quiz as an iPhone application, but it will run on an iPad. It will not look quite right on the iPad's screen, but that is okay for now. For the first part of this book, you will stick to iPhone applications and focus on learning the fundamentals of the iOS SDK, which are the same across devices. Later, you will see some options for iPad-only applications as well as how to make applications run natively on both types of devices.

Click Next and, in the final sheet, save the project in the directory where you plan to store the exercises in this book. Click Create to create the Quiz project.

Once the project is created, it will open in the Xcode workspace window (Figure 1.4).

Figure 1.4 Xcode workspace window

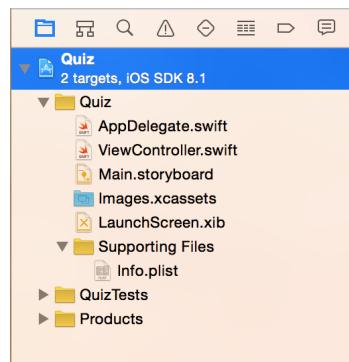


Take a look at the lefthand side of the workspace window. This area is called the *navigator area*, and it displays different *navigators* – tools that show you different parts of your project. You can choose which navigator to use by selecting one of the icons in the navigator selector, which is the bar just above the navigator area.

The navigator currently open is the *project navigator*. The project navigator shows you the files that make up your project (Figure 1.5). You can select a file to open it in the *editor area* to the right of the navigator area.

The files in the project navigator can be grouped into folders to help you organize your project. A few groups have been created by the template for you; you can rename them whatever you want or add new ones. The groups are purely for the organization of files and do not correlate to the filesystem in any way.

Figure 1.5 Quiz application's files in the project navigator



In the project navigator, find the files named `AppDelegate.swift` and `ViewController.swift`. Each of these are the files that define a *class* named `AppDelegate` and `ViewController`, respectively. The Single View Application template created these classes for you.

A class describes a kind of *object*. iOS development is object-oriented, and an iOS application consists primarily of a set of *objects* working together. When the Quiz application is launched, an object of the `AppDelegate` kind

will be created and object of the **ViewController** kind will be created. We refer to these objects as an *instance* of their class.

You will learn much more about how classes and objects work in ????. Right now, you are going to move on to some application design theory and then dive into development.

Model-View-Controller

Model-View-Controller, or MVC, is a design pattern used in iOS development. In MVC, every object is either a model object, a view object, or a controller object.

- *View objects* are visible to the user. Examples of view objects are buttons, text fields, and sliders. View objects make up an application’s user interface. In Quiz, the labels showing the question and answer and the buttons beneath them are view objects.
- *Model objects* hold data and know nothing about the user interface. In Quiz, the model objects will be two ordered lists of strings: one for questions and another for answers.

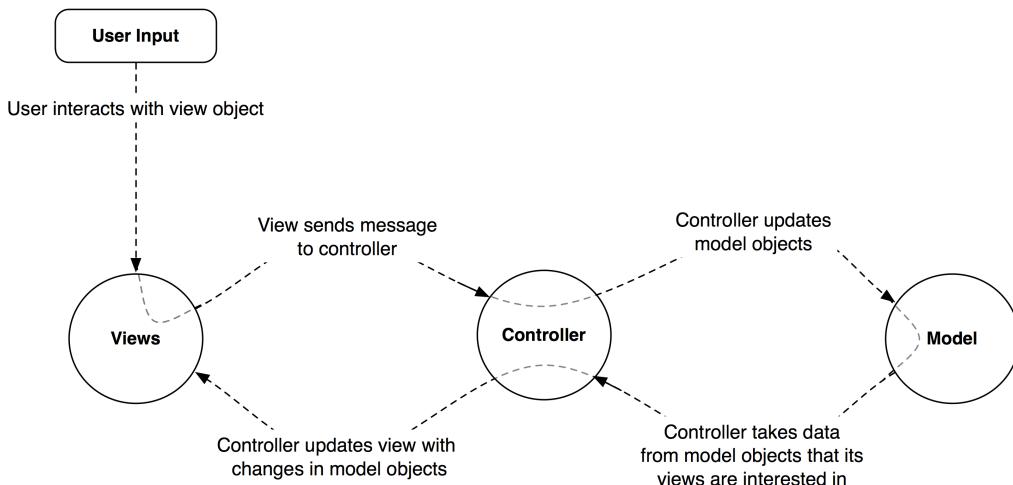
Usually, the model objects are modeling real things from the world of the user. For example, when you write an app for an insurance company, you will almost certainly end up with a custom model class called **InsurancePolicy**.

- *Controller objects* are the managers of an application. Controllers configure the views that the user sees and make sure that the view and model objects keep in sync.

In general, controllers typically handle “And then?” questions. For example, when the user selects an item from a list, the controller determines what that user sees next.

Figure 1.6 shows the flow of control in an application in response to user input, such as the user tapping a button.

Figure 1.6 MVC pattern



Notice that the models and views do not talk to each other directly; controllers sit squarely in the middle of everything, receiving messages from some objects and dispatching instructions to others.

Designing Quiz

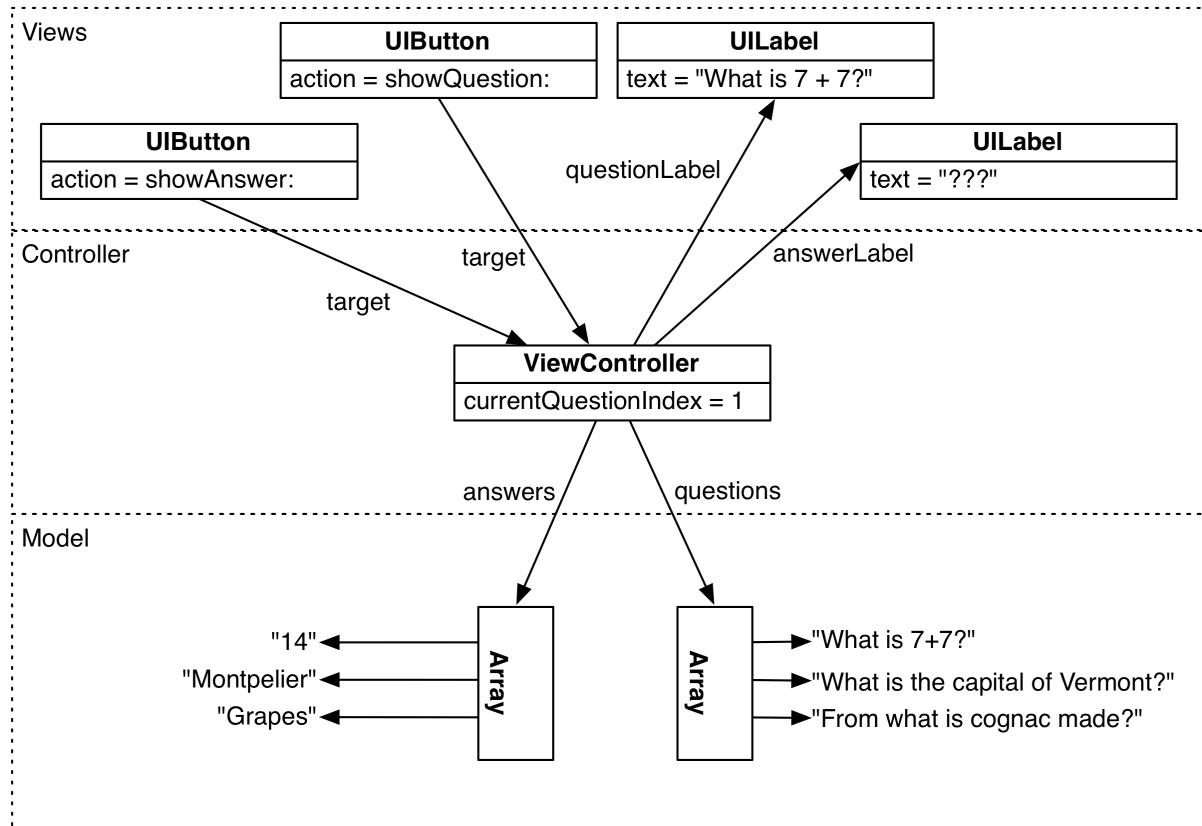
You are going to write the Quiz application using the MVC pattern. Here is a break down of the objects you will be creating and working with:

- 4 view objects: two instances of **UILabel** and two instances of **UIButton**

- 1 controller object: an instance of **ViewController**
- 2 model objects: two instances of **Array**

These objects and their relationships are laid out in the object diagram for Quiz shown in Figure 1.7.

Figure 1.7 Object diagram for Quiz



This diagram is the big picture of how the finished Quiz application will work. For example, when the Show Question button is tapped, it will trigger a *method* in **ViewController**. A method is a lot like a function – a list of instructions to be executed. This method will retrieve a new question from the array of questions and ask the top label to display that question.

It is okay if this object diagram does not make sense yet; it will by the end of the chapter. Refer back to it as you continue to see how the app is taking shape.

You are going to build Quiz in steps, starting with the visual interface for the application.

Building an Interface

Find and select `Main.storyboard` in the project navigator to open it in the editor area.

When Xcode opens a storyboard file, it opens it with **Interface Builder**, a visual tool where you can add and arrange objects to create a graphical user interface.

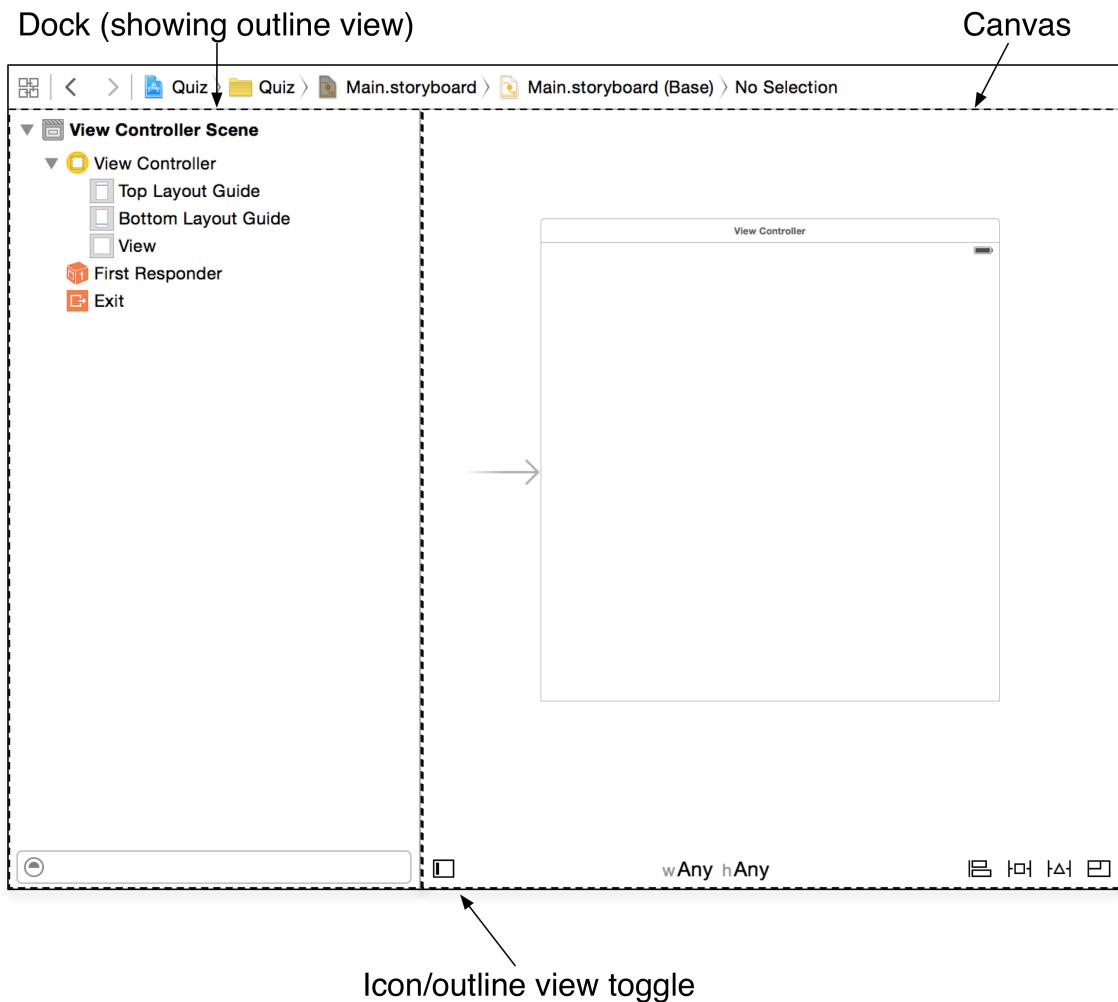
In many GUI builders on other platforms, you describe what you want an application to look like and then press a button to generate a bunch of code. Interface Builder is different. It is an object editor: you create and configure objects, like buttons and labels, and then save them into an archive. The archive is the storyboard file.

Interface Builder divided the editor area into two sections: the *Document Outline* is on the lefthand side and the *canvas* is on the right.

The Document Outline lists the objects in the storyboard file.

If the Document Outline is not visible, click the Show Document Outline button in the bottom lefthand corner of the canvas to reveal the outline view (Figure 1.8).

Figure 1.8 Editing a storyboard file in Interface Builder



The outline view tells you that `Main.storyboard` contains one *scene*: View Controller Scene. A scene represents a view controller with its view. Find the View Controller in the outline view, and then locate its View underneath.

The View object is an instance of `UIView`. This object forms the foundation of your user interface and you can see it displayed on the canvas. The canvas shows how your user interface will appear in the application.

Click on the View object in the document outline to select it in the canvas. You can move the view by dragging it around. Note that moving the view does not change anything about the actual object; it just re-organizes the canvas.

Right now, your interface consists solely of this view object.

Creating view objects

We'll be running this application on the iPhone 6 simulator, which has a 4.7-inch screen size. It would be very useful when laying out our interface if the view on the canvas reflected this size. To do this, you need to get to the attributes inspector in the *utility area*.

The utility area is to the right of the editor area and has two sections: the *inspector* and the *library*. The top section is the inspector, which contains settings for the file or object that is selected in the editor area. The bottom section is the library, which lists items that you can add to a file or project.

At the top of each section is a selector for different types of inspectors and libraries (Figure 1.9).

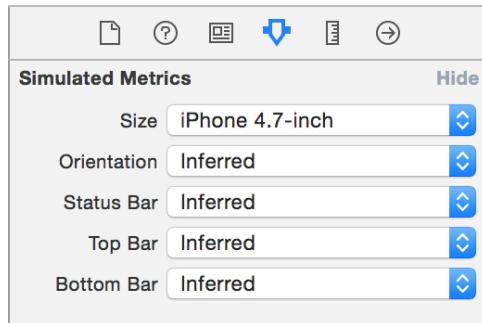
Figure 1.9 Xcode utility area



From the Inspector selector, select the tab to reveal the attributes inspector. Then select the View Controller in the document outline.

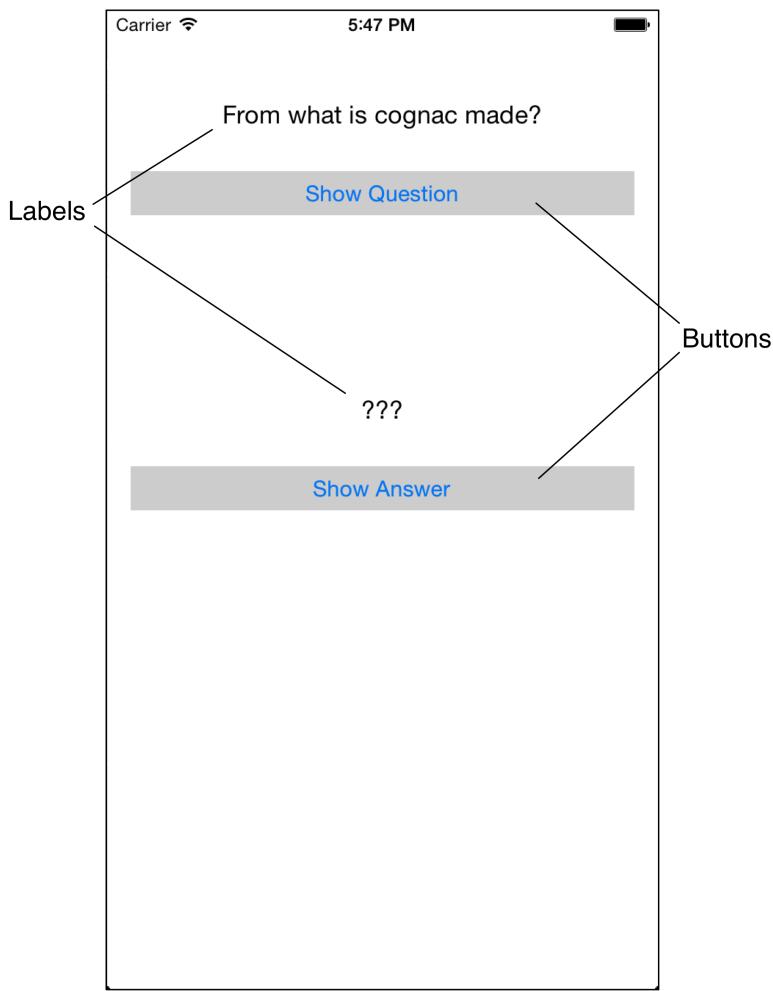
In the Attributes inspector, find the section at the top titled **Simulated Metrics**. Change the **Size** to **iPhone 4.7-inch** (Figure 1.10). Notice that the view on the canvas is no longer square; its size matches that of the iPhone 6.

Figure 1.10 Simulating a View Controller's Size



You need to add four additional view objects for Quiz: two labels and two buttons. To add these view objects, you need to get to the object library in the utility area.

Figure 1.11 Labels and buttons needed



From the library selector, select the tab to reveal the *object library*. The object library contains the objects that you can add to a storyboard file to compose your interface. Find the Label object. (Scroll down the list or use the search bar at the bottom of the library.) Select this object in the library and drag it onto the view object on the canvas. Position this label in the center of the view, near the top. Drag a second label onto the view and position it in the center, closer to the bottom.

Next, find Button in the object library and drag two buttons onto the view. Position one below each label.

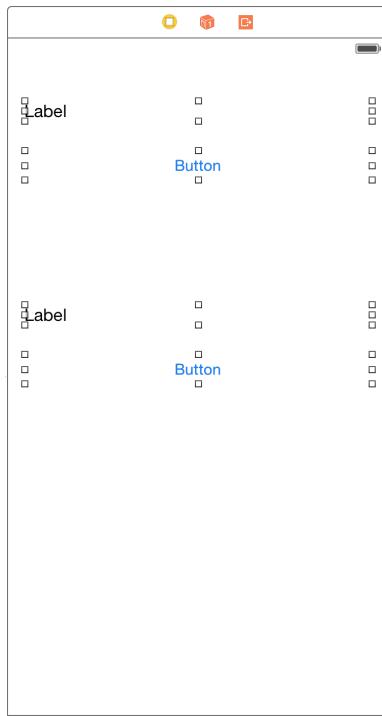
You have now created four view objects and added them to **ViewController**'s user interface. Confirm this in the document outline.

Configuring view objects

Now that you have created the view objects, you can configure their attributes. Some attributes, like size, position, and text, can be changed directly on the canvas. Others must be changed in the attributes inspector.

You can resize an object by selecting it on the canvas or in the outline view and then dragging its corners and edges in the canvas. Resize all four of your view objects to span most of the window.

Figure 1.12 Stretching the labels and buttons



You can edit the title of a button or a label by double-clicking it and typing in new text. Change the top button's title to Show Question and the bottom button's title to Show Answer. Change the bottom label to display ???.

Delete the text in the top label and leave it blank. (Eventually, this label will display the question to the user.) Your interface should look like Figure 1.13.

Figure 1.13 Setting the text on the labels and buttons

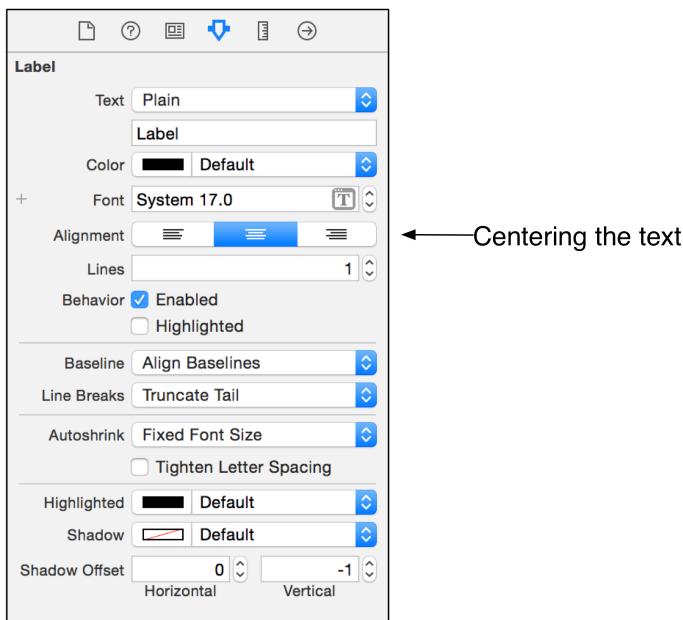


It would be nice if the text in the labels was centered. Setting the text alignment of a label must be done in the attributes inspector.

Select the tab to reveal the attributes inspector. Then select the bottom label on the canvas.

In the attributes inspector, find the segmented control for alignment. Select the centered text option, as shown in Figure 1.14.

Figure 1.14 Centering the label text



Back on the canvas, notice that the ??? is now centered in the bottom label. Select the top label in the canvas and return to the attributes inspector to set its text alignment. (This label has no text to display now, but it will in the running application.)

To inform the user where they are able to tap, you can change the background color of the buttons. Select the Show Question button on the canvas.

In the attributes inspector, scroll down until you see the attributes under the View header. Next to the Background label, click on the color (the white box with a red slash). This will bring up the full color picker. Pick a nice color to go with the button's blue text.

Do the same for the second button, but instead of clicking on the color on the left side, click on the right side which has text and the up and down arrows. This will bring up a list of recently used colors in chronological order as well as some system default colors. Use this to choose the same color for the second button's background color.

Your application's interface now looks like it should. But to begin making it functional, you need to make some connections between these view objects and the **ViewController** that will be running the show.

Making connections

A *connection* lets one object know where another object is in memory so that the two objects can communicate. There are two kinds of connections that you can make in Interface Builder: outlets and actions. An *outlet* points to, or references, an object. An *action* is a method that gets triggered by a button or some other view that the user can interact with, like a slider or a picker.

Let's start by creating outlets that point to the instances of **UILabel**. Time to leave Interface Builder briefly and write some code.

Declaring outlets

In the project navigator, find and select the file named **ViewController.swift**. The editor area will change from Interface Builder to Xcode's code editor.

In **ViewController.swift**, delete any code that the template added between the `class ViewController: UIViewController {` and `}` so that the file looks like this:

```
import UIKit

class ViewController: UIViewController {
}
```

Next, add the following code. Do not worry about understanding it right now; just get it in.

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var questionLabel: UILabel!
    @IBOutlet weak var answerLabel: UILabel!

}
```

Notice the bold type? In this book, code that you need to type in is always bold; the code that is not bold provides context for where to add the new stuff.

In this new code, you declared two properties. You will learn about properties in ????. For now, focus on the first line.

```
@IBOutlet weak var questionLabel: UILabel!
```

This code gives every instance of **ViewController** an outlet named **questionLabel**, which it can use to point to a **UILabel** object. The **@IBOutlet** keyword tells Xcode that you will set this outlet using Interface Builder.

Setting outlets

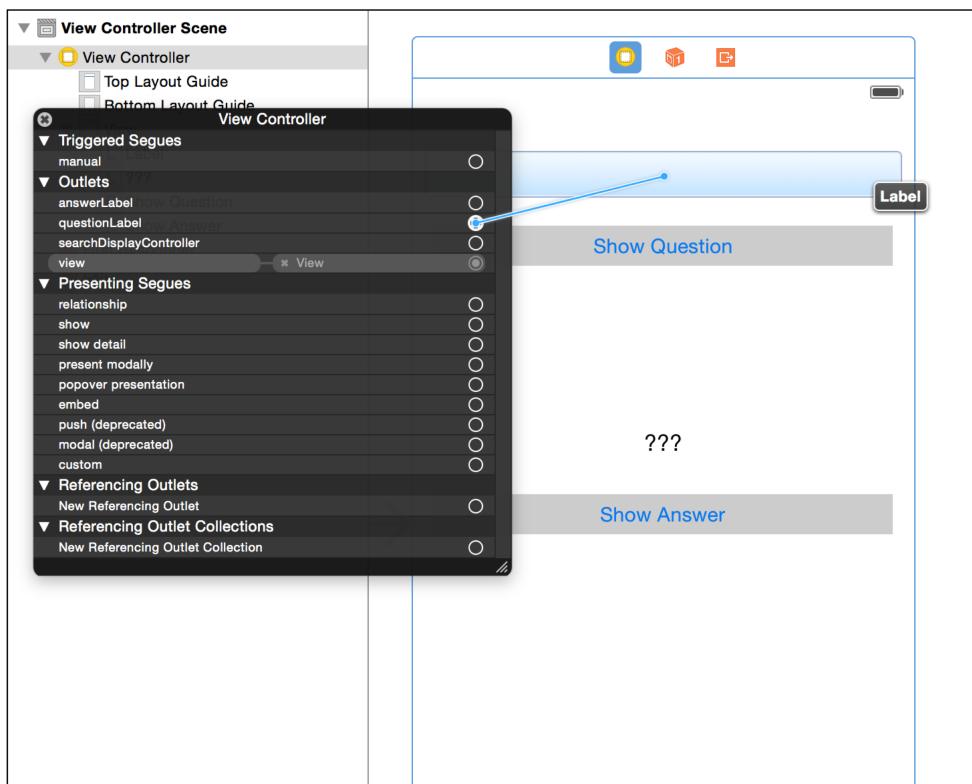
In the project navigator, select **Main.storyboard** to reopen Interface Builder.

You want the **questionLabel** outlet to point to the instance of **UILabel** at the top of the user interface.

In the Document Outline, find the View Controller object under the View Controller Scene.

In the dock, right-click or Control-click on the View Controller to bring up the connections panel (Figure 1.15). Then drag from the circle beside **questionLabel** to the **UILabel**. When the label is highlighted, release the mouse button, and the outlet will be set.

Figure 1.15 Setting **questionLabel**

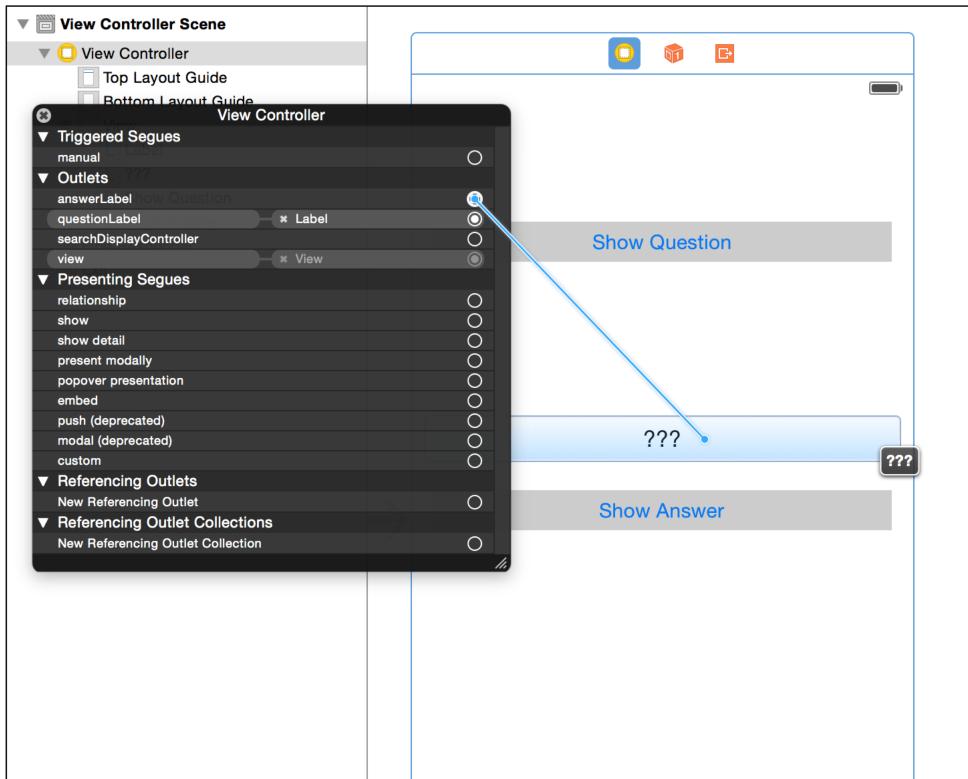


(If you do not see `questionLabel` in the connections panel, double-check your `ViewController.swift` file for typos.)

Now when the storyboard file is loaded, the `ViewController`'s `questionLabel` outlet will automatically point to the instance of `UILabel` at the top of the screen. This will allow the `ViewController` to tell this label what question to display.

Set the `answerLabel` outlet the same way: drag from the circle beside `answerLabel` to the bottom `UILabel` (Figure 1.16).

Figure 1.16 Setting `answerLabel`



Notice that you drag *from* the object with the outlet that you want to set *to* the object that you want that outlet to point to.

Your outlets are all set. The next connections you need to make involve the two buttons.

When a `UIButton` is tapped, it sends a message to another object. The object that receives the message is called the *target*. The message that is sent is called the *action*. This action is the name of the method that contains the code to be executed in response to the button being tapped.

In your application, the target for both buttons will be the instance of `ViewController`. Each button will have its own action. Let's start by defining the two action methods: `showQuestion(_:)` and `showAnswer(_:)`.

Defining action methods

Return to `ViewController.swift` and add the following code in between `class ViewController:` and `UIViewController {` and `}.`

```
class ViewController: UIViewController {
    @IBOutlet weak var questionLabel: UILabel!
    @IBOutlet weak var answerLabel: UILabel!

    @IBAction func showQuestion(sender: AnyObject) {
    }

    @IBAction func showAnswer(sender: AnyObject) {
    }
}
```

You will flesh out these methods after you make the target and action connections. The `@IBAction` keyword tells Xcode that you will be making these connections in Interface Builder.

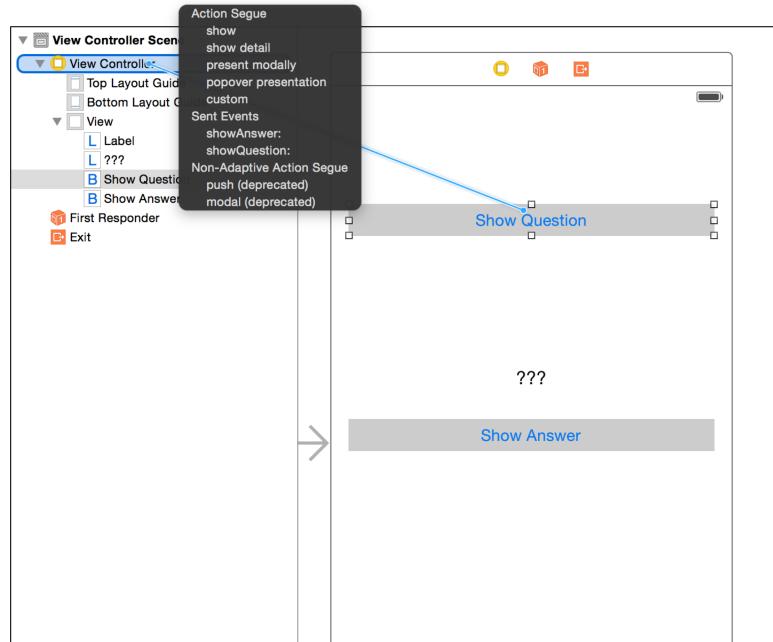
Setting targets and actions

To set an object's target, you Control-drag *from* the object *to* its target. When you release the mouse, the target is set, and a pop-up menu appears that lets you select an action.

Let's start with the Show Question button. You want its target to be `ViewController` and its action to be `showQuestion(_)`.

Reopen Main.storyboard. Select the Show Question button in the canvas and Control-drag (or right-click and drag) to the View Controller. When the View Controller is highlighted, release the mouse button and choose `showQuestion:` from the pop-up menu, as shown in Figure 1.17.

Figure 1.17 Setting Show Question target/action



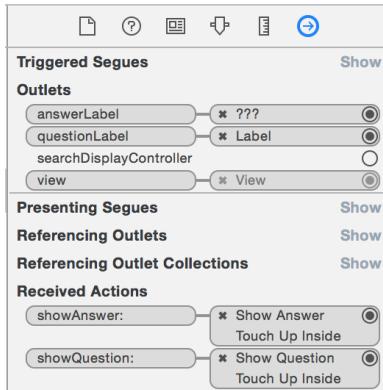
Now for the Show Answer button. Select the button and Control-drag from the button to the View Controller. Then choose `showAnswer:` from the pop-up menu.

Summary of connections

There are now five connections between your `ViewController` and the view objects. You have set the properties `answerLabel` and `questionLabel` to point at the label objects – two connections. The `ViewController` is the target for both buttons – two more. The project's template made one additional connection: the `view` pointer of `ViewController` is connected to the `View` object that represents the background of the application. That makes five.

You can check these connections in the *connections inspector*. Select the View Controller in the outline view. Then in the inspector, click the  tab to reveal the connections inspector. (Figure 1.18).

Figure 1.18 Checking connections in the inspector



Your storyboard file is complete. The view objects have been created and configured, and all the necessary connections have been made to the controller object. Let's move on to creating and connecting your model objects.

Creating Model Objects

View objects make up the user interface, so developers typically create, configure, and connect view objects using Interface Builder. Model objects, on the other hand, are set up in code.

In the project navigator, select `ViewController.swift`. Add the following code that declares an integer and two arrays of strings.

```
class ViewController: UIViewController {

    @IBOutlet weak var questionLabel: UILabel!
    @IBOutlet weak var answerLabel: UILabel!

    let questions = ["From what is cognac made?",  
                    "What is 7+7?",  
                    "What is the capital of Vermont?"]  
    let answers = ["Grapes",  
                  "14",  
                  "Montpelier"]  
    var currentQuestionIndex = 0
```

The arrays will be ordered lists containing questions and answers. The integer will keep track of what question the user is on.

Implementing action methods

In `ViewController.swift`, finish the implementations of `showQuestion(_:)` and `showAnswer(_:)`.

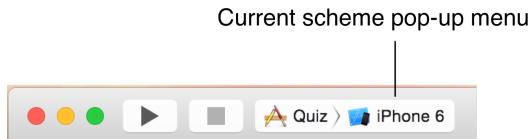
```
@IBAction func showQuestion(sender: AnyObject) {  
    ++currentQuestionIndex  
    if currentQuestionIndex == questions.count {  
        currentQuestionIndex = 0  
    }  
  
    let question = questions[currentQuestionIndex]  
    questionLabel.text = question  
    answerLabel.text = "???"  
}  
  
@IBAction func showAnswer(sender: AnyObject) {  
    let answer = answers[currentQuestionIndex]  
    answerLabel.text = answer  
}
```

Running on the Simulator

First, you are going to run Quiz on Xcode's iOS simulator. Later, you will see how to run it on an actual device.

To prepare Quiz to run on the simulator, find the current scheme pop-up menu on the Xcode toolbar (Figure 1.19).

Figure 1.19 iPhone 6 scheme selected



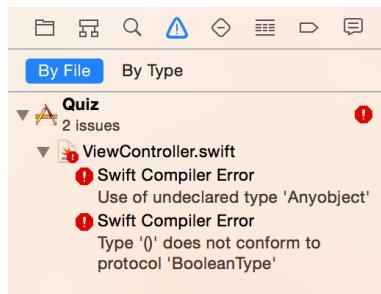
If it says something generic like iPhone 6, then the project is set to run on the simulator and you are good to go. If it says something like Christian's iPhone, then click it and choose iPhone 6 from the pop-up menu.

For this book, use the iPhone 6 scheme. If you use a scheme for one of the smaller iPhones, like the iPhone 4s, parts of your user interface might get cut off. We will discuss how to scale interfaces for both iPhone screen sizes (and iPad as well) in Chapter 10.

Next, click the triangular play button in the toolbar. This will build (compile) and then run the application. You will be doing this often enough that you may want to learn and use the keyboard shortcut Command-R.

If building turns up any errors, you can view them in the *issue navigator* by selecting the **!** tab in the navigator area (Figure 1.20).

Figure 1.20 Issue navigator with example errors and warnings



You can click on any error or warning in the issue navigator to be taken to the file and the line of code where the issue occurred. Find and fix any problems (i.e., code typos!) by comparing your code with the book's. Then try running the application again. Repeat this process until your application compiles.

Once your application has compiled, it will launch in the iOS simulator. If this is the first time you have used the simulator, it may take a while for Quiz to appear.

Play around with the Quiz application. You should be able to tap the Show Question button and see a new question in the top label; tapping Show Answer should show the right answer. If your application is not working as expected, double-check your connections in `Main.storyboard`.

Deploying an Application

Now that you have written your first iOS application and run it on the simulator, it is time to deploy it to a device.

To install an application on your development device, you need a developer certificate from Apple. Developer certificates are issued to registered iOS Developers who have paid the developer fee. This certificate grants you the ability to sign your code, which allows it to run on a device. Without a valid certificate, devices will not run your application.

Apple's Developer Program Portal (<http://developer.apple.com>) contains all the instructions and resources to get a valid certificate. The interface for the set-up process is continually being updated by Apple,

so it is fruitless to describe it here in detail. Instead, visit our guide at http://www.bignerdranch.com/ios_device_provisioning for instructions.

If you are curious about what exactly is going on here, there are four important items in the provisioning process:

Developer Certificate	This certificate file is added to your Mac's keychain using Keychain Access. It is used to digitally sign your code.
App ID	The application identifier is a string that uniquely identifies your application on the App Store. Application identifiers typically look like this: com.bignerdranchAwesomeApp, where the name of the application follows the name of your company.
	The App ID in your provisioning profile must match the <i>bundle identifier</i> of your application. A development profile can have a wildcard character (*) for its App ID and therefore will match any bundle identifier. To see the bundle identifier for the Quiz application, select the project in the project navigator. Then select the Quiz target and the General pane.
Device ID (UDID)	This identifier is unique for each iOS device.
Provisioning Profile	This is a file that lives on your development device and on your computer. It references a Developer Certificate, a single App ID, and a list of the device IDs for the devices that the application can be installed on. This file is suffixed with .mobileprovision.

When an application is deployed to a device, Xcode uses a provisioning profile on your computer to access the appropriate certificate. This certificate is used to sign the application binary. Then, the development device's UDID is matched to one of the UDIDs contained within the provisioning profile, and the App ID is matched to the bundle identifier. The signed binary is then sent to your development device, where it is confirmed by the same provisioning profile on the device and, finally, launched.

Open Xcode and plug your development device (iPhone, iPod touch, or iPad) into your computer. This should automatically open the Organizer window, which you can re-open at any time using the Window menu's Organizer item. In the Organizer window, you can select the Devices tab to view all of the provisioning information.

To run the Quiz application on your device, you must tell Xcode to deploy to the device instead of the simulator. Return to the current scheme description in Xcode's toolbar. Click the description and then choose iOS Device from the pop-up menu (Figure 1.21). If iOS Device is not an option, find the choice that reads something like Christian's iPhone.

Figure 1.21 Choosing the device



Build and run your application (Command-R), and it will appear on your device.

Application Icons

While running the Quiz application (on your development device or the simulator), return to the device's Home screen. You will see that its icon is a boring, default tile. Let's give Quiz a better icon.

An *application icon* is a simple image that represents the application on the iOS home screen. Different devices require different sized icons, and these requirements are shown in Table 1.1.

Table 1.1 Application icon sizes by device

Device	Application icon sizes
iPhone 6 Plus	180x180 pixels (@3x)

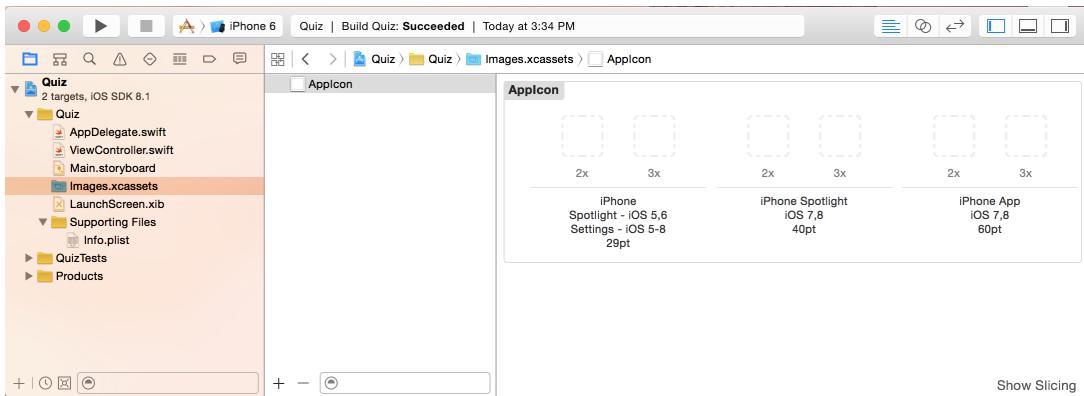
Device	Application icon sizes
iPhone 6 and iPhone 5	120x120 pixels (@2x)
iPad and iPad mini	152x152 pixels (@2x)

We have prepared an icon image file (size 120x120) for the Quiz application. You can download this icon (along with resources for other chapters) from <http://www.bignerdranch.com/solutions/iOSProgramming4ed.zip>. Unzip `iOSProgramming4ed.zip` and find the `Icon@2x.png` file in the `Resources` directory of the unzipped folder.

You are going to add this icon to your application bundle as a *resource*. In general, there are two kinds of files in an application: code and resources. Code (like `ViewController.swift`) is used to create the application itself. Resources are things like images and sounds that are used by the application at runtime. Storyboard files are also resources.

In the project navigator, find `Images.xcassets`. Select this file to open it and then select `AppIcon` from the resource list on the left side.

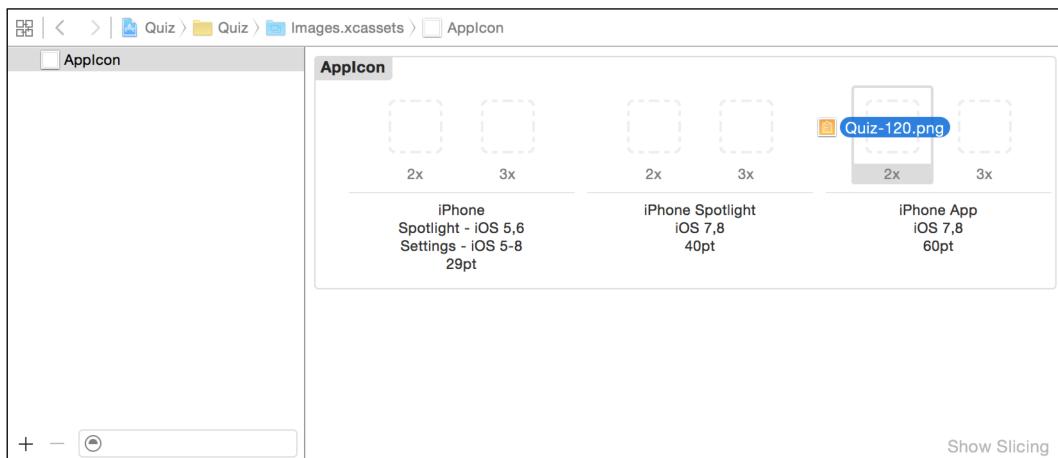
Figure 1.22 Showing the Asset Catalog



This panel is the *Asset Catalog* where you can manage all of the various images that your application will need.

Drag the `Icon@2x.png` file from Finder onto the margins of the `AppIcon` section. This will copy the file into your project's directory on the filesystem and add a reference to that file in the Asset Catalog. (You can Control-click on a file in the Asset Catalog and select the option to *Show in Finder* to confirm this.)

Figure 1.23 Adding the App Icon to the Asset Catalog



Build and run the application again. Exit the application and look for the Quiz application.

(If you do not see the icon, delete the application and then build and run again to redeploy it. On a device, do this as you would any other application. On the simulator, the easiest option is to reset the simulator. With the

simulator open, find its menu bar. Select iOS Simulator and then Reset Content and Settings.... This will remove all applications and reset the simulator to its default settings. You should see the app icon the next time you run the application.)

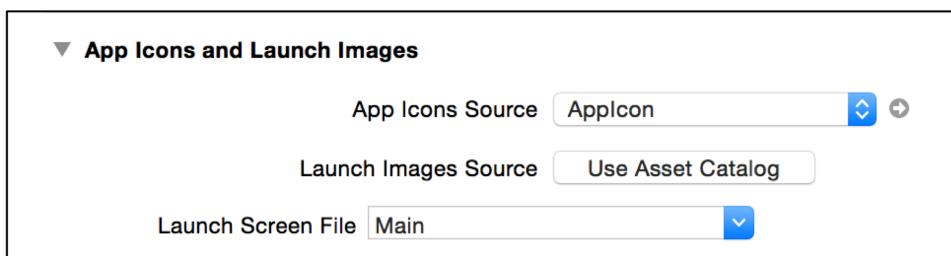
Launch Images

Another item you should set for the project is the *launch image*, which appears while an application is loading. The launch image has a specific role on iOS: it conveys to the user that the application is indeed launching and depicts the user interface that the user will interact with once the application has finished launching. Therefore, a good launch image is a content-less screenshot of the application. For example, the Clock application's launch image shows the four tabs along the bottom, all in the unselected state. Once the application loads, the correct tab is selected and the content becomes visible. (Keep in mind that the launch image is replaced after the application has launched; it does not become the background image of the application.)

An easy way to accomplish this is to allow Xcode to generate all of the possible launch screen images for you using a *launch screen file*. A launch screen file is one of the interface files in the project that will be used to generate launch images.

Open the project settings by clicking on Homeowner in the Project navigator. Under App Icons and Launch Images, choose Main from the Launch Screen File drop-down (Figure 1.24). Launch images will now be generated from Main.storyboard.

Figure 1.24 Setting the Launch Screen File



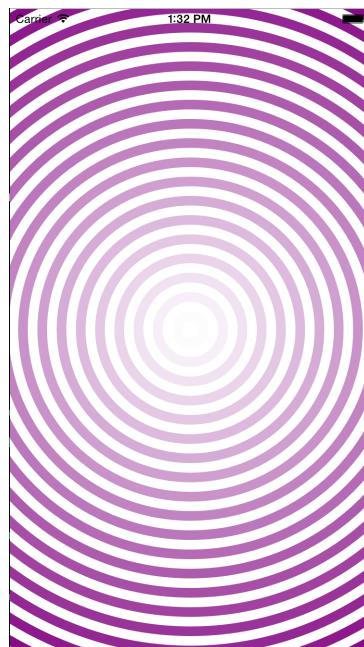
Congratulations! You have written your first application and installed it on your device. Now it is time to dive into the big ideas that make it work.

2

Views and the View Hierarchy

In this chapter, you will learn about views and the view hierarchy. In particular, you are going to write an app named Hypnosister that draws a full-screen set of concentric circles.

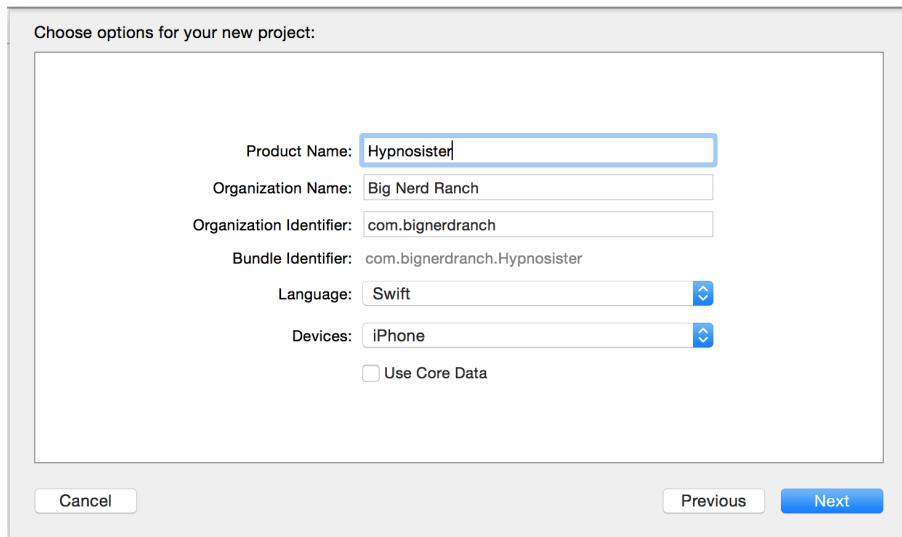
Figure 2.1 Hypnosister



In Xcode, select File → New → Project... (or use the keyboard shortcut Command-Shift-N). From the iOS section, select Application, choose the Single View Application template, and click Next.

Enter Hypnosister for the product name. Make sure Swift is selected from the Language drop down and iPhone is selected from the Devices drop down. Also make sure the Use Core Data box is unchecked. Confirm that it matches Figure 2.2. Click Next and then Create on the following screen.

Figure 2.2 Configuring Hypnosister



In ???, you used a storyboard file to lay out the interface. Storyboards help remove a lot of repetitive code in your project and can assist in visualizing the flow of the application, but they can obfuscate what is happening under the covers, so to speak. So for the beginning of this book, we are going to remove the storyboard from our projects so that you can focus on understanding how the various pieces of the puzzle fit together.

From the Project navigator, delete the two files `Main.storyboard` and `ViewController.swift`. Then, select Hypnosister from the top of the Project navigator. Make sure the Hypnosister target is selected, then under General remove the text that says Main next to Main Interface.

Finally, open `AppDelegate.swift` and add code to `application(_:didFinishLaunchingWithOptions:)` to set up the window.

```
func application(application: UIApplication,
                 didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
    // Override point for customization after application launch.
    window = UIWindow(frame: UIScreen mainScreen().bounds)

    window!.backgroundColor = UIColor.whiteColor()
    window!.makeKeyAndVisible()

    return true
}
```

Hypnosister will not have any user interaction so that you can focus on how views are drawn to the screen. Let's start with a little theory of views and the view hierarchy.

View Basics

- A view is an instance of `UIView` or one of its subclasses.
- A view knows how to draw itself.
- A view handles events, like touches.
- A view exists within a hierarchy of views. The root of this hierarchy is the application's window.

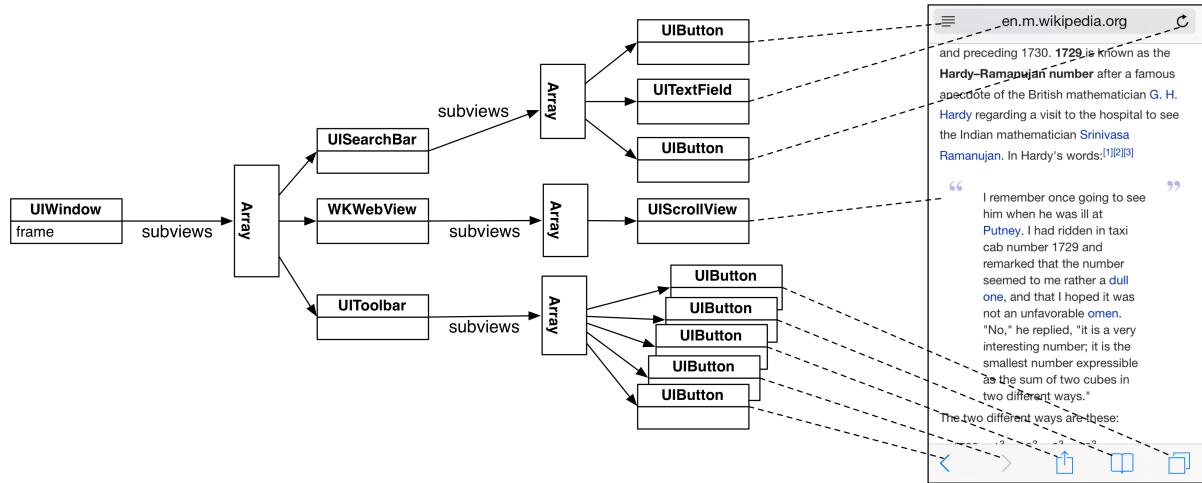
In Chapter 1, you created four views for the Quiz app: two instances of `UIButton` and two instances of `UILabel`. You created and configured these views in Interface Builder, but you can also create views programmatically. In Hypnosister, you will create views programmatically.

The View Hierarchy

Every application has a single instance of **UIWindow** that serves as the container for all the views in the application. The window is created when the application launches. Once the window is created, you can add other views to it.

When a view is added to the window, it is said to be a *Subview* of the window. Views that are subviews of the window can also have subviews, and the result is a hierarchy of view objects with the window at its root.

Figure 2.3 An example view hierarchy and the interface that it creates

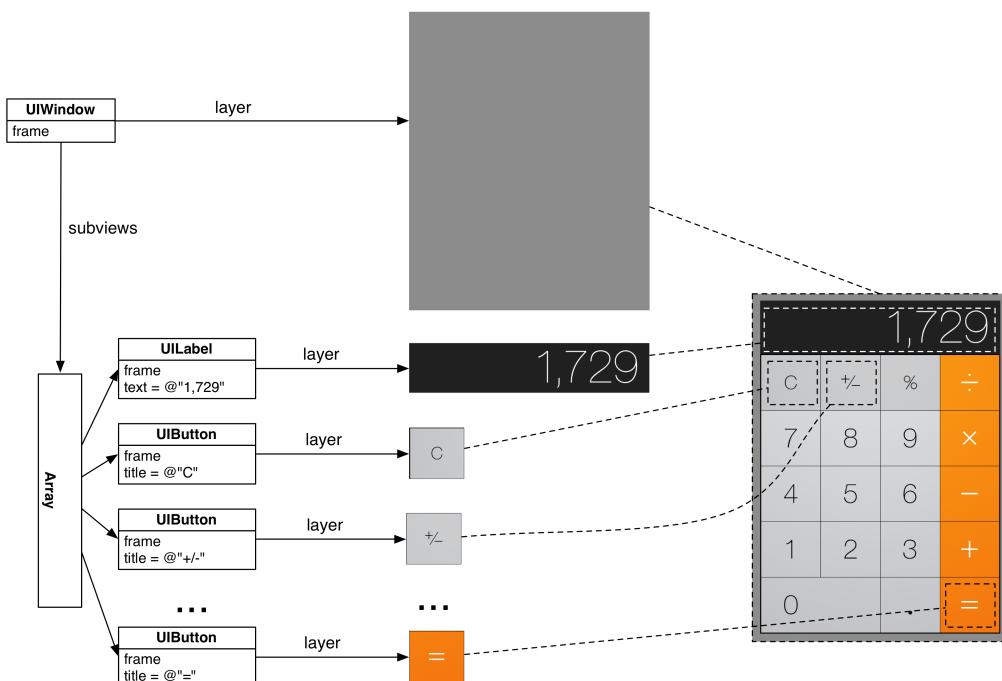


Once the view hierarchy has been created, it will be drawn to the screen. This process can be broken into two steps:

- Each view in the hierarchy, including the window, draws itself. It renders itself to its *layer*, which is an instance of **CALayer**. (You can think of a view's layer as a bitmap image.)
- The layers of all the views are composited together on the screen.

Figure 2.4 shows another example view hierarchy and the two drawing steps.

Figure 2.4 Views render themselves and then are composited together



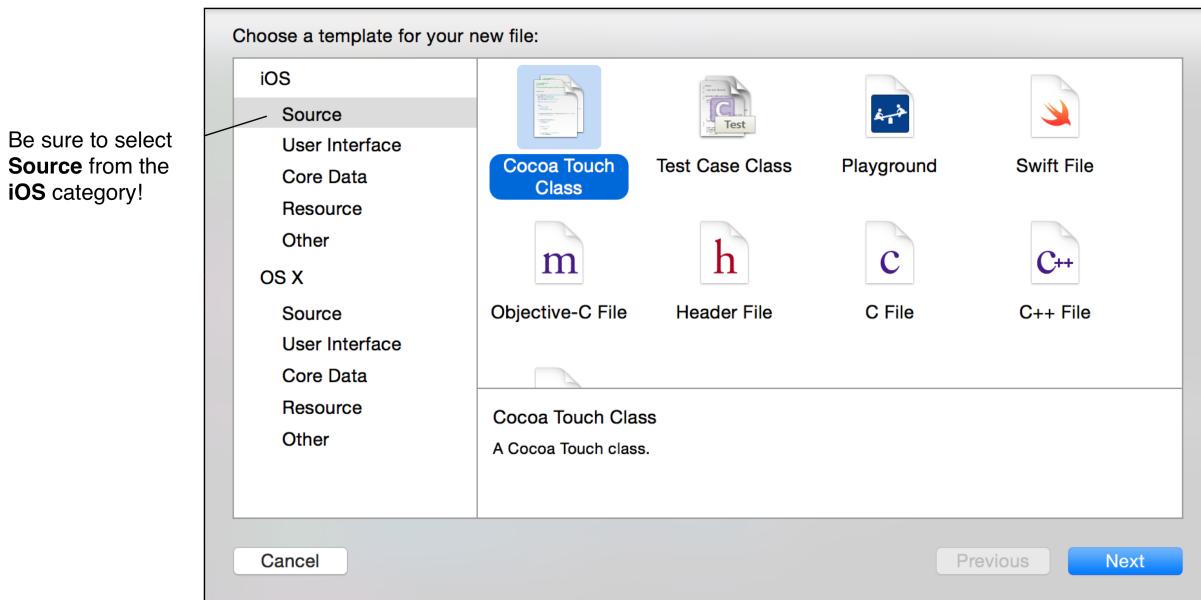
Classes like **UIButton** and **UILabel** already know how to render themselves to their layers. For instance, in Quiz, you created instances of **UILabel** and told them what text to display, but you did not have to tell them how to draw text. Apple's developers took care of that.

Apple, however, does not provide a class whose instances know how to draw concentric circles. Thus, for Hypnosister, you are going to create your own **UIView** subclass and write custom drawing code.

Subclassing **UIView**

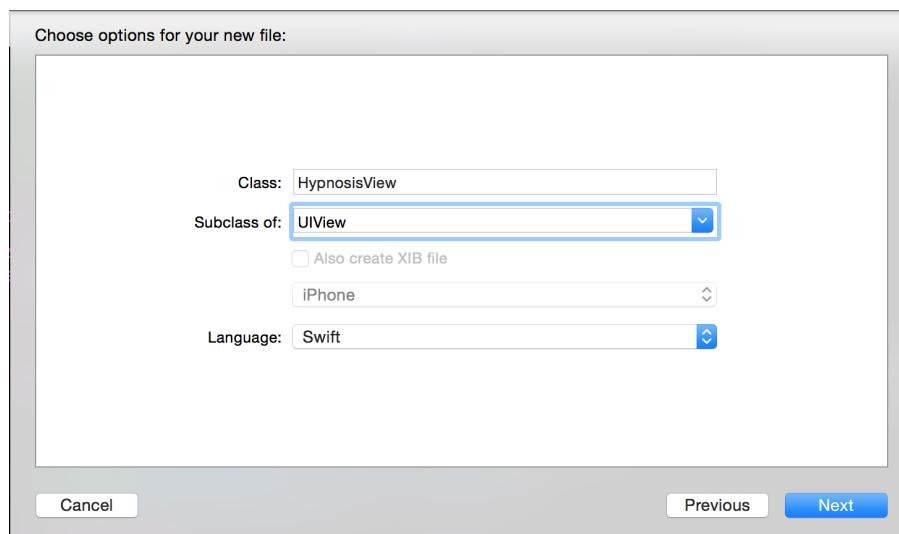
To create a **UIView** subclass, select File → New → File... (or press Command-N). From the iOS section, select Source and then choose Cocoa Touch Class (Figure 2.5).

Figure 2.5 Creating a new class



Click Next. On the next pane, name the class **HypnosisView** and select **UIView** as the superclass. Finally, confirm that the Language is Swift.

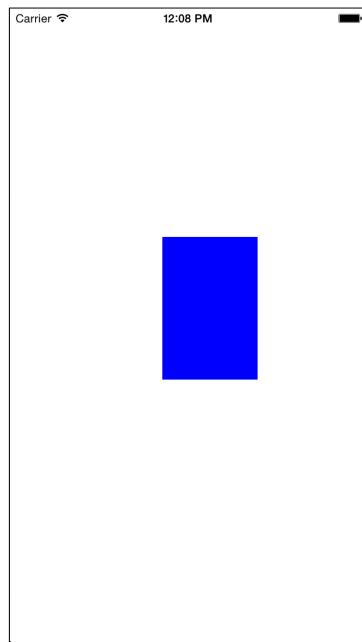
Figure 2.6 Choosing **UIView** as the superclass



Click Next. Make sure that Hypnosister is checked beside Targets and then click Create.

Before writing the concentric circle drawing code for **HypnosisView**, let's focus on how to create a view programmatically and get it on screen. To keep things simple, in this first part, an instance of **HypnosisView** view will not draw concentric circles. Instead, it will draw a rectangle with a blue background.

Figure 2.7 Initial version of **HypnosisView**



Views and frames

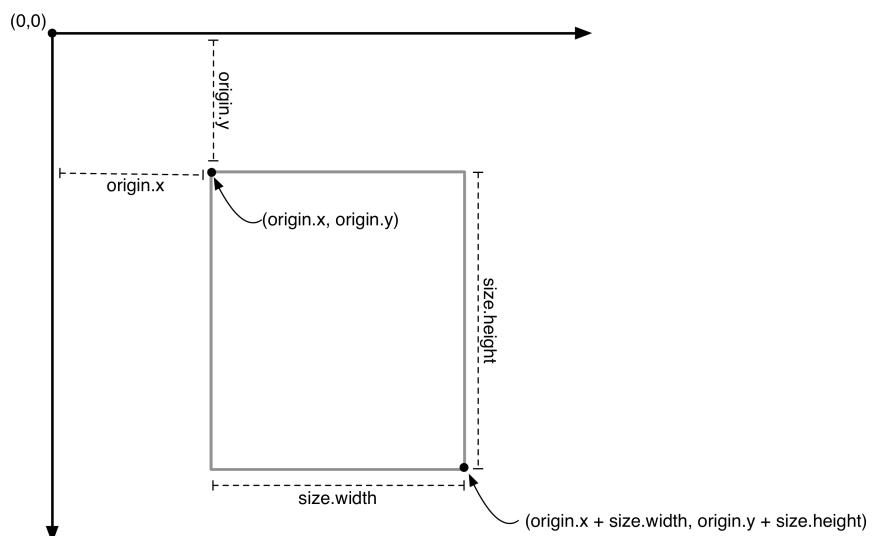
When you initialize a view programmatically, you use its `init(frame:)` designated initializer. This method takes one argument, a `CGRect`, that will become the view's `frame`, a property on `UIView`.

```
var frame: CGRect
```

A view's `frame` specifies the view's size and its position relative to its superview. Because a view's size is always specified by its `frame`, a view is always a rectangle.

A `CGRect` contains the members `origin` and `size`. The `origin` is a structure of type `CGPoint` and contains two `CGFloat` members: `x` and `y`. The `size` is a structure of type `CGSize` and has two `CGFloat` members: `width` and `height` (Figure 2.8).

Figure 2.8 `CGRect`



Open `AppDelegate.swift`.

In `AppDelegate.swift`, find the template's implementation of `application(_:didFinishLaunchingWithOptions:)`. After the line that creates the window, create a `CGRect` that will be the frame of a `HypnosisView`. Next, create an instance of `HypnosisView` and set its `backgroundColor` property to blue. Finally, add the `HypnosisView` as a subview of the window to make it part of the view hierarchy.

```
func application(application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
    // Override point for customization after application launch.
    window = UIWindow(frame: UIScreen.mainScreen().bounds)

    let firstFrame = CGRect(x: 160, y: 240, width: 100, height: 150)
    let firstView = HypnosisView(frame: firstFrame)
    firstView.backgroundColor = UIColor.blueColor()
    window!.addSubview(firstView)

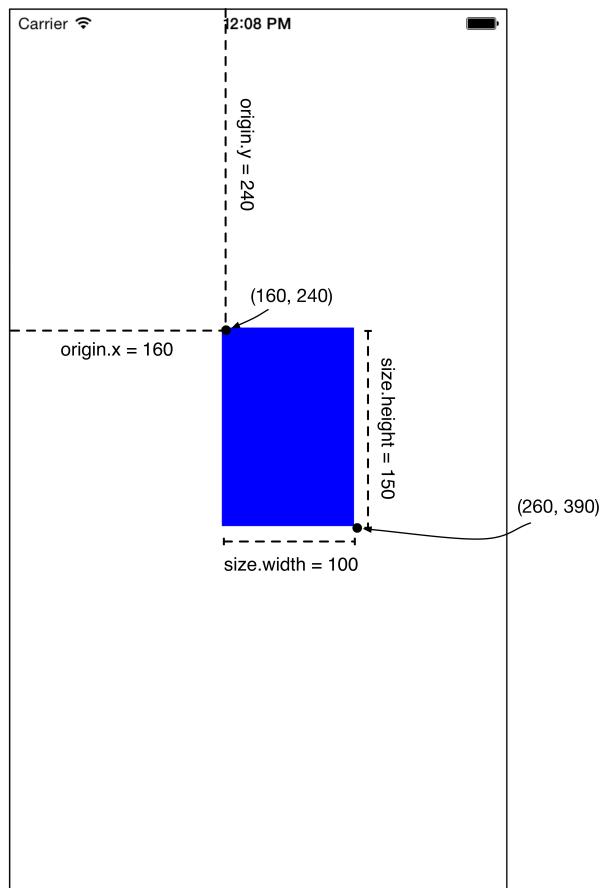
    window!.backgroundColor = UIColor.whiteColor()
    window!.makeKeyAndVisible()
    return true
}
```

To create a `CGRect`, you use its initializer and pass in the values for the `origin.x`, `origin.y`, `size.width` and `size.height`.

To set the `backgroundColor`, you used the `UIColor` class method `blueColor()`. This is a convenience method; it allocates and initializes an instance of `UIColor` that is configured to be blue. There are a number of `UIColor` convenience methods for common colors, such as `greenColor()`, `blackColor()`, and `clearColor()`.

Build and run the application. The blue rectangle is the instance of `HypnosisView`. Because the `HypnosisView`'s frame's `origin` is `(160, 240)`, its top left corner is 160 points to the right and 240 points down from the top-left corner of the window (its superview). The view stretches 100 points to the right and 150 points down from its `origin`, in accordance with its frame's `size`.

Figure 2.9 Hypnosister with one `HypnosisView`



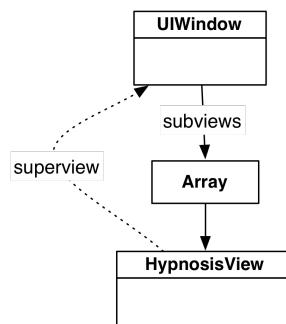
Note these values are in points, not pixels. If the values were in pixels, then they would not be consistent across displays of different resolutions (i.e., Retina vs. non-Retina). A single point is a relative unit of a measure; it will be a different number of pixels depending on how many pixels there are in the display. Sizes, positions, lines, and curves are always described in points to allow for differences in display resolution.

On a Retina Display, a pixel is half a point tall and half a point wide by default. On a non-Retina Display, one pixel is one point tall and one point wide by default. When printing to paper, an inch is 72 points long.

In Xcode's console, notice the comment informing you that “Application windows are expected to have a root view controller at the end of application launch.” A view controller is an object that controls some set of an application’s view hierarchy, and most iOS apps have one or more view controllers. Hypnosister, however, is simple enough that it does not need a view controller, so you can ignore this comment. You will learn about view controllers in Chapter 4.

Take a look at the view hierarchy that you have created:

Figure 2.10 **UIWindow** has one subview – a **HypnosisView**



Every instance of **UIView** has a **superview** property. When you add a view as a subview of another view, the inverse relationship is automatically established. In this case, the **HypnosisView**’s **superview** is the **UIWindow**. (To avoid a strong reference cycle, the **superview** property is a weak reference.)

Let’s experiment with your view hierarchy. In `AppDelegate.swift`, create another instance of **HypnosisView** with a different `frame` and background color.

```

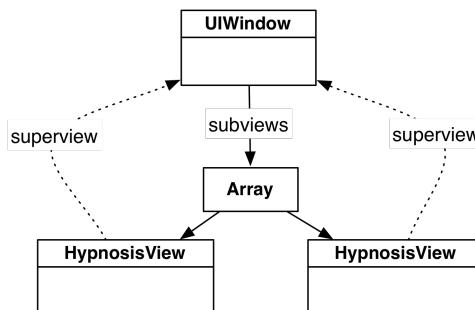
...
window!.addSubview(firstView)

let secondFrame = CGRect(x: 20, y: 30, width: 50, height: 50)
let secondView = HypnosisView(frame: secondFrame)
secondView.backgroundColor = UIColor.greenColor()
window!.addSubview(secondView)

window!.backgroundColor = UIColor.whiteColor()
...
  
```

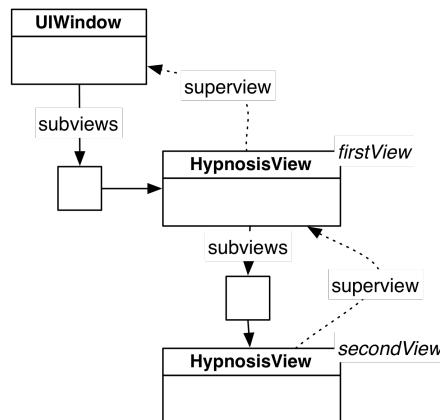
Build and run again. In addition to the blue rectangle, you will see a green square near the top lefthand corner of the window. Figure 2.11 shows the updated view hierarchy.

Figure 2.11 **UIWindow** has two subviews as siblings



A view hierarchy can be deeper than two levels. Let's make that happen by adding the second instance of **HypnosisView** as a subview of the first instance of **HypnosisView** instead of the window:

Figure 2.12 One **HypnosisView** as a subview of the other



In **AppDelegate.swift**, make this change.

```

...
let secondView = HypnosisView(frame: secondFrame)
secondView.backgroundColor = UIColor.greenColor()

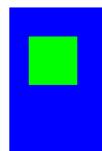
window!.addSubview(secondView)
firstView.addSubview(secondView)

...
  
```

Build and run the application. Notice that `secondView`'s position on the screen has changed. A view's `frame` is relative to its superview so the top-left corner of `secondView` is now inset (20, 30) points from the top-left corner of `firstView`.

Figure 2.13 Hypnosister with new hierarchy

Carrier 12:15 PM



(If the green instance of **HypnosisView** looks smaller than it did previously, that is just an optical illusion. Its size has not changed.)

Now that you have had some experience with the view hierarchy, remove the second instance of **HypnosisView** before continuing.

```
...
window!.addSubview(firstView)

let secondFrame = CGRect(x: 20, y: 30, width: 50, height: 50)
let secondView = UIView(frame: secondFrame)
secondView.backgroundColor = UIColor.greenColor()
firstView.addSubview(secondView)

window!.backgroundColor = UIColor.whiteColor()

...
```

Custom Drawing in drawRect(_:)

So far, you have subclassed **UIView**, created instances of the subclass, inserted them into the view hierarchy, and specified their frames and background colors. In this section, you will write the custom drawing code for **HypnosisView** in its **drawRect(_:)** method.

The **drawRect(_:)** method is the rendering step where a view draws itself onto its layer. **UIView** subclasses override **drawRect(_:)** to perform custom drawing. For example, the **drawRect(_:)** method of **UIButton** draws light-blue text centered in a rectangle.

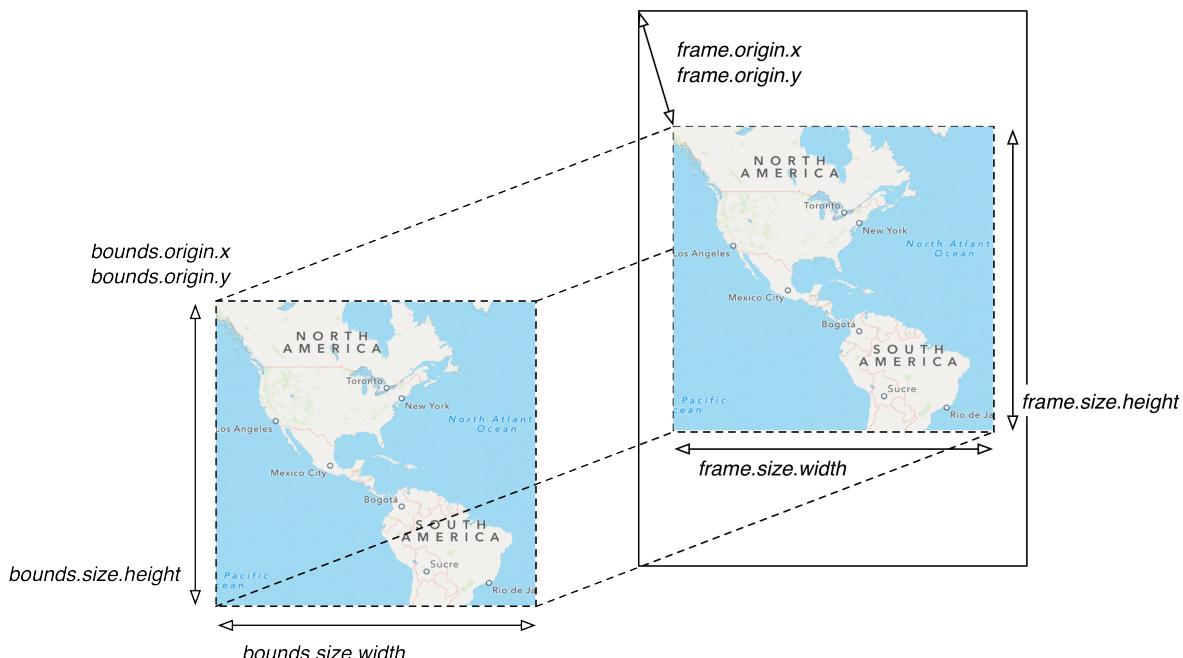
The first thing that you typically do when overriding **drawRect(_:)** is get the bounds rectangle of the view. The **bounds** property, inherited from **UIView**, is the rectangle that defines the area where the view will draw itself.

Each view has a coordinate system that it uses when drawing itself. The **bounds** is a view's rectangle in *its own* coordinate system. The **frame** is the same rectangle in *its superview's* coordinate system.

You might be wondering, “Why do we need another rectangle when we already have **frame**? ”

The **frame** and **bounds** rectangles have distinct purposes. A view's **frame** rectangle is used during compositing to lay out the view's layer relative to the rest of the view hierarchy. The **bounds** rectangle is used during the rendering step to lay out detailed drawing within the boundaries of the view's layer. (Figure 2.14).

Figure 2.14 bounds vs. frame



You can use the bounds property of the window to define the frame for a full-screen instance of **HypnosisView**.

In `AppDelegate.swift`, update `firstView`'s frame to match the bounds of the window.

```
func application(application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
    // Override point for customization after application launch.
    window = UIWindow(frame: UIScreen.mainScreen().bounds)

    let firstFrame = CGRect(x: 160, y: 240, width: 100, height: 150)
    let firstFrame = window!.bounds
    let firstView = HypnosisView(frame: firstFrame)
    firstView.backgroundColor = UIColor.blueColor()
    window!.addSubview(firstView)

    window!.backgroundColor = UIColor.whiteColor()
    window!.makeKeyAndVisible()
    return true
}
```

Build and run the application, and you will be greeted with a full-sized view with a blue background.

Drawing a single circle

You are going to ease into the drawing code by drawing a single circle – the largest that will fit within the bounds of the view.

In `HypnosisView.swift`, uncomment `drawRect(_:_)` if the template commented it out, or add in the code if it is not present.

```
override func drawRect(rect: CGRect) {
}
```

Add code to `drawRect(_:_)` that finds the center point of bounds.

```
override func drawRect(rect: CGRect) {
    let bounds = self.bounds

    // Figure out the center of the bounds rectangle
    let centerX = bounds.origin.x + bounds.size.width / 2.0
    let centerY = bounds.origin.y + bounds.size.height / 2.0
    let center = CGPointMake(x: centerX, y: centerY)
}
```

Next, set the radius for your circle to be half of the smaller of the view's dimensions. (Determining the smaller dimension will draw the right circle in portrait and landscape orientations.)

```
override func drawRect(rect: CGRect) {
    let bounds = self.bounds

    // Figure out the center of the bounds rectangle
    let centerX = bounds.origin.x + bounds.size.width / 2.0
    let centerY = bounds.origin.y + bounds.size.height / 2.0
    let center = CGPointMake(x: centerX, y: centerY)

    // The circle will be the largest that will fit in the view
    let radius = min(bounds.size.width, bounds.size.height) / 2.0
}
```

UIBezierPath

The next step is to draw the circle using the **UIBezierPath** class. Instances of this class define and draw lines and curves that you can use to make shapes, like circles.

First, create an instance of **UIBezierPath**.

```

override func drawRect(rect: CGRect) {
    let bounds = self.bounds

    // Figure out the center of the bounds rectangle
    let centerX = bounds.origin.x + bounds.size.width / 2.0
    let centerY = bounds.origin.y + bounds.size.height / 2.0
    let center = CGPoint(x: centerX, y: centerY)

    // The circle will be the largest that will fit in the view
    let radius = min(bounds.size.width, bounds.size.height)

    let path = UIBezierPath()
}

```

The next step is defining the path that the **UIBezierPath** object should follow. How do you define a circle-shaped path? The best place to find an answer to this question is the **UIBezierPath** class reference in Apple's developer documentation.

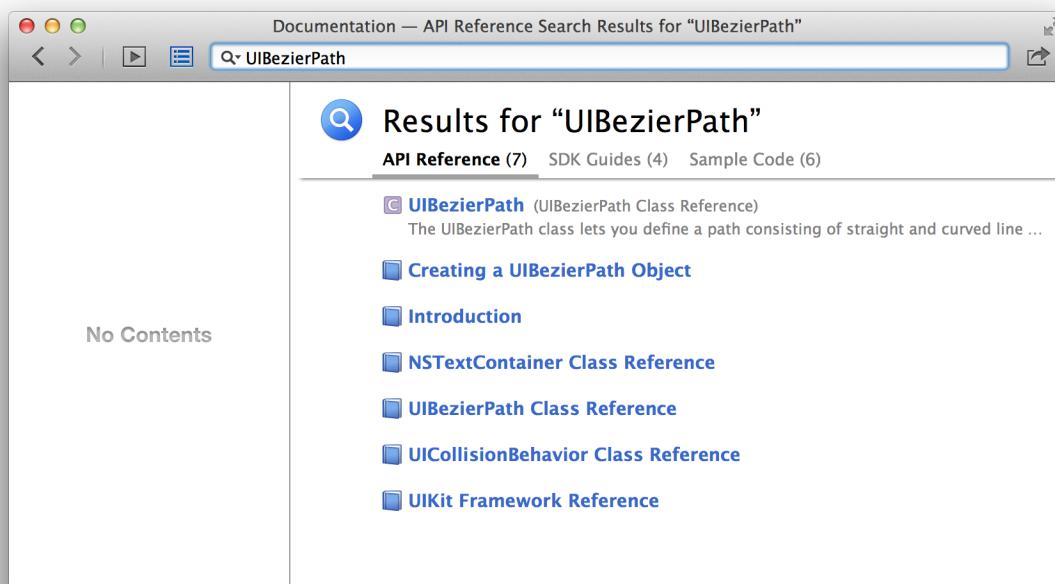
Using the developer documentation

From Xcode's menu, select Help → Documentation and API Reference. You can also use the keyboard shortcut Option-Command-? (be sure to hold down the Shift key, too, to get the '?').

(When you access the documentation, Xcode may try to go get the latest for you from Apple. You may be asked for your Apple ID and password.)

When the documentation browser opens, search for **UIBezierPath**. You will be offered several results. Find and select **UIBezierPath Class Reference**.

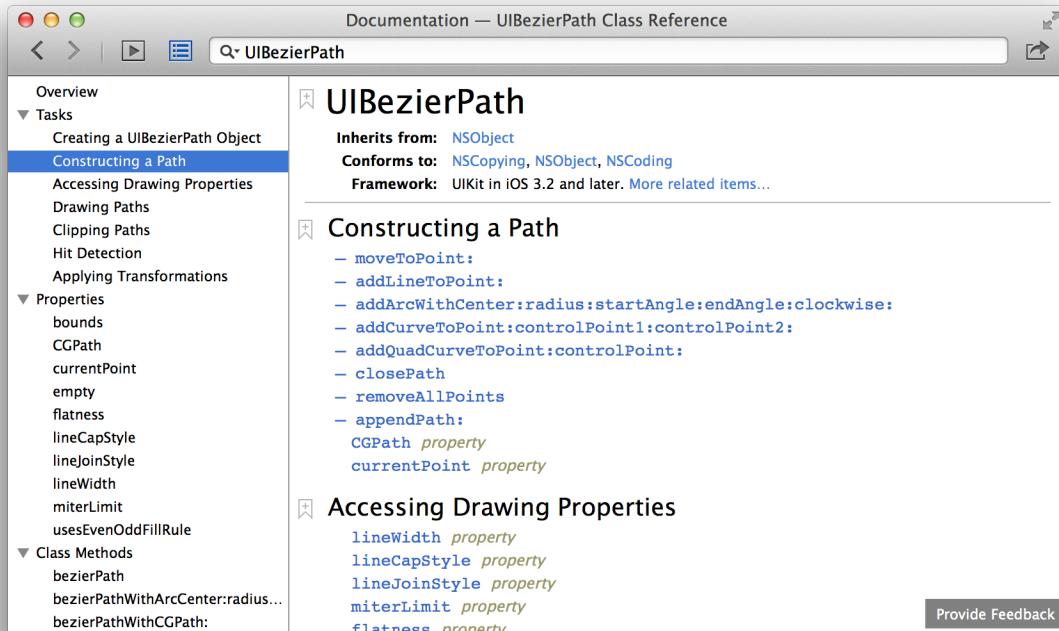
Figure 2.15 Documentation results



This page opens to an overview of the class, which is interesting, but let's stay focused on your circle-shaped path question. The lefthand side of the reference is the table of contents. (If you do not see a table of contents, select the **Table of Contents** icon at the top left of the browser.)

In the table of contents, find the Tasks section. This is a good place to begin the hunt for a method that does something specific. The first task is **Creating a UIBezierPath Object**. You have already done that, so take a look at the second task: **Constructing a Path**. Select this task, and you will see a list of relevant **UIBezierPath** methods.

Figure 2.16 Methods for constructing a path



A likely candidate for a circular path is

`addArcWithCenter(center:radius:startAngle:endAngle:clockwise:)`. Click this method to see more details about its parameters. You have already computed the center and the radius. The start and end angle values are in radians. To draw a circle, you will use `0` for the start angle and `M_PI * 2` for the end angle. (If your trigonometry is rusty, you can take our word on this or click the Figure 1 link within the Discussion of this method's documentation to see a diagram of the unit circle.) Finally, because you are drawing a complete circle, the clockwise parameter will not matter. It is a required parameter, however, so you will need to give it a value.

In `HypnosisView.swift`, send a message to the `UIBezierPath` that defines its path.

```
override func drawRect(rect: CGRect) {
    let bounds = self.bounds

    // Figure out the center of the bounds rectangle
    let centerX = bounds.origin.x + bounds.size.width / 2.0
    let centerY = bounds.origin.y + bounds.size.height / 2.0
    let center = CGPointMake(x: centerX, y: centerY)

    // The circle will be the largest that will fit in the view
    let radius = min(bounds.size.width / 2.0, bounds.size.height / 2.0)

    let path = UIBezierPath()
    // Add an arc to the path at center, with radius of radius,
    // from 0 to 2*PI radians (a circle)
    path.addArcWithCenter(center,
        radius: radius,
        startAngle: 0,
        endAngle: CGFloat(M_PI * 2.0),
        clockwise: true)
}
```

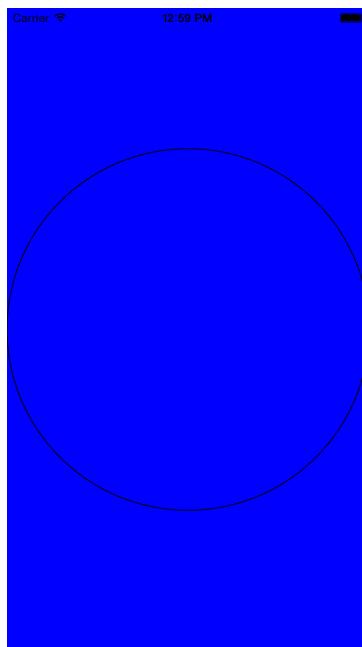
You have defined a path, but you have not drawn anything yet. Back in the `UIBezier` class reference, find and select the Drawing Paths task. From these methods, the best choice is `stroke()`. (The other methods either fill in the entire shape or require a `CGBitmapMode` that you do not need.)

In `HypnosisView.swift`, send a message to the `UIBezierPath` that tells it to draw.

```
override func drawRect(rect: CGRect) {  
    ...  
  
    let path = UIBezierPath()  
  
    // Add an arc to the path at center, with radius of radius,  
    // from 0 to 2*PI radians (a circle)  
    path.addArcWithCenter(center,  
        radius: radius,  
        startAngle: 0,  
        endAngle: CGFloat(M_PI * 2.0),  
        clockwise: true)  
  
    // Draw the line  
    path.stroke()  
}
```

Build and run the application, and you will see a thin, black outline of a circle that is as wide as the screen (or as tall if you are in landscape orientation).

Figure 2.17 **HypnosisView** with a single circle



Based on the original plan for Hypnosister, the line describing your circle is not yet right. It should be wider and purple.

To see how to fix these issues, return to the **UIBezierPath** reference. In the table of contents, find the Properties section. One of these properties should stand out as useful in this case – `lineWidth`. Select this property. You will see that `lineWidth` is of type `CGFloat` and that its default is `1.0`.

In `HypnosisView.swift`, make the width of the line 10 points.

```
override func drawRect(rect: CGRect) {
    ...

    let path = UIBezierPath()

    // Add an arc to the path at center, with radius of radius,
    // from 0 to 2*PI radians (a circle)
    path.addArcWithCenter(center,
        radius: radius,
        startAngle: 0,
        endAngle: CGFloat(M_PI * 2.0),
        clockwise: true)

    // Configure line width to 10 points
    path.lineWidth = 10

    // Draw the line
    path.stroke()
}
```

Build and run the application to confirm that the line is now wider.

There is no property in **UIBezierPath** that deals with the color of the line. But there is a clue in the class overview. Use the table of contents to return to the Overview. In the fifth paragraph (as of this writing), there is a parenthetical aside that reads, “You set the stroke and fill color using the **UIColor** class.”

The **UIColor** class is linked, so you can click it to be taken directly to the **UIColor** class reference. In **UIColor**’s Tasks section, select Drawing Operations and browse through the associated methods. For your purposes, you could use either **set()** or **setStroke()**. You will use **setStroke()** to make your code more obvious to others.

The **setStroke()** method is an instance method, so you need an instance of **UIColor** to send it to. Recall that **UIColor** has convenience methods that return common colors. You can see these methods listed under the Class Methods section of the **UIColor** reference, including one named **purpleColor()**.

Now you have the information you need. In **HypnosisView.swift**, add code to create a purple **UIColor** instance and send it the **setStroke()** message so that when the path is drawn, it will be drawn in purple.

```
override func drawRect(rect: CGRect) {
    ...

    // Configure line width to 10 points
    path.lineWidth = 10

    // Configure the drawing color to purple
    UIColor.purpleColor().setStroke()

    // Draw the line
    path.stroke()
}
```

Build and run the application, and you will see a wider, purple outline of a circle.

By now, you will have noticed that a view’s **backgroundColor** is drawn regardless of what **drawRect(_)** does. Often, you will set the **backgroundColor** of a custom view to be transparent, or “clear-colored,” so that only the results of **drawRect(_)** show.

In **AppDelegate.swift**, remove the code that sets the background color of the view.

```
let firstView = HypnosisView(frame: frame)
//firstView.backgroundColor = UIColor.blueColor()
window!.addSubview(firstView)
```

Then, in **HypnosisView.swift**, add code to **init(frame:)** to set the background color of every **HypnosisView** to clear.

```
override init(frame: CGRect) {
    super.init(frame: frame)

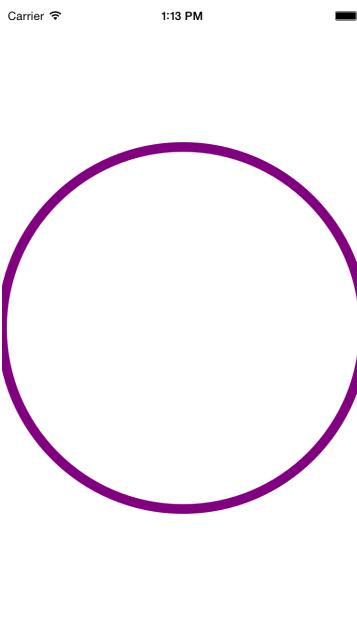
    // All HypnosisViews start with a clear background color
    backgroundColor = UIColor.clearColor()
}
```

Per Swift's rules of initialization, now that you've overridden one initializer, you must implement all required initializers. Implement `init(coder:)` which is called when a view is created from an interface file such as a storyboard.

```
required init(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
}
```

Build and run the application. Figure 2.18 shows the clear background and the resulting circle.

Figure 2.18 **HypnosisView** with clear background



Drawing concentric circles

There are two approaches you can take to draw multiple concentric circles inside the **HypnosisView**. You can create multiple instances of **UIBezierPath**, each one representing one circle. Or you can add multiple circles to the single instance of **UIBezierPath**, and each circle will be a sub-path. Although it is slightly more efficient to use one instance, you'll be adding code in the next chapter for different circles to have different colors, and that isn't possible with one **UIBezierPath**.

To fill the screen with concentric circles, you need to determine the radius of the outermost circle. You will start drawing a circle with a minimum radius and then draw circles with an increasing radius for as long as the radius is smaller than the radius of the outermost circle.

For the maximum radius, you are going to use half of the hypotenuse of the entire view. This means that the outermost circle will nearly circumscribe the view, and you will only see bits of purple in the corners.

In `HypnosisView.swift`, replace the code that draws one circle with code that draws concentric circles.

```
override func drawRect(rect: CGRect)
{
    let bounds = self.bounds

    // Figure out the center of the bounds rectangle
    let centerX = bounds.origin.x + bounds.size.width / 2.0
    let centerY = bounds.origin.y + bounds.size.height / 2.0
    let center = CGPointMake(x: centerX, y: centerY)

    // The circle will be the largest that will fit in the view
let radius = min(bounds.size.width / 2.0, bounds.size.height / 2.0)

    // The largest circle will circumscribe the view
    let maxRadius =
        CGFloat(hypot(CDouble(bounds.size.width), CDouble(bounds.size.height)) / 2.0)

    // Keep drawing bigger circles until the radius is
    // larger than the maximum visible radius
    for var radius: CGFloat = 0.0; radius < maxRadius; radius += 20 {
        let path = UIBezierPath()

        // Add an arc to the path at center, with radius of radius,
        // from 0 to 2*PI radians (a circle)
        path.addArcWithCenter(center,
            radius: radius,
            startAngle: 0,
            endAngle: CGFloat(M_PI * 2.0),
            clockwise: true)

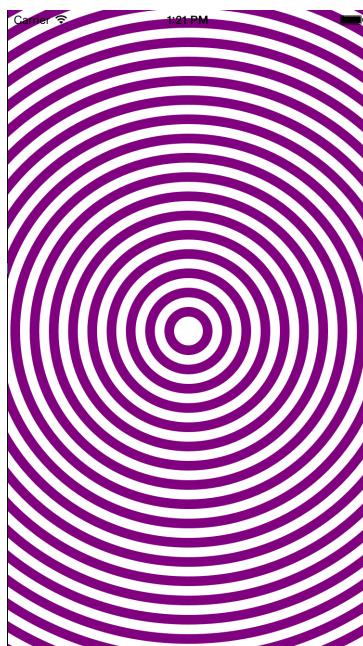
        // Configure line width to 10 points
        path.lineWidth = 10;

        // Configure the drawing color to purple
        UIColor.purpleColor().setStroke()

        // Draw the line
        path.stroke()
    } // Don't miss this trailing brace!
}
```

Build and run the application. You should now see concentric circles.

Figure 2.19 **HypnosisView** drawing concentric circles



You have seen only a sampling of what **UIBezierPath** can do. Be sure to check out the documentation and try some of the challenges at the end of this chapter to get a better feel for some of the clever things you can do by stringing together arcs, lines, and curves.

Redrawing Views

Let's take a look at how views are redrawn in response to an event. You will update Hypnosister so that it appears that the circles are radiating out from the center of the view. To accomplish this, you will add a timer to the **HypnosisView** that will increase a radius offset for the circles 30 times a second. A change in radius offset which will require the view to redraw itself.

The first step is to declare two properties to **HypnosisView**: one for the radius offset and another for the timer.

Open **HypnosisView.swift** and add the following code near the top of the file.

```
class HypnosisView: UIView {
    var radiusOffset: CGFloat = 0
    var timer: NSTimer?
```

Two properties have been added to the **HypnosisView** class. The first is a **CGFloat** variable named **radiusOffset** that is initialized with a default value of 0. The second property is an optional instance of **NSTimer**; we'll discuss why it's declared as optional soon.

Timers have keep a strong reference to their target, so to avoid a strong ownership cycle, you will create and start the time when the view is added as a subview and then stop and destroy the timer when it is removed from its superview. When a view is added as a subview, it gets sent the message **didMoveToSuperview()**; when it is removed from its superview, views get sent the message **removeFromSuperview()**. In **HypnosisView.swift**, override these two methods to set up and tear down the timer.

```
override func didMoveToSuperview() {
    if superview != nil {
        timer = NSTimer.scheduledTimerWithTimeInterval(1.0/30.0,
            target: self,
            selector: "timerFired:",
            userInfo: nil,
            repeats: true)
    }
}

override func removeFromSuperview() {
    timer?.invalidate()
    timer = nil

    super.removeFromSuperview()
}
```

This code creates an instance of **NSTimer** that fires 30 times a second. When it fires, it send a message to its target (the **HypnosisView**), and it calls the method identified by the selector (**timerFired(_:)**).

Since the timer is not initialized in **init**, it must be an optional variable.

In **HypnosisView.swift**, implement the **timerFired(_:)** method.

```
func timerFired(timer: NSTimer) {
    println("pew")

    // Increment the radius offset
    radiusOffset += 1.0

    // Reset the radius offset at 20 to create a nice, looping effect
    if radiusOffset > 20 {
        radiusOffset = 0
    }
}
```

Next, update `drawRect(_:_)` to account for the radius offset. Also change the circle color to be more transparent the smaller its radius is.

```
override func drawRect(rect: CGRect)
{
    ...

    for var radius = 0.0; radius < maxRadius; radius += 20 {
        let path = UIBezierPath()

        // Add an arc to the path at center, with radius of radius,
        // from 0 to 2*PI radians (a circle)
        path.addArcWithCenter(center,
            radius: radius + radiusOffset, // Add the radius offset to the current radius
            startAngle: 0,
            endAngle: CGFloat(M_PI * 2.0),
            clockwise: true)

        // Configure line width to 10 points
        path.lineWidth = 10;

        // Configure the drawing color to purple
        UIColor.purpleColor().setStroke()
        let alpha = ((radius + radiusOffset - 10) / maxRadius)
        UIColor.purpleColor().colorWithAlphaComponent(alpha).setStroke()

        // Draw the line
        path.stroke()
    }
}
```

Build and run the application. It is "pew"ing to the console, but the circles are static - the view is not visually changing at all. The reason is because the view is not being redrawn. To understand why and how to fix the problem, you need to know about the run loop.

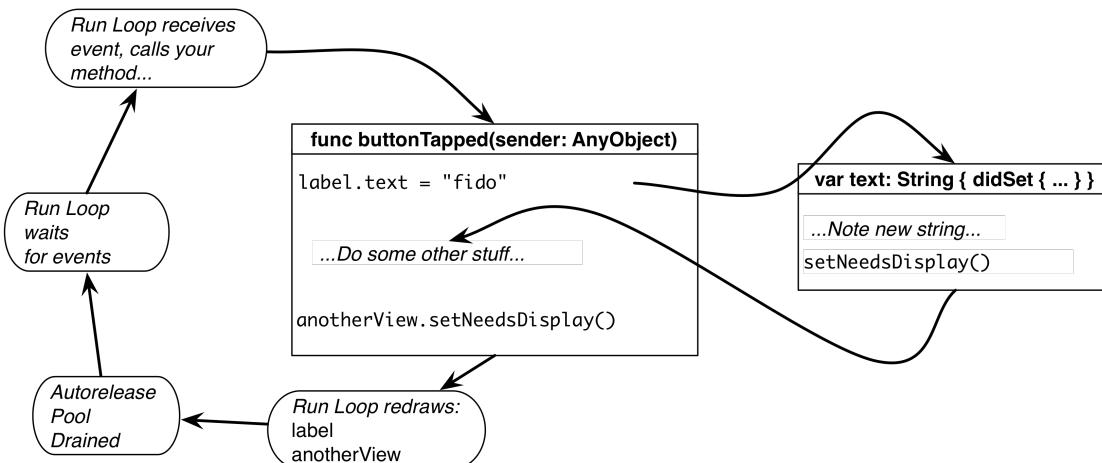
The Run Loop

When an iOS application is launched, it starts a *run loop*. The run loop's job is to listen for events, such as a timer firing. When an event occurs, the run loop then finds the appropriate handler methods for the event. Those handler methods call other methods, which call more methods, and so on. Once all of the methods have completed, control returns to the run loop.

When the run loop regains control, it checks a list of “dirty views” – views that need to be re-rendered based on what happened in the most recent round of event handling. The run loop then sends the `drawRect(_:_)` message to the views in this list before all of the views in the hierarchy are composited together again.

Figure 2.20 shows where redrawing the screen happens in the run loop using an example of the user entering text into a text field.

Figure 2.20 Redrawing views with the run loop



These two optimizations – only re-rendering views that need it and only sending `drawRect(_:)` once per event – keep iOS interfaces responsive. If iOS applications had to redraw every view every time an event was processed, there would be a lot of time wasted doing unnecessary work. Batching the redrawing of views at the end of a run loop cycle prevents needlessly redrawing a view more than once if more than one of its properties is changed in a single event.

Let's look at what is happening in Hypnosister. You know that the timer event is being routed correctly to the `HypnosisView` because you see your log message. But when `timerFired(_:)` finishes executing and control returns to the run loop, the run loop does not send `drawRect(_:)` to the `HypnosisView`.

To get a view on the list of dirty views, you must send it the message `setNeedsDisplay()`. The subclasses of `UIView` that are part of the iOS SDK send themselves `setNeedsDisplay()` whenever their content changes. For example, an instance of `UILabel` will send itself `setNeedsDisplay()` when its `text` property changes, since changing the text of a label requires the label to re-render its layer. In custom `UIView` subclasses, like `HypnosisView`, you must send this message yourself.

In `HypnosisView.swift`, implement a custom `didSet` observer for the `radiusOffset` property in order to send `setNeedsDisplay()` to the view whenever this property is changed.

```
var radiusOffset: CGFloat = 0 {
    didSet {
        setNeedsDisplay()
    }
}
```

Build and run the application again. You should see the circles expanding, providing a very hypnotic effect.

(There is another possible optimization when redrawing: you can mark only a portion of a view as needing to be redrawn. This is done by sending `setNeedsDisplayInRect(rect:)` to a view. When `drawRect(_:)` is sent to the dirty view, the argument to this method that we have been ignoring the whole time will be the rectangle passed to `setNeedsDisplayInRect(rect:)`. Overall, you do not gain that much performance and you end up doing some difficult work to get this partial redrawing behavior to work right, so most people do not bother with it unless their drawing code is obviously slowing the app down.)

More Developer Documentation

The API Reference, which contains the class references, is an essential part of the developer documentation and an essential part of a developer's life. But there is more to the documentation than the API Reference. The documentation also provides:

SDK Guides	organized by topic rather than by class or method and excellent for learning more about specific topics in Objective-C and iOS development
Sample Code	small, complete projects that demonstrate how Apple expects the class in question to be used

It would be difficult to overstate how important Apple's documentation is to the daily work of iOS developers. As you go through this book, take a moment to look up new classes and methods as you encounter them and see what else they can do. Also read through SDK guides and download sample code projects that pique your interest. You can see the available guides and sample code in the iOS Developer Library at developer.apple.com/library.

Bronze Challenge: Draw an Image

The challenge is to load an image from the filesystem and draw it on top of the concentric circles, as in Figure 2.21.

Figure 2.21 Drawing an Image



Find an image file. A PNG with some transparent parts would be especially interesting. (The zip file you downloaded has `logo.png` that will work nicely.) Drag it into your Xcode project.

Creating a **UIImage** object from that file is one line:

```
let logoImage: UIImage? = UIImage(named: "logo.png")
```

In your **drawRect(_:)** method, compositing it onto your view is just one more:

```
logoImage?.drawInRect(someRect)
```

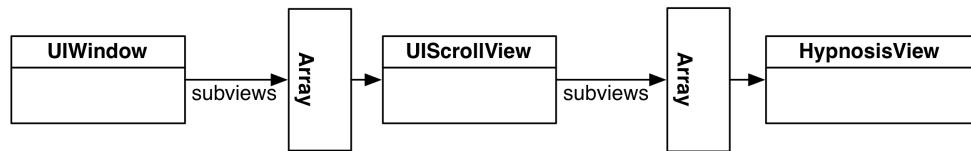
3

UIScrollView and Delegation

Scroll views are typically used for views that are larger than the screen. For example, when viewing a web page in Safari, the whole page is larger than the screen itself, but you are able to move around and zoom to see the entire site. This is accomplished through the use of **UIScrollView**.

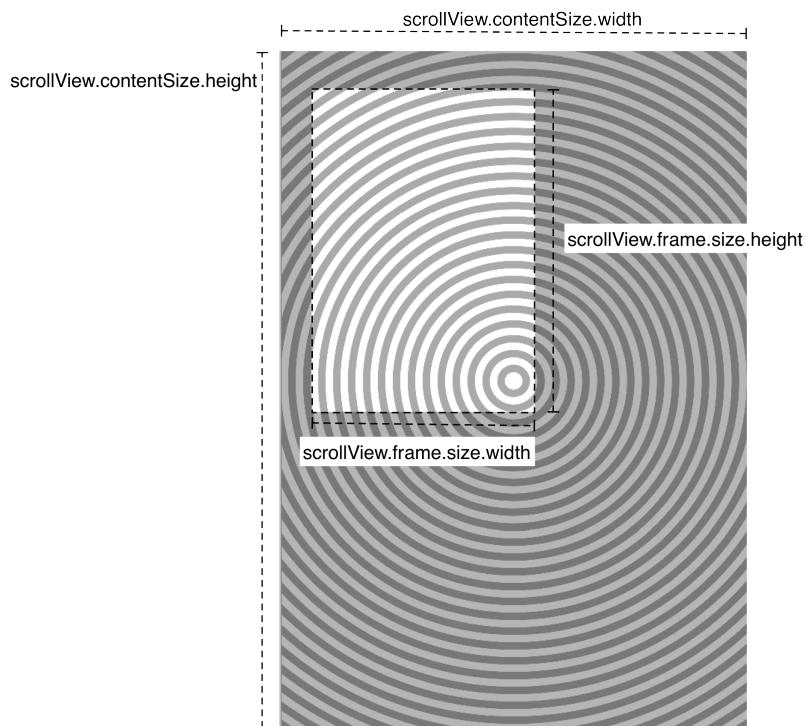
In this chapter, you are going to add an instance of **UIScrollView** to Hypnosister. This scroll view will be a direct subview of the window, and the instance of **HypnosisView** will be a subview of the scroll view, as shown in Figure 3.1.

Figure 3.1 View hierarchy with **UIScrollView**



Using UIScrollView

A scroll view draws a rectangular portion of its subview, and moving your finger, or *panning*, on the scroll view changes the position of that rectangle on the subview. Thus, you can think of the scroll view as a viewing port that you can move around (Figure 3.2). The size of the scroll view is the size of this viewing port. The size of the area that it can be used to view is the **UIScrollView**'s **contentSize**, which is typically the size of the **UIScrollView**'s subview.

Figure 3.2 **UIScrollView** and its content area

UIScrollView is a subclass of **UIView**, so it can be initialized using `init(frame:)` and it can be added as a subview to another view.

In `AppDelegate.swift`, put a super-sized version of **HypnosisView** inside a scroll view and add that scroll view to the window:

```
func application(application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
    // Override point for customization after application launch.
    window = UIWindow(frame: UIScreen.mainScreen().bounds)

    let firstFrame = CGRectMake(x: 50, y: 50, width: 100, height: 100)
    let firstView = UIView(frame: firstFrame)
    window!.addSubview(firstView)

    // Create CGRects for frames
    var screenRect = window!.bounds
    var bigRect = screenRect
    bigRect.size.width *= 2.0
    bigRect.size.height *= 2.0

    // Create a screen-sized scroll view and add it to the window
    let scrollView = UIScrollView(frame: screenRect)
    window!.addSubview(scrollView)

    // Create a super-sized hypnosis view and add it to the scroll view
    let hypnosisView = HypnosisView(frame: bigRect)
    scrollView.addSubview(hypnosisView)

    // Tell the scroll view how big its content area is
    scrollView.contentSize = bigRect.size

    window!.backgroundColor = UIColor.whiteColor()
    window!.makeKeyAndVisible()
    return true
}
```

Build and run your application. You can pan your view up and down, left and right to see more of the super-sized **HypnosisView**.

Figure 3.3 Top right quadrant of big **HypnosisView**



In this section, you will add another subview to the window. It will be a mini-map that shows what section of the scroll view you are currently viewing.

Figure 3.4 **HypnosisView** with Mini Map



The mini-map will be a **UIView** subclass. It will have a method that takes in a **UIScrollView** and will compute the relative values of the view port. It will then draw the square that represents these values.

Create a new **UIView** subclass named **MiniMapView**. Open **MiniMapView.swift** and give the view a semi-transparent, light gray background color. Also remove the stubbed out **drawRect(_:)** method.

```
class MiniMapView: UIView {

    override init(frame: CGRect) {
        super.init(frame: frame)

        backgroundColor = UIColor.lightGrayColor().colorWithAlphaComponent(0.7)
    }

    required init(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)
    }

    /*
    // Only override drawRect: if you perform custom drawing.
    // An empty implementation adversely affects performance during animation.
    override func drawRect(rect: CGRect) {
        // Drawing code
    }
    */
}

}
```

Now add four variables to represent the relative values for x, y, width and height. Give each of these an default value of `0.0`.

```
class MiniMapView: UIView {

    var relativeX: CGFloat = 0.0
    var relativeY: CGFloat = 0.0
    var relativeWidth: CGFloat = 0.0
    var relativeHeight: CGFloat = 0.0
}
```

Each of these values will represent the relative corresponding value - in other words, represented as a percentage. For example, if the scroll view's content size width is 100 and it's `contentOffset` has an x of 50, the `relativeX` would be `0.5`.

Just as with the `radiusOffset` property on `HypnosisView`, if any of these values are changed, `MiniMapView` needs to redraw itself. Add a `didSet` observer for each property to call `setNeedsDisplay()`.

```
var relativeX: CGFloat = 0.0 {
    didSet {
        setNeedsDisplay()
    }
}
var relativeY: CGFloat = 0.0 {
    didSet {
        setNeedsDisplay()
    }
}
var relativeWidth: CGFloat = 0.0 {
    didSet {
        setNeedsDisplay()
    }
}
var relativeHeight: CGFloat = 0.0 {
    didSet {
        setNeedsDisplay()
    }
}
```

Implement `drawRect(_:)` to draw a rectangle based on these properties.

```

override func drawRect(rect: CGRect) {
    // Drawing code

    let bounds = self.bounds

    // Compute the actual values for the rectangle
    let x = bounds.width * relativeX
    let y = bounds.height * relativeY
    let width = bounds.width * relativeWidth
    let height = bounds.height * relativeHeight

    // Create the rectangle
    let path = UIBezierPath()
    path.moveToPoint(CGPoint(x: x, y: y))
    path.addLineToPoint(CGPoint(x: x+width, y: y))
    path.addLineToPoint(CGPoint(x: x+width, y: y+height))
    path.addLineToPoint(CGPoint(x: x, y: y+height))
    path.closePath()

    // Fill the rectangle with a semi-transparent gray color
    UIColor.grayColor().colorWithAlphaComponent(0.8).setFill()
    path.fill()
}

```

Finally, implement a new method named `updateWithScrollView(_:)` to populate the relative values.

```

func updateWithScrollView(sv: UIScrollView) {
    relativeX = sv.contentOffset.x / sv.contentSize.width
    relativeY = sv.contentOffset.y / sv.contentSize.height
    relativeWidth = sv.bounds.width / sv.contentSize.width
    relativeHeight = sv.bounds.height / sv.contentSize.height
}

```

Setting the values for each property will call `setNeedsDisplay()`. Even though it will be called four times from this method, the view will only get redrawn once; since the calls to `setNeedsDisplay()` occur within the same tick of the run loop, they will be batched together into one redraw.

With the `MiniMapView` class complete, let's add it to the window and have it update using the scroll view that was created earlier. Open `AppDelegate.swift` and add an instance of `MiniMapView` to the window.

```

...
// Tell the scroll view how big its content area is
scrollView.contentSize = bigRect.size

let miniMap = MiniMapView(frame: CGRect(x: 10, y: 30, width: 75, height: 135))
window!.addSubview(miniMap)
miniMap.updateWithScrollView(scrollView)

...

```

Build and run the app. You will see the mini-map in the upper left hand corner of the window, and you'll see the darker shaded rectangle corresponding to the current view port. Unfortunately when you scroll around, the mini-map does not update. To get the mini-map updating as you scroll, you need to know about *delegation*.

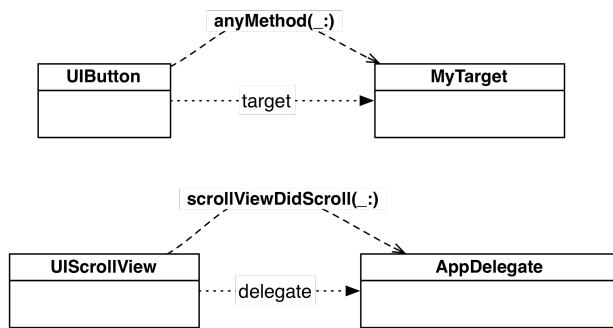
Delegation

Delegation is an object-oriented approach to *callbacks*. A callback is a function that is supplied in advance of an event and is called every time the event occurs. Some objects need to make a callback for more than one event. For instance, the scroll view wants to “callback” when it finds a new location and when it encounters an error.

However, there is no built-in way for two (or more) callback functions to coordinate and share information. This is the problem addressed by delegation – you supply a single delegate to receive all of the event messages for a particular object. This delegate object can then store, manipulate, act on, and relay the related information as it sees fit.

Let's compare delegation with another object-oriented approach to callbacks: target-action pairs. You used target-action pairs with the **UIButtons** in your Quiz application from Chapter 1. In a target-action pair, you have a target object that you send an action message to when a particular event occurs (like a button tap). A new target-action pair must be created for each distinct event (like a tap, a double tap, or a long press). With delegation, you set the delegate once and then can send it messages for many different events. The delegate will implement the methods that correspond to the events it wants to hear about (Figure 3.5).

Figure 3.5 Target-action vs. delegation

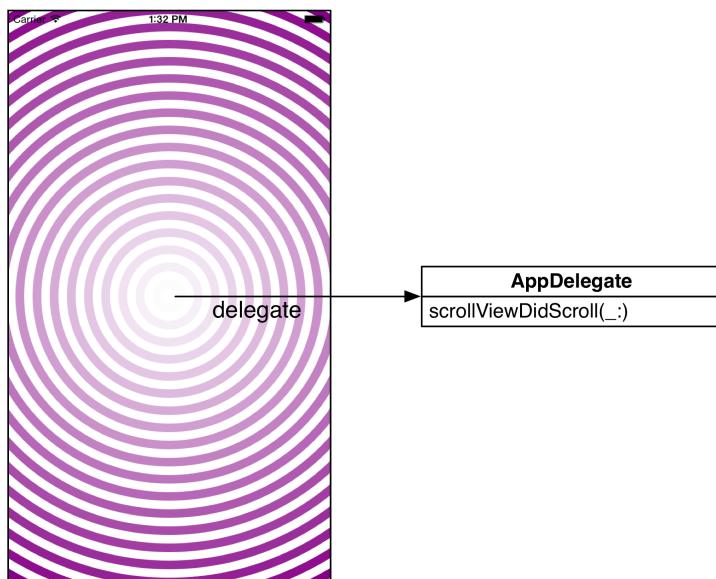


Also, with a target-action pair, the action message can be any message. In delegation, an object can only send its delegate messages from a specific set listed in a *protocol*.

When a scroll view scrolls, it calls the method `scrollViewDidScroll(_ :)` to its *delegate*. What object is the scroll view's delegate? We get to decide.

Every **UIScrollView** has a *delegate* property, and we can set this property to point to the object that we want to receive the “scroll view did scroll” method. For Hypnosister, this object is the **AppDelegate**.

Figure 3.6 AppDelegate as UIScrollView delegate



In **AppDelegate.swift**, set the *delegate* property of the scroll view to be the instance of **AppDelegate**.

```

// Create a screen-sized scroll view and add it to the window
let scrollView = UIScrollView(frame: screenRect)

// There will be an error from this line of code; ignore it for now
scrollView.delegate = self

window!.addSubview(scrollView)

```

Because the **UIScrollView** calls the `scrollViewDidScroll(_ :)` method to its delegate, you must implement this method in **AppDelegate.swift**. For now, just have this method print the content offset of the scroll view.

(Be very careful that there are no typos or capitalization errors or the method won't be called. The name of the method that you implement must exactly match the name of the method that the scroll view expects.)

```
func scrollViewDidScroll(scrollView: UIScrollView) {
    println("Content offset: \(scrollView.contentOffset)")
}
```

You can't run the application right now due to the error when setting the delegate. This is because your class must explicitly agree to the contract of the `UIScrollViewDelegate protocol`.

Protocols

For every object that can have a delegate, there is a corresponding *protocol* that declares the messages that the object can send its delegate. The delegate implements methods from the protocol for events it is interested in. When a class implements methods from a protocol, it is said to *conform to* the protocol.

The protocol for `UIScrollView`'s delegate looks like this:

```
// Note that we omitted a few methods from
// the real declaration of this protocol
// for brevity's sake

protocol UIScrollViewDelegate : NSObjectProtocol {
    optional func scrollViewDidScroll(scrollView: UIScrollView)
    optional func scrollViewDidZoom(scrollView: UIScrollView)
    optional func scrollViewWillBeginDragging(scrollView: UIScrollView)
    optional func viewForZoomingInScrollView(scrollView: UIScrollView) -> UIView?
}
```

This protocol, like all protocols, is declared with `protocol` followed by its name, `UIScrollViewDelegate`. The `NSObjectProtocol` after the colon refers to the `NSObject` protocol and tells us that `UIScrollViewDelegate` inherits all of the methods in the `NSObject` protocol. The methods specific to `UIScrollViewDelegate` are declared next.

Note that a protocol is not a class; it is simply a list of methods. You cannot create instances of a protocol, it cannot have instance variables, and these methods are not implemented anywhere in the protocol. Instead, implementation is left to each class that conforms to the protocol.

We call protocols used for delegation *delegate protocols*, and the naming convention for a delegate protocol is the name of the delegating class plus the word `Delegate`. Not all protocols are delegate protocols, however, and you will see an example of a different kind of protocol in Chapter 9.

The protocols we have mentioned so far are part of the iOS SDK, but you can also write your own protocols.

Protocol methods

In the `UIScrollViewDelegate` protocol, there are two kinds of methods: methods that handle information updates and methods that handle requests for input. For example, the scroll view's delegate implements the `scrollViewDidScroll(_)` method if it wants to hear from the scroll view that the user has scrolled. This is an information update.

On the other hand, `viewForZoomingInScrollView(_)` is a request for input. A scroll view calls this method on its delegate to ask what subview it should scale when zooming is about to occur. The method returns a `UIView`, which is the delegate's answer.

Methods declared in a protocol can be required or optional. By default, protocol methods are required. If a protocol has optional methods, these are preceded by the directive `optional`. Looking back at the `UIScrollViewDelegate` protocol, you can see that all of its methods are optional. This is typically true of delegate protocols.

Before sending an optional message, the object first asks its delegate if it is okay to call that method by calling another method, `respondsToSelector(aSelector:)`. Every object implements this method, which checks at runtime whether an object implements a given method. For example, `UIScrollView` could implement a method that looks like this:

```
func scrollViewScrolled {
    // If the delegate is set and implements
    // the scrollViewScrolled method, call
    // that method
    if delegate?.respondsToSelector("scrollViewScrolled:") {
        // Force the optional method to be called with the !
        delegate?.scrollViewScrolled!(self)
    }

    // Alternatively, this can be done with optional chaining
    // which will call respondsToSelector automatically
    delegate?.scrollViewScrolled?(self)
}
```

If a method in a protocol is required, then the method will be called without checking first.

To prevent this from happening, the compiler will insist that a class implement the required methods in a protocol. But, for the compiler to know to check for implementations of a protocol's required methods, the class must explicitly state that it conforms to a protocol. This is done in the class declaration: the protocols that a class conforms to are added to a comma-delimited list following the superclass (if there is a superclass).

In `AppDelegate.swift`, declare that `AppDelegate` conforms to the `UIScrollViewDelegate` protocol.

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate, UIScrollViewDelegate {
```

Build and run the application. Now that you have declared that `AppDelegate` conforms to the `UIScrollViewDelegate` protocol, the error from the line of code where you set the delegate of the `scrollView` disappears. Furthermore, if you want to implement additional methods from the `UIScrollViewDelegate` protocol in `AppDelegate`, those methods will now be auto-completed by Xcode. As you scroll around, the content offset will print to the console. Let's have the mini-map update as you scroll around.

Since the mini-map needs to be updated in `scrollViewDidScroll(_:)`, you need a property that references it in `AppDelegate`. Add this property to `AppDelegate.swift`.

```
class AppDelegate: UIResponder, UIApplicationDelegate, UIScrollViewDelegate {

    var window: UIWindow?
    var miniMap: MiniMapView?
```

Then set the property on `AppDelegate` after instantiating the `MiniMapView`.

```
let miniMap = MiniMapView(frame: CGRect(x: 10, y: 30, width: 75, height: 135))
window!.addSubview(miniMap)
miniMap.updateWithScrollView(scrollView)
self.miniMap = miniMap
```

Since the property and the local variable have the same name, you need to use `self` to differentiate the two.

Finally, update `scrollViewDidScroll(_:)` to update the mini-map.

```
func scrollViewDidScroll(scrollView: UIScrollView) {
    println("Content offset: \(scrollView.contentOffset)")
    miniMap?.updateWithScrollView(scrollView)
}
```

Build and run the application. As you scroll around, the mini-map will now update.

Zooming

A `UIScrollView` can zoom in and out on its content. To zoom, a scroll view needs to know the minimum and maximum zoom levels, and it needs to know the view to zoom in on.

In `AppDelegate.swift`, set the zoom properties of the `UIScrollView`.

```
// Create a screen-sized scroll view and add it to the window
let scrollView = UIScrollView(frame: screenRect)

scrollView.minimumZoomScale = 1.0
scrollView.maximumZoomScale = 2.0

scrollView.delegate = self

window!.addSubview(scrollView)
```

Build and run the application. You should see the **HypnosisView**, which you cannot zoom.

To zoom, you must implement the method `viewForZoomingInScrollView(_:)` in the **UIScrollView**'s delegate. This method returns the instance of the view to zoom in on. The view will be the instance of **HypnosisView**, and the **UIScrollView**'s delegate will be the **AppDelegate**.

In order to return the **HypnosisView** from `viewForZoomingInScrollView(_:)`, you need a property that references that view. Add this variable to **AppDelegate**.

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate, UIScrollViewDelegate {

    var window: UIWindow?
    var miniMap: MiniMapView?
    var hypothesisView: HypnosisView?
```

Then set this property after you create the **HypnosisView**.

```
// Create a super-sized hypnosis view and add it to the scroll view
let hypothesisView = HypnosisView(frame: bigRect)

// Set the property to reference the local variable
self.hypothesisView = hypothesisView

scrollView.addSubview(hypothesisView)
```

Finally, implement `viewForZoomingInScrollView(_:)` in **AppDelegate.swift** to return this view.

```
func viewForZoomingInScrollView(scrollView: UIScrollView) -> UIView? {
    return hypothesisView
}
```

Build and run the application. Pinch and pull with two fingers to zoom in and out. (On the simulator, you can simulate two fingers by holding down the Option key, clicking, and then moving the mouse.) Pan with one finger to scroll once you have zoomed in.

For the More Curious: Panning and paging

Another use for a scroll view is panning between a number of view instances.

In **AppDelegate.swift**, shrink the **HypnosisView** back to the size of the screen and add a second screen-sized **HypnosisView** as another subview of the **UIScrollView**. Set the scroll view's `contentSize` to be twice as wide as the screen, but the same height.

```
// Create CGRects for frames
var screenRect = window!.bounds
var bigRect = screenRect
bigRect.size.width *= 2.0
bigRect.size.height *= 2.0

// Create a screen-sized scroll view and add it to the window
let scrollView = UIScrollView(frame: screenRect)
window!.addSubview(scrollView)

// Create a super-sized hypnosis view and add it to the scroll view
// let hypothesisView = HypnosisView(frame: bigRect)

// Create a screen-sized hypnosis view and add it to the scroll view
let hypothesisView = HypnosisView(frame: screenRect)

scrollView.addSubview(hypothesisView)

// Add a second screen-sized hypnosis view just off screen to the right
screenRect.origin.x += screenRect.size.width
let anotherView = HypnosisView(frame: screenRect)
scrollView.addSubview(anotherView)

// Tell the scroll view how big its content area is
scrollView.contentSize = bigRect.size;
```

Build and run the application. Pan from left to right to see each instance of **HypnosisView**. Notice that you can stop in between the two views.

Figure 3.7 In between the two hypnosis views



Sometimes you want this, but other times, you do not. To force the scroll view to snap its viewing port to one of the views, turn on paging for the scroll view in `AppDelegate.swift`.

```
let scrollView = UIScrollView(frame: screenRect)
scrollView.pagingEnabled = true
```

Build and run the application. Pan to the middle of two views and notice how it snaps to one or the other view. Paging works by taking the size of the scroll view's bounds and dividing up the `contentSize` it displays into sections of the same size. After the user pans, the view port will scroll to show only one of these sections.

4

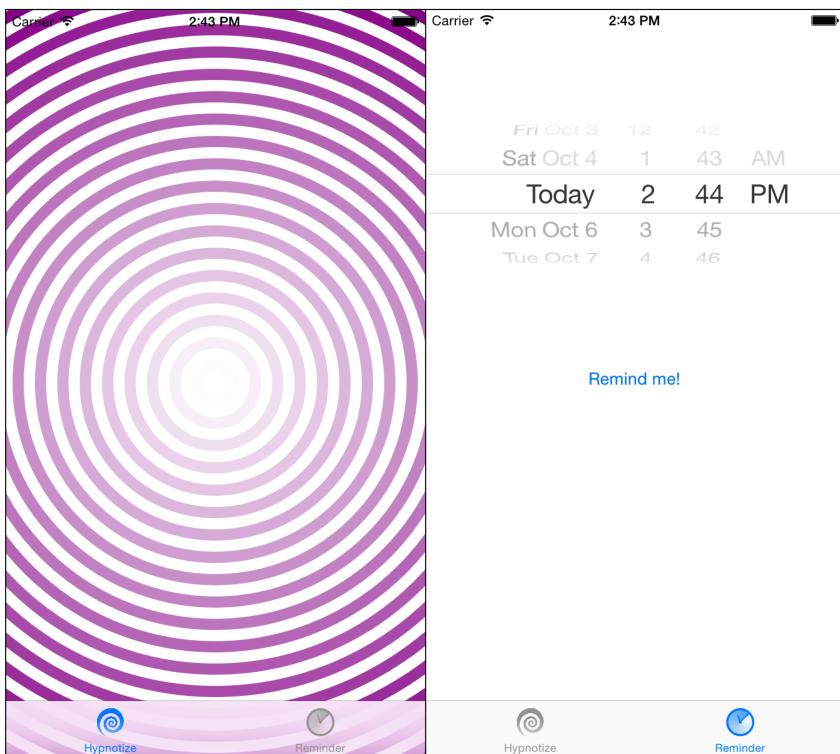
View Controllers

In Chapter 3, you created a view hierarchy (a scroll view with two subviews) and presented it on screen by explicitly adding the scroll view as a subview of the application’s window. It is more common to do this using a *view controller*.

A view controller is an instance of a subclass of **UIViewController**. A view controller manages a view hierarchy. It is responsible for creating view objects that make up the hierarchy, for handling events associated with the view objects in its hierarchy, and for adding its hierarchy to the window.

In this chapter, you will change Hypnosister to use view controllers. The user will be able to switch between two view hierarchies – one for viewing the **HypnosisView** that you created in ??? and the other for setting a reminder for hypnosis on a future date.

Figure 4.1 The two faces of Hypnosister

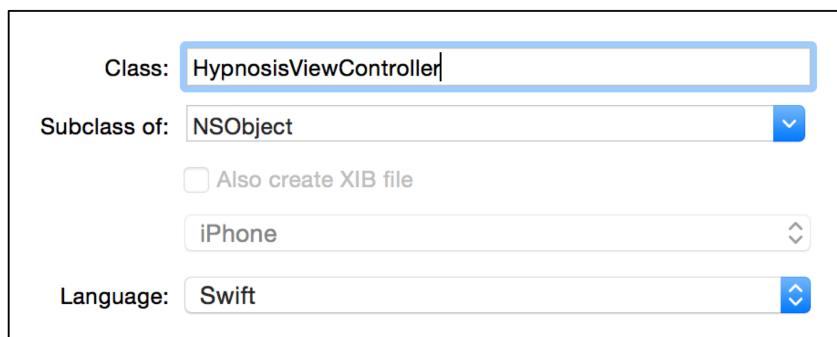


To make this happen, you are going to create two **UIViewController** subclasses: **HypnosisViewController** and **ReminderViewController**. You will use the **UITabBarController** class to allow the user to switch between the view hierarchies of the two view controllers.

Subclassing UIViewController

From the File menu, select New → File... From the iOS section, select Source and then choose Cocoa Touch Class. Click Next.

Name this class **HypnosisViewController** and choose **NSObject** as its superclass (Figure 4.2). Make sure the Language has Swift selected. Click Next and save the files to finish creating the class.

Figure 4.2 Creating **HypnosisViewController**

You created the class with the **NSObject** template to start with the simplest template possible. By starting simple, you get the chance to see how the pieces work together.

Open **HypnosisViewController.swift** and change the superclass to **UIViewController**.

```
class HypnosisViewController: NSObject {
class HypnosisViewController: UIViewController {
}
```

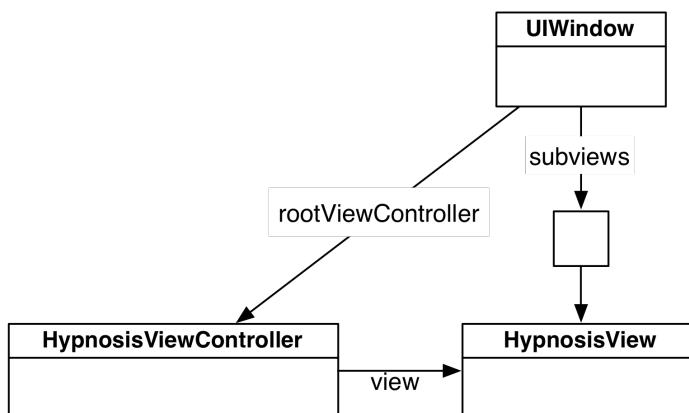
The view of a view controller

As a subclass of **UIViewController**, **HypnosisViewController** inherits an important property:

```
var view: UIView
```

This property points to a **UIView** instance that is the root of the view controller's view hierarchy. When the **view** of a view controller is added as a subview of the window, the view controller's entire view hierarchy is added.

Figure 4.3 Object diagram for HypnoNerd



A view controller's view is not created until it needs to appear on the screen. This optimization is called *lazy loading*, and it can often conserve memory and improve performance.

There are two ways that a view controller can create its view hierarchy:

- programmatically, by overriding the **UIViewController** method **loadView()**.
- in Interface Builder, by loading a NIB file. (Recall that a NIB file is the file that gets loaded and the XIB file is what you edit in Interface Builder.)

Because the view hierarchy of **HypnosisViewController** consists of only one view, it is a good candidate for being created programmatically.

Creating a view programmatically

Open **HypnosisViewController.swift** and override **loadView** to create a screen-sized instance of **HypnosisView** and set it as the **view** of the view controller.

```

class HypnosisViewController: UIViewController {

    override func loadView() {
        // Create a view
        let frame = UIScreen.mainScreen().bounds
        let backgroundView = HypnosisView(frame: frame)

        // Set it as *the* view of this view controller
        view = backgroundView
    }
}

```

When a view controller is created, its `view` property is `nil`. If a view controller is asked for its `view` and its `view` is `nil`, then the view controller is sent the `loadView` message.

The next step is to add the view hierarchy of the `HypnosisViewController` to the application window so that it will appear on screen to users.

Setting the root view controller

There is a convenient property for adding a view controller's view hierarchy to the window: `UIWindow`'s `rootViewController` property. Setting a view controller as the `rootViewController` adds that view controller's `view` as a subview of the window. It also automatically resizes the `view` to be the same size as the window.

In `AppDelegate.swift`, remove the code that programmatically adds views to the window.

```

func application(application: UIApplication!,
    didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
    // Override point for customization after application launch.
    window = UIWindow(frame: UIScreen.mainScreen().bounds)

    // Create CGRects for frames
    var screenRect = window!.bounds
    var bigRect = screenRect
    bigRect.size.width *= 2.0
    bigRect.size.height *= 2.0

    // Create a screen sized scroll view and add it to the window
    let scrollView = UIScrollView(frame: screenRect)

    scrollView.minimumZoomScale = 1.0
    scrollView.maximumZoomScale = 2.0

    scrollView.delegate = self

    window!.addSubview(scrollView)

    // Create a super sized hypnosis view and add it to the scroll view
    let hypnosisView = HypnosisView(frame: bigRect)
    scrollView.addSubview(hypnosisView)

    // Tell the scroll view how big its content area is
    scrollView.contentSize = bigRect.size

    let miniMap = MiniMapView(frame: CGRectMake(x: 10, y: 30, width: 75, height: 135))
    window!.addSubview(miniMap)
    miniMap.updateWithScrollView(scrollView)
    self.miniMap = miniMap

    window!.backgroundColor = UIColor.whiteColor()
    window!.makeKeyAndVisible()
    return true
}

```

Then create an instance of `HypnosisViewController` and set it as the `rootViewController` of the window.

```
func application(application: UIApplication!,  
    didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {  
    // Override point for customization after application launch.  
    window = UIWindow(frame: UIScreen.mainScreen().bounds)  
  
    let hvc = HypnosisViewController()  
    window!.rootViewController = hvc  
  
    window!.backgroundColor = UIColor.whiteColor()  
    window!.makeKeyAndVisible()  
    return true  
}
```

The view of the root view controller appears at the start of the run of the application. Thus, the window asks for it when setting the view controller as its `rootViewController`.

Given what you learned in Chapter 2, you can imagine what the core of `rootViewController's didSet` property observer might look like:

```
var rootViewController: UIViewController? {  
    didSet {  
        if let rootVC = rootViewController {  
            // Get the view of the root view controller  
            let rootView = rootVC.view  
  
            // Make a frame that fits the window's bounds  
            let viewFrame = bounds  
            rootView.frame = viewFrame;  
  
            // Insert this view as window's subview  
            addSubview(rootView)  
        }  
    }  
}
```

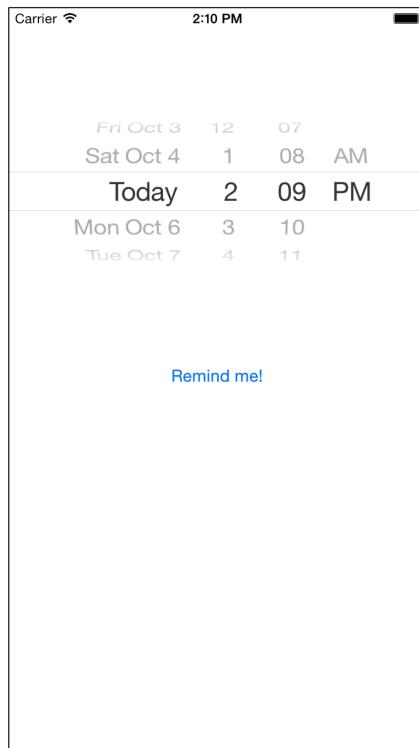
At the beginning of this implementation, the `HypnosisViewController` is asked for its `view`. Because the `HypnosisViewController` has just been created, its `view` is `nil`. So it is sent the `loadView()` message that creates its `view`.

Build and run the application. Hypnosister doesn't look like it has changed a whole lot. Under the hood, however, it is quite different. You are using a view controller to present the `HypnosisView` instead of adding the view object itself to the window. This adds a layer of complexity, which, as you will see by the end of the chapter, gives you power and flexibility to do neat things.

Another `UIViewController`

In this section, you are going to create the `ReminderViewController` class. Eventually, this view controller will enable the user to pick a date to receive a reminder to be hypnotized. This reminder will take the form of a notification that will appear even if Hypnosister is not running at the time.

Figure 4.4 **ReminderViewController**



Create a new Cocoa Touch Class (Command-N). Name it **ReminderViewController** and make it a subclass of **NSObject**.

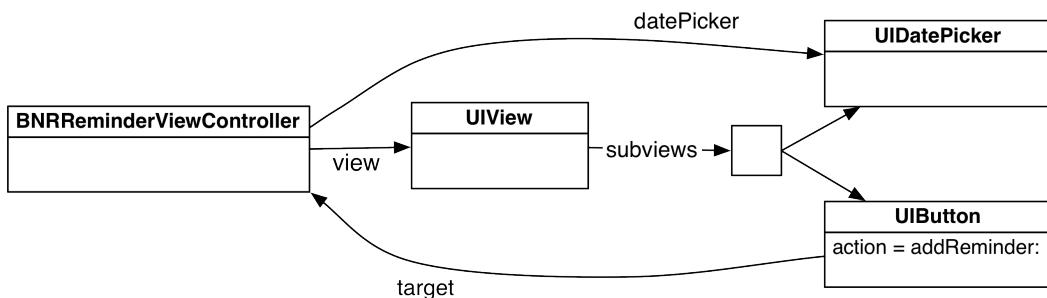
In **ReminderViewController.swift**, change the superclass to **UIViewController**.

```
class ReminderViewController: NSObject {
class ReminderViewController: UIViewController {

}
```

The **ReminderViewController**'s view will be a full-screen **UIView** with two subviews – an instance of **UIDatePicker** and an instance of **UIButton** (Figure 4.5).

Figure 4.5 Object diagram of **ReminderViewController**'s view hierarchy



In addition, the view controller will have a **datePicker** property that points to the **UIDatePicker** object. Finally, the view controller will be the target of the **UIButton** and must implement its action method **addReminder(_:)**.

Because **ReminderViewController**'s view has subviews, it will be easier to create this view controller's view hierarchy in Interface Builder.

Creating a view in Interface Builder

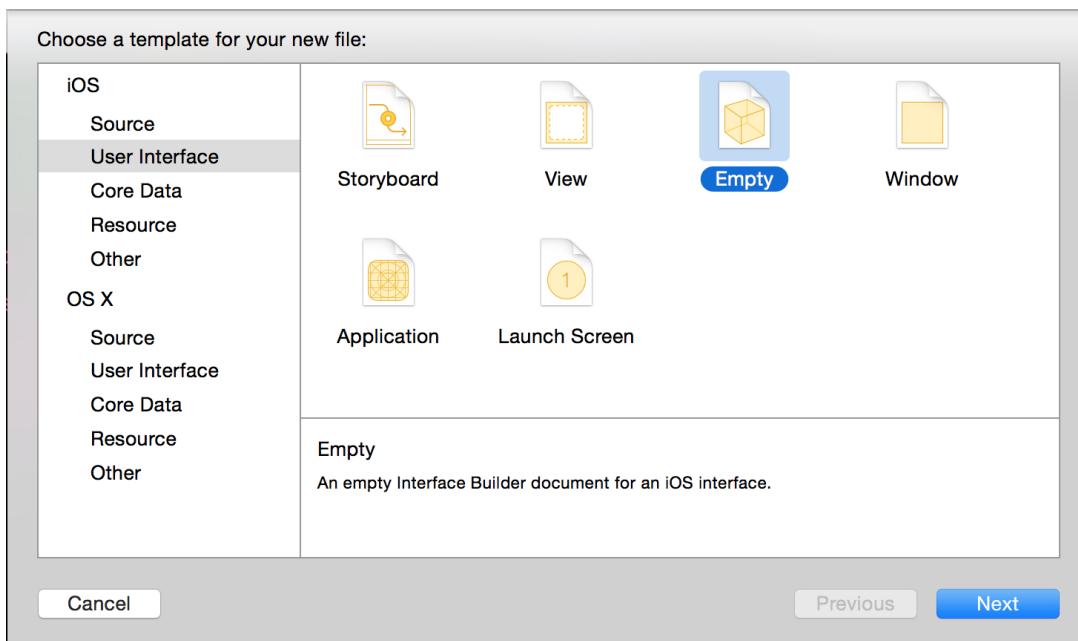
First, open `ReminderViewController.swift`. Declare a property for `datePicker`. Then add a simple implementation for `addReminder(_:)` that logs the picked date.

```
class ReminderViewController: UIViewController {
    @IBOutlet weak var datePicker: UIDatePicker!
    @IBAction func addReminder(sender: AnyObject) {
        let date = datePicker.date
        println("Setting a reminder for \(date)")
    }
}
```

Recall from Chapter 1 that the `@IBOutlet` and `@IBAction` keywords tell Xcode that you will be making these connections in Interface Builder. The first step is creating a XIB file.

Create a new XIB file by selecting `File → New → File...`. From the iOS section, select `User Interface`, choose the `Empty` template, and click `Next` (Figure 4.6).

Figure 4.6 Creating an empty XIB



Name this file `ReminderViewController` and save it. (It is important to name this and other files as we tell you. Sometimes, people will name files something different as they are working through this book. This is not a good idea. Many of the names are based on assumptions built into the iOS SDK.)

You now have a new file: `ReminderViewController.xib`. Select this file in the project navigator to open it in Interface Builder.

A XIB file contains objects: dragging an object onto the canvas saves that object into the XIB file. When the XIB file is loaded, those objects are loaded back into memory.

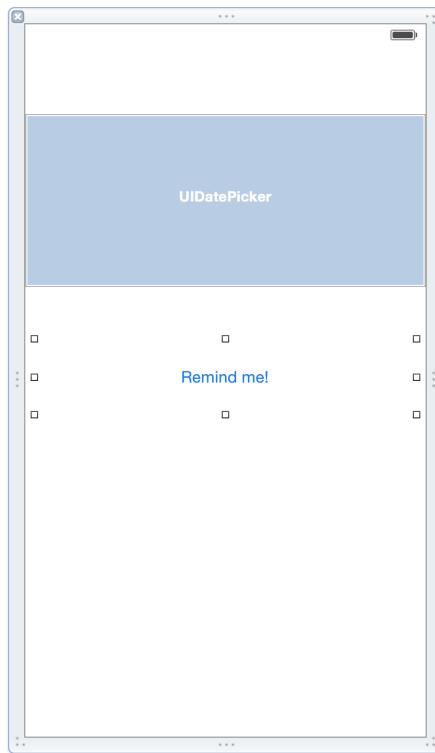
Creating view objects

In the object library (at the bottom of Xcode's righthand pane), search for `UIView`. Drag a View object onto the canvas.

With the view selected, open the Attributes inspector (Command-Option-4). From the Size drop-down, select iPhone 4.7-inch. Now the size of the canvas will match the size of the view when running on an iPhone 6.

Next, find a Date Picker and a Button in the library and drag them onto the view. Position and resize the subviews as shown in Figure 4.7. Remember that you can double-click the button to change its title.

Figure 4.7 **ReminderViewController**'s XIB file



In the document outline to the left of the canvas, you can see your view hierarchy: the View is the root and the Picker and Button are its subviews.

Figure 4.8 Hierarchy in **ReminderViewController.xib**



Loading a NIB file

When a view controller gets its view hierarchy by loading a NIB file, you do not override `loadView()`. The default implementation of `loadView()` knows how to handle loading a NIB file.

The **ReminderViewController** does need to know which NIB file to load. You can do this in **UIViewController**'s designated initializer:

```
init(nibName nibNameOrNil: String?, bundle nibBundleOrNil: NSBundle?)
```

In this method, you pass the name of the NIB file to be loaded and the bundle in which to look for that file.

In `AppDelegate.swift`, create an instance of **ReminderViewController** and tell it where to find its NIB file. Finally, make the **ReminderViewController** object the `rootViewController` of the window.

```
func application(application: UIApplication!,  
    didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {  
    // Override point for customization after application launch.  
    window = UIWindow(frame: UIScreen.mainScreen().bounds)  
  
    // This line will generate a warning, ignore it for now  
    let hvc = HypnosisViewController()  
    window!.rootViewController = hvc  
  
    // This variable represent the app bundle  
    let appBundle = NSBundle mainBundle()  
  
    // Look in the appBundle for the file ReminderViewController.xib  
    let rvc = ReminderViewController(nibName: "ReminderViewController", bundle: appBundle)  
  
    window!.rootViewController = rvc  
  
    window!.backgroundColor = UIColor whiteColor()  
    window!.makeKeyAndVisible()  
  
    return true  
}
```

The bundle that you are getting by calling the `mainBundle()` method is the application bundle. This bundle is a directory on the filesystem that contains the application's executable as well as resources (like NIB files) that the executable will use. This is where `ReminderViewController.xib` will be.

You have created and configured the objects in the view hierarchy. You have written an initializer for the view controller so that it can find and load the correct NIB file. You have set the view controller to be the root view controller to add it to the window's view hierarchy. But if you build and run now, the application will crash. Try it and see. When the application crashes, notice the exception in the console:

```
'-[UIViewController _loadViewFromNibNamed:bundle:] loaded the  
"ReminderViewController" nib but the view outlet was not set.'
```

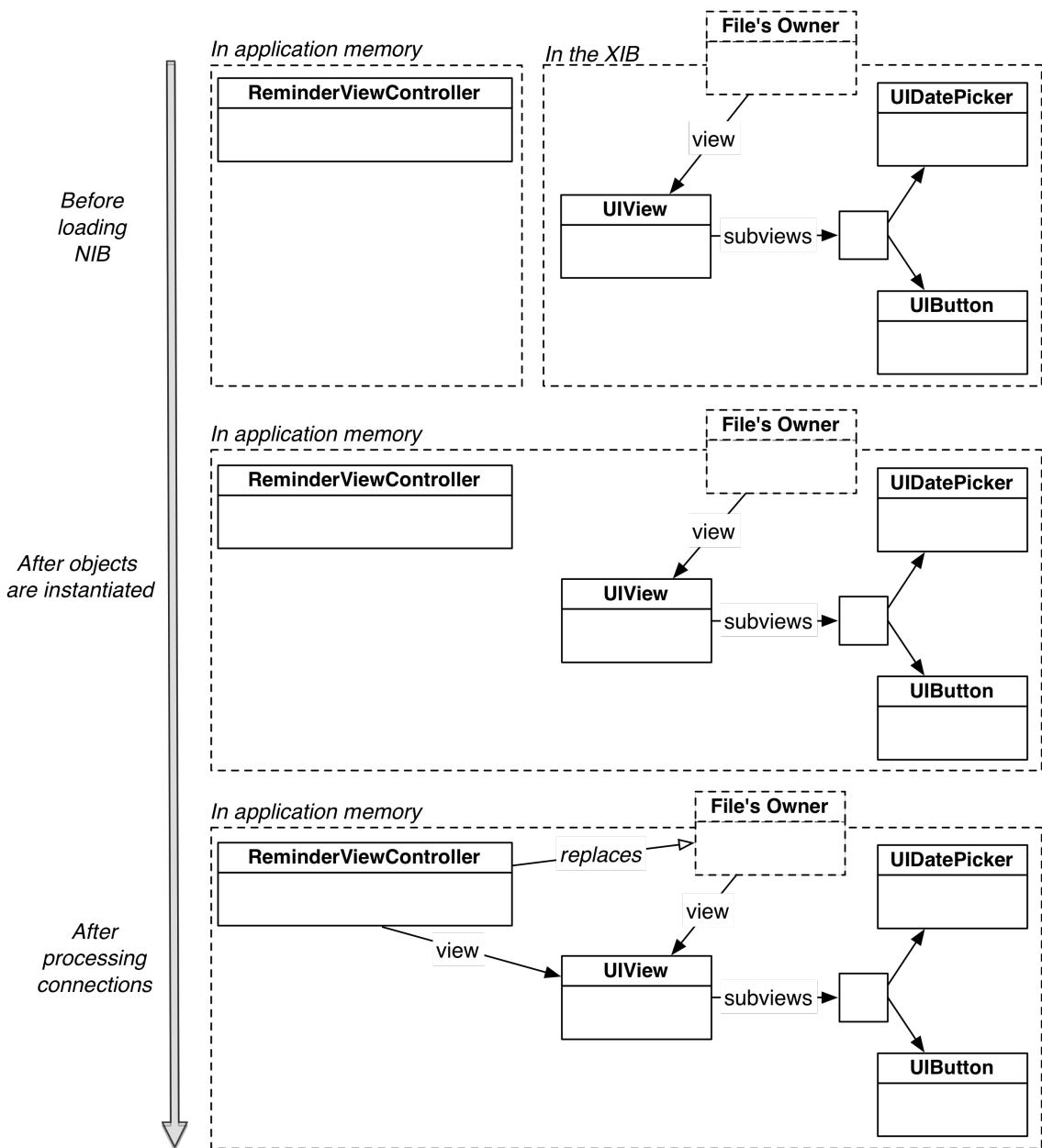
When the corresponding NIB file was loaded, these objects were instantiated. But you have not made connections to link the instantiated objects with the `ReminderViewController` in the running application. This includes the view controller's `view` property. Thus, when the view controller tries to get its `view` added to the screen, an exception is thrown because `view` is `nil`.

How can you associate a view object created in a XIB file with a view controller in a running application? This is where the File's Owner object comes in.

Connecting to File's Owner

The File's Owner object is a placeholder – it is a hole intentionally left in the XIB file. Loading a NIB, then, is a two-part process: instantiate all of the objects archived in the XIB and then drop the object that is loading the NIB into the File's Owner hole and establish the prepared connections (Figure 4.9).

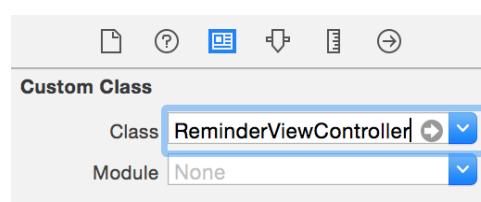
Figure 4.9 NIB loading timeline



So if you want to connect to the object that loads the NIB at runtime, you connect to the File's Owner when working in the XIB. The first step is to tell the XIB file that the File's Owner is going to be an instance of **ReminderViewController**.

Reopen **ReminderViewController.xib**. Select the File's Owner object in the document outline. Then click the **Identity Inspector** tab in the inspector area to show the *identity inspector*. Change the Class for File's Owner to **ReminderViewController** (Figure 4.10).

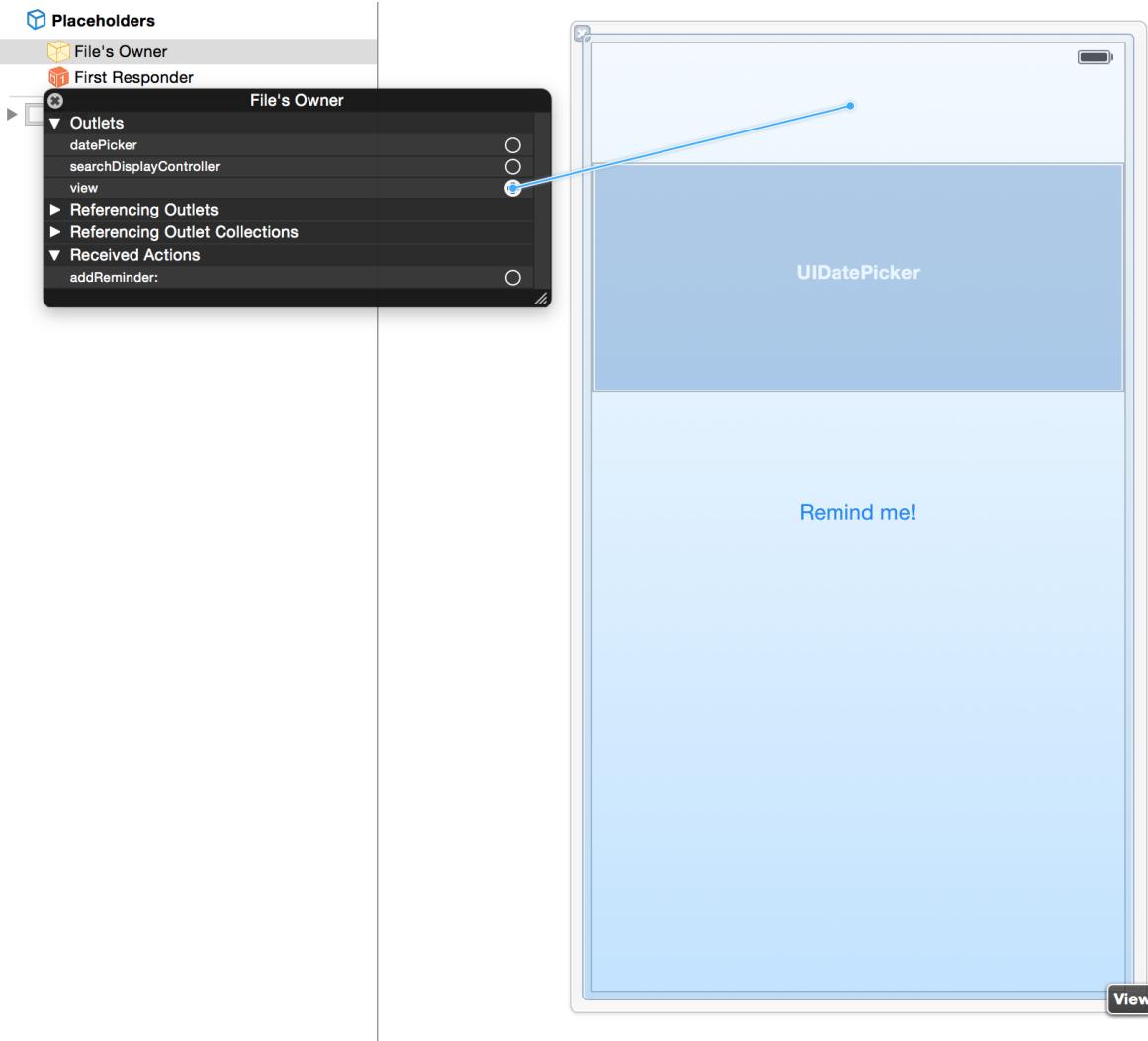
Figure 4.10 Identity inspector for File's Owner



Now you can make the missing connections. Let's start with the `view` outlet.

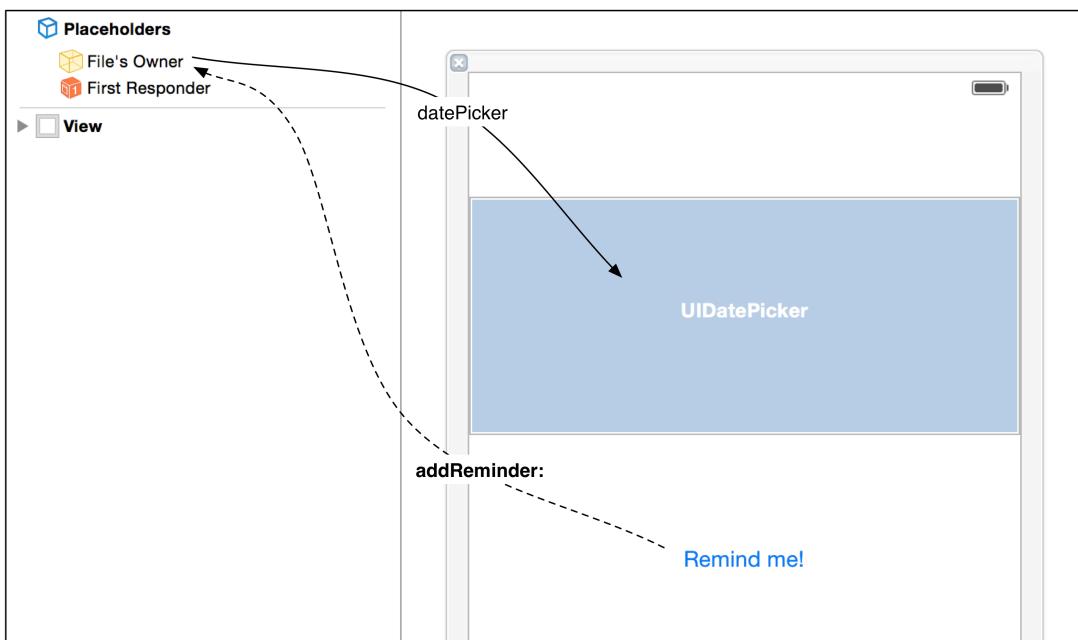
In the dock, Control-click File's Owner to bring up the panel of available connections. Drag from `view` to the `UIView` object in the canvas to set the `view` outlet to point at the `UIView` (Figure 4.11).

Figure 4.11 Set `view` outlet



Now when the `ReminderViewController` loads the NIB file, it will be able to load its `view`. Build and run the application to confirm that the `ReminderViewController`'s `view` that you created in `ReminderViewController.xib` now appears and that the application no longer crashes right away.

Finish by making the remaining connections (Figure 4.12). Control-click to reveal the File's Owner's outlets and drag to connect the `datePicker` outlet to the `UIDatePicker`. Then Control-drag from `UIButton` in the canvas to the File's Owner and select `addReminder(_:)` to set the action.

Figure 4.12 **ReminderViewController** XIB connections

Build and run the application. Select a time, tap the Remind Me button, and check the console for the log message. Later in the chapter, you will update `addReminder(_:)` to register a local notification.

Earlier in `ReminderViewController.swift`, you declared the `datePicker` outlet as weak. An `@IBOutlet` is declared as weak so that the outlet view is tied to the lifecycle of the view controller's view. If the view controller's view has been loaded into memory, the outlet will be owned by its superview. Hypothetically, if the view controller's view ever gets destroyed, you want its subviews to be released as well. If the outlet was not weak, the outlet view's strong reference count would be (at least) 2: one owner from its superview and another from the strong reference. In this case, if the view controller's view goes away, the outlet view still has a owner, and so it sticks around.

UITabBarController

View controllers become more interesting when the user's actions can cause another view controller to be presented. In this book, you will learn a number of ways to present view controllers. You will start with a **UITabBarController** that will allow the user to swap between instances of **HypnosisViewController** and **ReminderViewController**.

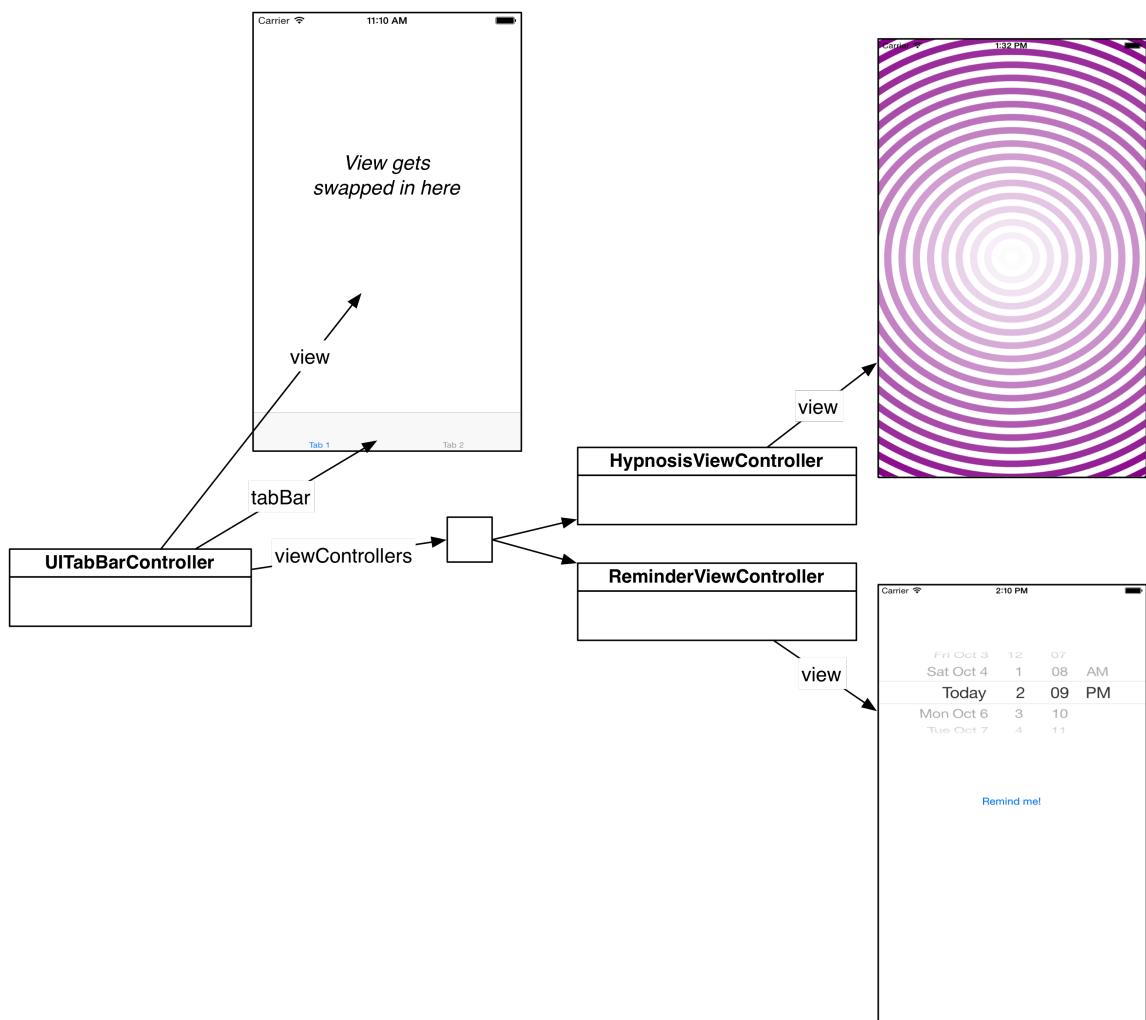
UITabBarController keeps an array of view controllers. It also maintains a tab bar at the bottom of the screen with a tab for each view controller in this array. Tapping on a tab results in the presentation of the view of the view controller associated with that tab.

In `AppDelegate.swift`, create an instance of **UITabBarController**, give it both view controllers, and install it as the `rootViewController` of the window.

```
func application(application: UIApplication!,  
    didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {  
    // Override point for customization after application launch.  
    window = UIWindow(frame: UIScreen.mainScreen().bounds)  
  
    let hvc = HypnosisViewController()  
  
    // This variable represent the app bundle  
    let appBundle = NSBundle mainBundle()  
  
    // Look in the appBundle for the file ReminderViewController.xib  
    let rvc = ReminderViewController(nibName: "ReminderViewController", bundle: appBundle)  
  
    let tbc = UITabBarController()  
    tbc.viewControllers = [hvc, rvc]  
  
window!.rootViewController = rvc  
    window!.rootViewController = tbc  
  
    window!.backgroundColor = UIColor whiteColor()  
    window!.makeKeyAndVisible()  
  
    return true  
}
```

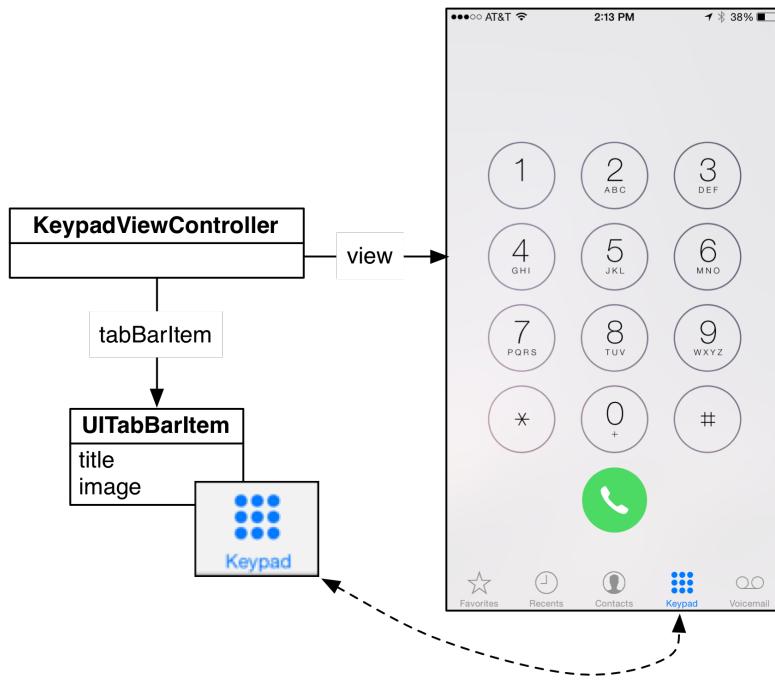
Build and run the application. The bar at the bottom is actually two tabs. Tap on the left and right sides of the tab bar to switch between the two view controllers. In the next section, you will create tab bar items to make the two tabs obvious.

UITabBarController is itself a subclass of **UIViewController**. A **UITabBarController**'s view is a **UIView** with two subviews: the tab bar and the view of the selected view controller (Figure 4.13).

Figure 4.13 **UITabBarController** diagram

Tab bar items

Each tab on the tab bar can display a title and an image. Each view controller maintains a `tabBarItem` property for this purpose. When a view controller is contained by a **UITabBarController**, its tab bar item appears in the tab bar. Figure 4.14 shows an example of this relationship in the iPhone's Phone application.

Figure 4.14 **UITabBarItem** example

First, you need to add a few files to your project that will be the images for the tab bar items. Open the Asset Catalog by opening `Images.xcassets` in the project navigator. Then, find `Hypno.png`, `Time.png`, `Hypno@2x.png`, and `Time@2x.png` in the Resources directory of the file that you downloaded earlier (<http://www.bignerdranch.com/solutions/iOSProgramming4ed.zip>). Drag these files into the images set list on the left side of the Asset Catalog.

In `HypnosisViewController.swift`, override `UIViewController`'s designated initializer, `init(nibName:bundle:)`, to get and set a tab bar item for `HypnosisViewController`. Since you are implementing your own initializer, the view controller will no longer inherit `init()` so additionally implement `init()` to call the designated initializer.

```

override init(nibName nibNameOrNil: String?, bundle nibBundleOrNil: NSBundle?) {
    super.init(nibName: nibNameOrNil, bundle: nibBundleOrNil)

    // Set the tab bar item's title
    tabBarItem.title = "Hypnotize"

    // Put an image on the tab bar item
    tabBarItem.image = UIImage(named: "Hypno.png")
}

convenience override init() {
    self.init(nibName: nil, bundle: nil)
}

required init(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
}
    
```

In `ReminderViewController.swift`, do the same thing.

```

override init(nibName nibNameOrNil: String!, bundle nibBundleOrNil: NSBundle!) {
    super.init(nibName: nibNameOrNil, bundle: nibBundleOrNil)

    // Set the tab bar item's title
    tabBarItem.title = "Reminder"

    // Put an image on the tab bar item
    tabBarItem.image = UIImage(named: "Time.png")
}

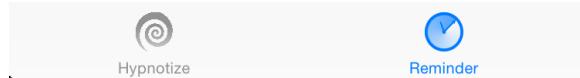
convenience override init() {
    self.init(nibName: "ReminderViewController", bundle: nil)
}

required init(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)
}

```

Build and run the application, and you will see helpful images and titles in the tab bar. (Figure 4.15).

Figure 4.15 Tab bar items with labels and icons



UIViewController Initializers

When you created a tab bar item for `HypnosisViewController`, you overrode `init(nibName:bundle:)`. However, when you initialized the `HypnosisViewController` instance in `AppDelegate.swift`, you called the default initializer (`init()`) and still got the tab bar items. This is because `init(nibName:bundle:)` is the designated initializer of `UIViewController`. Calling `init` on view controller calls `init(nibName:bundle:)` and passes `nil` for both arguments. You implemented `init()` to provide this same default behavior.

`HypnosisViewController` does not use a NIB file to create its view, so the filename parameter is irrelevant. What happens if you send `init` to a view controller that does use a NIB file? Let's find out.

In `AppDelegate.swift`, change your code to initialize the `ReminderViewController` with `init()` rather than `init(nibName:bundle:)`.

```

let hvc = HypnosisViewController()

// This variable represent the app bundle
let AppBundle = NSBundle.mainBundle()

// Look in the AppBundle for the file ReminderViewController.xib
let rvc = ReminderViewController(nibName: "ReminderViewController", bundle: AppBundle)
let rvc = ReminderViewController()

let tbc = UITabBarController()

```

Build and run the application, and it will work just as before.

Adding a Local Notification

Now you are going to implement the reminder feature using a *local notification*. A local notification is a way for an application to alert the user even when the application is not currently running.

(An application can also use push notifications that are implemented using a backend server. For more about push notifications, read Apple's *Local and Push Notification Programming Guide*.)

Before you can present a local notification, you need to tell the application what kind of notification you'll be presenting to the user. This can be an alert presented to the user, a badge on the app icon, or a sound being played. You will present an alert to the user.

In `AppDelegate.swift`, register the `Alert` notification type with the application.

```
window = UIWindow(frame: UIScreen.mainScreen().bounds)

let settings = UIUserNotificationSettings(forTypes: .Alert,
    categories: nil)
application.registerUserNotificationSettings(settings)

let hvc = HypnosisViewController()
```

Getting a local notification to display is easy. You create a `UILocalNotification` and give it some text and a date. Then you schedule the notification with the shared application – the single instance of `UIApplication`.

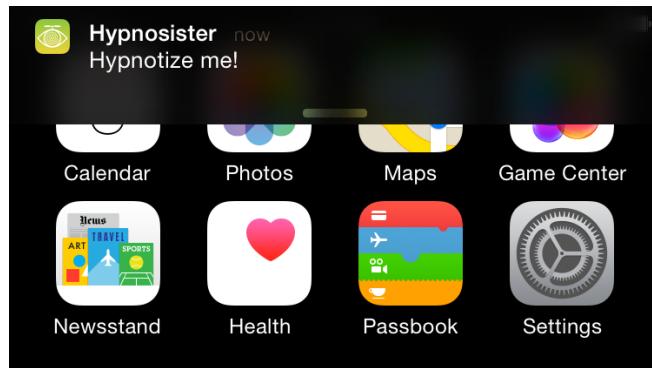
Update the `addReminder:` method to do this:

```
@IBAction func addReminder(sender: AnyObject) {
    let date = datePicker.date
    println("Setting a reminder for \(date)")

    let note = UILocalNotification()
    note.alertBody = "Hypnotize me!"
    note.fireDate = date
    UIApplication.sharedApplication().scheduleLocalNotification(note)
}
```

Build and run the application. The application will ask for your permission to display notification. Accept the request. Then use the date picker to select a time in the very near future and tap the Remind Me button. To see the notification, Hypnosister needs to be in the background. Press the Home button at the bottom of the device or select **Hardware → Home** in the simulator. When the time that you picked is reached, a notification banner will appear at the top of the screen (Figure 4.16). Tapping on the notification will launch the Hypnosister application.

Figure 4.16 Local notification



There is an issue: the user can select a time in the past. It would be nice if the date picker did not allow this. You will take care of this shortly.

Loaded and Appearing Views

Now that you have two view controllers, the lazy loading of views that you learned about earlier becomes more important.

When the application launches, the tab bar controller defaults to loading the view of the first view controller in its array, the `HypnosisViewController`. This means that the `ReminderViewController`'s view is not needed and will only be needed when (or if) the user taps the tab to see it.

You can test this behavior for yourself – when a view controller finishes loading its view, it is sent the message `viewDidLoad()`.

In `HypnosisViewController.swift`, override `viewDidLoad()` to log a statement to the console.

```
override func viewDidLoad() {
    // Always call the super implementation of viewDidLoad
    super.viewDidLoad()

    println("HypnosisViewController loaded its view.")
}
```

In `ReminderViewController.swift`, override the same method.

```
override func viewDidLoad() {
    // Always call the super implementation of viewDidLoad
    super.viewDidLoad()

    println("ReminderViewController loaded its view.")
}
```

Build and run the application. The console reports that `HypnosisViewController` loaded its view right away. Tap `ReminderViewController`'s tab, and the console will report that its view is now loaded. At this point, both views have been loaded, so switching between the tabs now will no longer trigger the `viewDidLoad()` method. (Try it and see.)

To preserve the benefits of lazy loading, you should never access the `view` property of a view controller in `init(nibName:bundle:)`. Asking for the `view` in the initializer will cause the view controller to load its `view` prematurely.

Accessing subviews

Often, you will want to do some extra initialization of the subviews that are defined in the XIB file before they appear to the user. However, you cannot do this in the view controller's initializer because the NIB file has not yet been loaded. If you try, any references that the view controller declares that will eventually point to subviews will be pointing to `nil`. If you try to call a method on that view, iOS will crash (since our outlets are usually implicitly unwrapped optionals.).

So where can you access a subview? There are two main options, depending on what you need to do.

The first option is the `viewDidLoad()` method that you overrode to spot lazy loading. The view controller receives this message after the view controller's NIB file is loaded, at which point all of the view controller's pointers will be pointing to the appropriate objects. The second option is another `UIViewController` method `viewWillAppear(_:)`. The view controller receives this message just before its `view` is added to the window.

What is the difference? You override `viewDidLoad()` if the configuration only needs to be done once during the run of the app. You override `viewWillAppear(_:)` if you need the configuration to be done and redone every time the view controller appears on screen.

There is a subview of the `ReminderViewController`'s view that needs some extra work – the date picker. Currently, users can pick reminder times in the past. You are going to configure the date picker to only allow users to select a time that is at least 60 seconds in the future.

This is something that will need to be done every time the view appears, not just once after the view is loaded, so you are going to override `viewWillAppear(_:)`.

In `ReminderViewController.swift`, override `viewWillAppear(_:)` to set the `minimumDate` of the date picker.

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)

    datePicker.minimumDate = NSDate(timeIntervalSinceNow: 60)
}
```

Build and run the application. Select the Reminder tab and confirm that the date picker will only allow the user to select a date in the future.

If you had overridden `viewDidLoad()` instead, then `datePicker`'s `minimumDate` would be set to 60 seconds after the view was initially loaded and would remain unchanged for the entire run of the application. If the app ran for very long, then users would soon be able to select times in the past.

Wondering about the animated flag on this method? It indicates whether the appearance or disappearance transition is animated or not. In the case of **UITabBarController**, the transition is not animated. Later in the book, in Chapter 7, you will use **UINavigationController**, which animates view controllers being pushed on and off screen.

Interacting with View Controllers and Their Views

Let's look at some methods that are called during the lifecycle of a view controller and its view. Some of these methods you have already seen, and some are new:

- **application(_: didFinishLaunchingWithOptions:)** is where you instantiate and set an application's root view controller.

This method gets called exactly once when the application has launched. Even if you go to another app and come back, this method does not get called again. If you reboot your phone and start the app again, **application(_: didFinishLaunchingWithOptions:)** will get called again.

- **init(nibName:bundle:)** is the designated initializer for **UIViewController**.

When a view controller instance is created, its **init(nibName:bundle:)** gets called once. Note that in some apps, you may end up creating several instances of the same view controller class. This method will get called once on each as it is created.

- **loadView()** is overridden to create a view controller's view programmatically.

- **viewDidLoad()** can be overridden to configure views created by loading a NIB file. This method gets called after the view of a view controller is created.

- **viewWillAppear(_:)** can be overridden to configure views created by loading a NIB file.

This method and **viewDidAppear(_:)** will get called every time your view controller is moved on screen. **viewWillDisappear(_:)** and **viewDidDisappear(_:)** will get called every time your view controller is moved offscreen. So if you launch the app you are working on and hop back and forth between Hypnosis and Reminder, **ReminderViewController**'s **viewDidLoad()** method will be called once, but **viewWillAppear(_:)** will be called each time you view the Reminder tab.

Bronze Challenge: Another Tab

Give the **UITabBarController** a third tab that presents a quiz to the user. (Hint: you can reuse files from your Quiz project for this challenge.)

Silver Challenge: Controller Logic

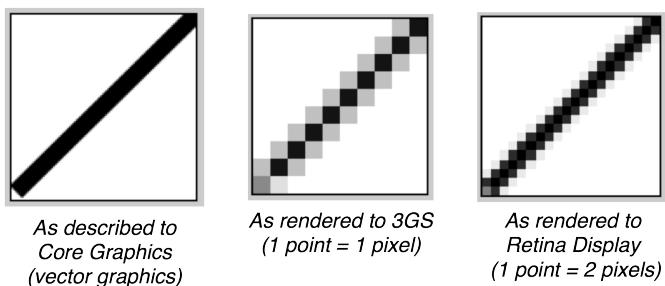
Add a **UISegmentedControl** to **HypnosisViewController**'s view with segments for Purple, Green, and Blue. When the user taps the segmented control, change the color of the circles in **HypnosisView**. Be sure to create a copy of the project and work from that copy while attempting this challenge.

For the More Curious: Retina Display

With the release of iPhone 4, Apple introduced the Retina display for the iPhone and iPod touch. The Retina display has much higher resolution – 640x1136 pixels (on a 4-inch display) and 640x960 pixels (on a 3.5-inch display) compared to 320x480 pixels on earlier devices. Let's look at what you should do to make graphics look their best on both displays.

For vector graphics, like **HypnosisView**'s **drawRect(_:)** method, you do not need to do anything; the same code will render as crisply as the device allows. However, if you draw using Core Graphics functions, these graphics will appear differently on different devices. In Core Graphics, also called Quartz, lines, curves, text, etc. are described in terms of *points*. On a non-Retina display, a point is 1x1 pixel. On a Retina display, a point is 2x2 pixels (Figure 4.17).

Figure 4.17 Rendering to different resolutions



Given these differences, bitmap images (like JPEG or PNG files) will be unattractive if the image is not tailored to the device's screen type. Say your application includes a small image of 25x25 pixels. If this image is displayed on a Retina display, then the image must be stretched to cover an area of 50x50 pixels. At this point, the system does a type of averaging called anti-aliasing to keep the image from looking jagged. The result is an image that is not jagged – but it is fuzzy (Figure 4.18).

Figure 4.18 Fuzziness from stretching an image



You could use a larger file instead, but the averaging would then cause problems in the other direction when the image is shrunk for a non-Retina display. The only solution is to bundle two image files with your application: one at a pixel resolution equal to the number of points on the screen for non-Retina displays and one twice that size in pixels for Retina displays.

Fortunately, you do not have to write any extra code to handle which image gets loaded on which device. All you have to do is suffix the higher-resolution image with @2x, but make sure the @2x goes before the file extension. You want to use `Fido@2x.png`, not `Fido.png@2x`. Then, when you use `UIImage`'s `initWithNamed:` initializer to load the image, this method looks in the bundle and gets the file that is appropriate for the particular device.

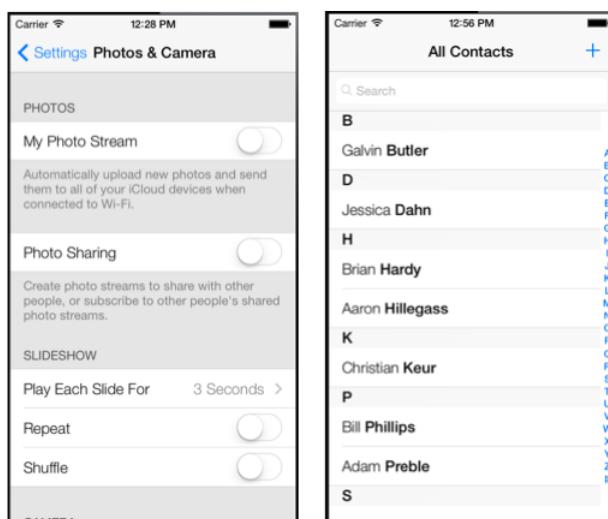
5

UITableView and UITableViewcontroller

Many iOS applications show the user a list of items and allow the user to select, delete, or reorder items on the list. Whether an application displays a list of people in the user's address book or a list of bestselling items on the App Store, it is a **UITableView** doing the work.

A **UITableView** displays a single column of data with a variable number of rows. Figure 5.1 shows some examples of **UITableView**.

Figure 5.1 Examples of **UITableView**

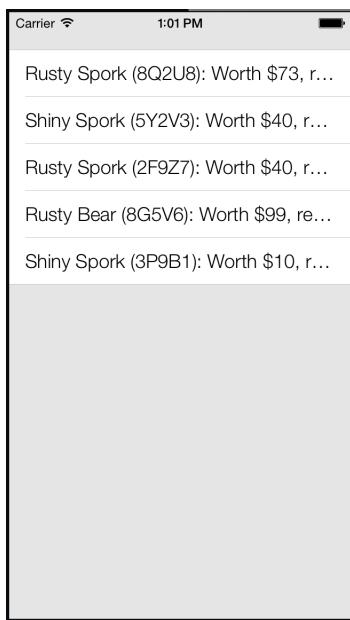


Beginning the Homepwner Application

In this chapter, you are going to start an application called Homepwner that keeps an inventory of all your possessions. In the case of a fire or other catastrophe, you will have a record for your insurance company. ("Homepwner," by the way, is not a typo. If you need a definition for the word "pwn," please visit www.urbandictionary.com.)

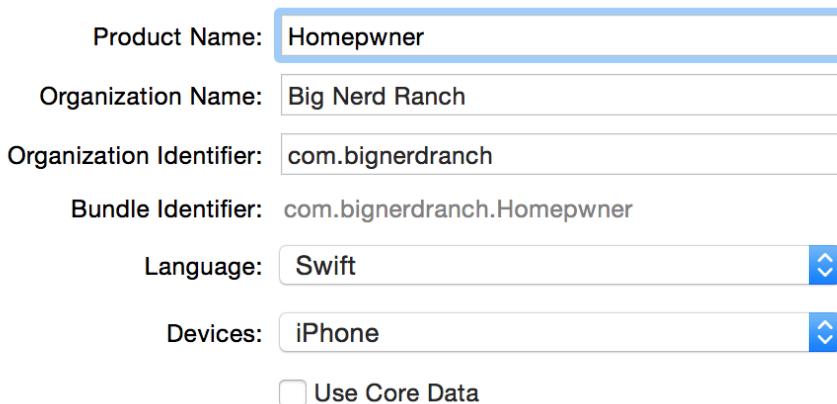
So far, your iOS projects have been small, but Homepwner will grow into a realistically complex application over the course of nine chapters. By the end of this chapter, Homepwner will present a list of **Item** objects in a **UITableView**, as shown in Figure 5.2.

Figure 5.2 Homepwner: phase 1



Create a new iOS Single View Application project and configure it as shown in Figure 5.3.

Figure 5.3 Configuring Homepwner



Just as you did in

[<aref></aref>](#)

, you will strip down the template so you can learn how everything works. This is crucial to your understanding of the frameworks and how iOS works.

From the Project navigator, delete the two files `Main.storyboard` and `ViewController.swift`. Then, select `Homepwner` from the top of the Project navigator. Under General, remove the text that says `Main` next to `Main Interface`.

Finally, open `AppDelegate.swift` and add code to `application(_:didFinishLaunchingWithOptions:)` to set up the window.

```
func application(application: UIApplication,
               didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
    // Override point for customization after application launch.
    window = UIWindow(frame: UIScreen.mainScreen().bounds)

    window!.backgroundColor = UIColor.whiteColor()
    window!.makeKeyAndVisible()
    return true
}
```

UITableViewController

A **UITableView** is a view object. Recall in the Model-View-Controller design pattern, which iOS developers do their best to follow, each class is exactly one of the following:

- *Model*: Holds data and knows nothing about the user interface.
- *View*: Is visible to the user and knows nothing about the model objects.
- *Controller*: Keeps the user interface and the model objects in sync. Controls the flow of the application; for example, the controller might be responsible for showing a “Really delete this item?” message before actually deleting some data.

Thus, a **UITableView**, a view object, does not handle application logic or data. When using a **UITableView**, you must consider what else is necessary to get the table working in your application:

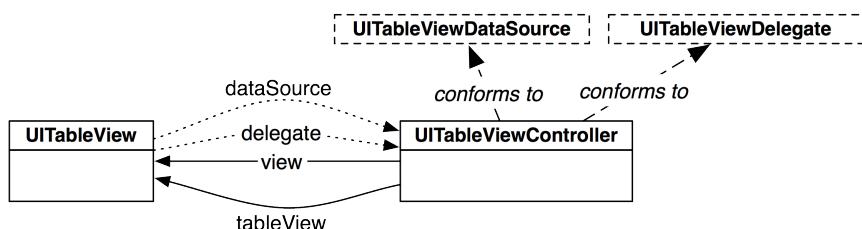
- A **UITableView** typically needs a view controller to handle its appearance on the screen.
- A **UITableView** needs a *data source*. A **UITableView** asks its data source for the number of rows to display, the data to be shown in those rows, and other tidbits that make a **UITableView** a useful user interface. Without a data source, a table view is just an empty container. The *dataSource* for a **UITableView** can be any type of object as long as it conforms to the **UITableViewDataSource** protocol.
- A **UITableView** typically needs a *delegate* that can inform other objects of events involving the **UITableView**. The delegate can be any object as long as it conforms to the **UITableViewDelegate** protocol.

An instance of the class **UITableViewController** can fill all three roles: view controller, data source, and delegate.

UITableViewController is a subclass of **UIViewController**, so a **UITableViewController** has a *view*. A **UITableViewController**'s *view* is always an instance of **UITableView**, and the **UITableViewController** handles the preparation and presentation of the **UITableView**.

When a **UITableViewController** creates its *view*, the *dataSource* and *delegate* properties of the **UITableView** are automatically set to point at the **UITableViewController** (Figure 5.4).

Figure 5.4 **UITableViewController-UITableView** relationship



Subclassing UITableViewController

Now you are going to write a subclass of **UITableViewController** for Homeowner. For this view controller, you will use the **Cocoa Touch Class** template. From the File menu, select New and then File.... From the iOS section, select Source, choose Cocoa Touch Class, and click Next. Then, select **NSObject** from the Subclass of pop-up menu and enter **ItemsViewController** as the name of the new class. Click Next and then click Create on the next sheet to save your class.

Open **ItemsViewController.swift** and change its superclass:

```

import UIKit

class ItemsViewController: NSObject {
    class ItemsViewController: UITableViewController {
}
  
```

The designated initializer of **UITableViewController** is **init(style:)**, which takes a constant that determines the style of the table view. There are two options: **UITableViewStylePlain** and **UITableViewStyleGrouped**.

Open **AppDelegate.swift**. In **application(_:didFinishLaunchingWithOptions:)**, create an instance of **ItemsViewController** and set it as the **rootViewController** of the window.

```
func application(application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
    // Override point for customization after application launch.
    window = UIWindow(frame: UIScreen.mainScreen().bounds)

    // Create a ItemsViewController
    let ivc = ItemsViewController(style: .Plain)

    // Place ItemsViewController's table view in the window hierarchy
    window!.rootViewController = ivc

    window!.backgroundColor = UIColor.whiteColor()
    window!.makeKeyAndVisible()

    return true
}
```

Build and run your application. You should see an empty table view, as shown in Figure 5.5. As a subclass of **UIViewController**, a **UITableViewController** inherits the **view** method. This method calls **loadView()**, which creates and loads an empty view object if none exists. A **UITableViewController**'s view is always an instance of **UITableView**, so sending **view** to the **UITableViewController** gets you a bright, shiny, and empty table view.

Figure 5.5 Empty **UITableView**



Your table view needs some rows to display. Remember the **Item** class you wrote in ????. Now you are going to use that class again: each row of the table view will display an instance of **Item**.

Locate the file for **Item** (**Item.swift** from the **RandomItems** project) in Finder and drag them onto Homeowner's project navigator.

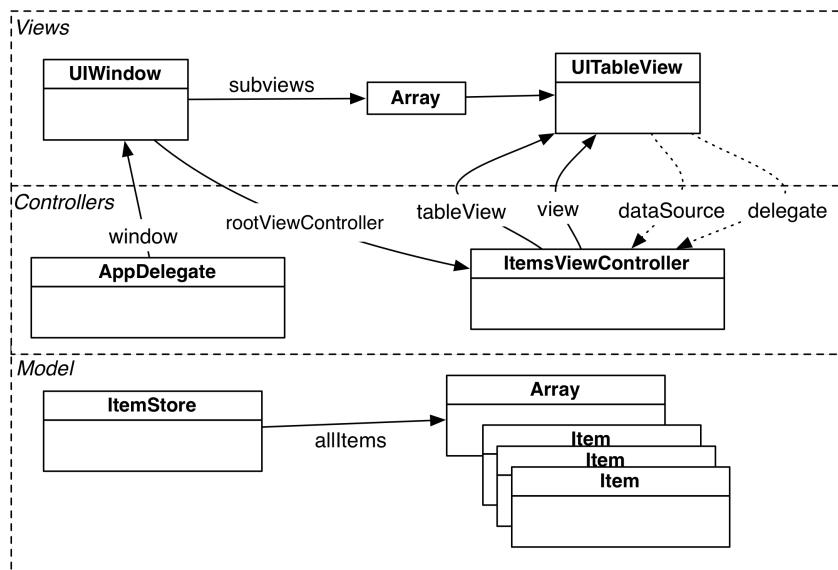
When dragging these files onto your project window, select the checkbox labeled **Copy items into destination group's folder** when prompted. This will copy the files from their current directory to your project's directory on the filesystem and add them to your project.

UITableView's Data Source

The process of providing a **UITableView** with rows in Cocoa Touch is different from the typical procedural programming task. In a procedural design, you tell the table view what it should display. In Cocoa Touch, the table view asks another object – its **dataSource** – what it should display. In this case, the **ItemsViewController** is the data source, so it needs a way to store item data.

In ???, you used an **Array** to store **Item** instances. You will do the same thing in this chapter, but with a little twist. The **Array** that holds the **Item** instances will be abstracted into another object – an **ItemStore** (Figure 5.6). Why not just use an array? Eventually, the **ItemStore** object will also take care of the saving and loading of the items.

Figure 5.6 Homepwner object diagram



If an object wants to see all of the items, it will ask the **ItemStore** for the array that contains them. In future chapters, you will make the store responsible for performing operations on the array, like reordering, adding, and removing items. It will also be responsible for saving and loading the items from disk.

Creating ItemStore

From the File menu, select New and then File.... Create a new Cocoa Touch Class that is a subclass of **NSObject** and name it **ItemStore**.

The **ItemsViewController** controller will call a method on **ItemStore** when it wants a new **Item** to be created. The **ItemStore** will oblige, create the object, and add it to an array of instances of **Item**. The **ItemsViewController** will also ask the **ItemStore** for all of the items in the store when it wants to populate its **UITableView**.

In **ItemStore.swift**, declare a property to store the list of **Items**.

```
class ItemStore: NSObject {
    var allItems: [Item] = []
}
```

Now implement **createItem()** in **ItemStore.swift** to create and return a new **Item**.

```
func createItem() -> Item {
    let newItem = Item(random: true)
    allItems.append(newItem)
    return newItem
}
```

Giving the controller access to the store

In `ItemsViewController.swift`, add a property for an `ItemStore` and create a new initializer that takes in an `ItemStore` instance as its only argument. You'll also need to implement the required `init(coder:)`, which we will discuss later.

```
class ItemsViewController: UITableViewController {

    let itemStore: ItemStore

    init(itemStore: ItemStore) {
        self.itemStore = itemStore
        super.init(nibName: nil, bundle: nil)
    }

    required init(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
}
```

Then, open `AppDelegate.swift` and instantiate the `ItemsViewController` using the new initializer.

```
func application(application: UIApplication!,
    didFinishLaunchingWithOptions launchOptions: NSDictionary!) -> Bool {
    // Override point for customization after application launch.
    window = UIWindow(frame: UIScreen.mainScreen().bounds)

    // Create a ItemStore
    let itemStore = ItemStore()

    // Create a ItemsViewController
    let ivc = ItemsViewController(style: .Plain)

    // Create a ItemsViewController and give it an ItemStore
    let ivc = ItemsViewController(itemStore: itemStore)

    // Place ItemsViewController's table view in the window hierarchy
    window!.rootViewController = ivc

    window!.backgroundColor = UIColor.whiteColor()
    window!.makeKeyAndVisible()

    return true
}
```

Implementing data source methods

In `ItemsViewController.swift`, update the designated initializer to add five random items to the `ItemStore`.

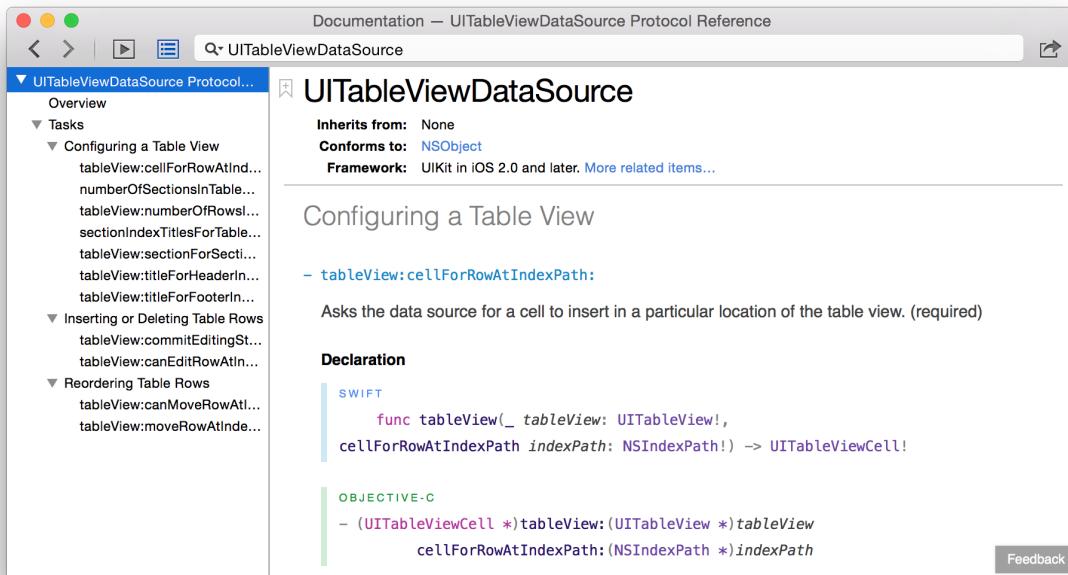
```
init(itemStore: ItemStore) {
    self.itemStore = itemStore
    super.init(nibName: nil, bundle: nil)

    for _ in 0..<5 {
        self.itemStore.createItem()
    }
}
```

Now that there are some items in the store, you need to teach `ItemsViewController` how to turn those items into rows that its `UITableView` can display. When a `UITableView` wants to know what to display, it sends messages from the set of messages declared in the `UITableViewDataSource` protocol.

From the Help menu, choose Documentation and API Reference. Search for the `UITableViewDataSource` protocol reference and then select Tasks from the lefthand pane (Figure 5.7).

Figure 5.7 UITableViewDataSource protocol documentation



In the Configuring a Table View task, notice that the first two methods are notated as (*required*). For **ItemsViewController** to conform to **UITableViewDataSource**, it must implement **tableView(_:_:numberOfRowsInSection:)** and **tableView(_:_:cellForRowAtIndexPath:)**. These methods tell the table view how many rows it should display and what content to display in each row.

Whenever a **UITableView** needs to display itself, it sends a series of messages (the required methods plus any optional ones that have been implemented) to its **dataSource**. The required method **tableView(_:_:numberOfRowsInSection:)** returns an integer value for the number of rows that the **UITableView** should display. In the table view for Homeowner, there should be a row for each entry in the store.

In **ItemsViewController.swift**, implement **tableView(_:_:numberOfRowsInSection:)**.

```
override func tableView(tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    return itemStore.allItems.count
}
```

Wondering about the section that this method refers to? Table views can be broken up into sections, and each section has its own set of rows. For example, in the address book, all names beginning with “D” are grouped together in a section. By default, a table view has one section, and in this chapter, you will work with only one. Once you understand how a table view works, it is not hard to use multiple sections. In fact, using sections is the first challenge at the end of this chapter.

The second required method in the **UITableViewDataSource** protocol is **tableView(_:_:cellForRowAtIndexPath:)**. To implement this method, you need to learn about another class – **UITableViewCell**.

UITableViewCells

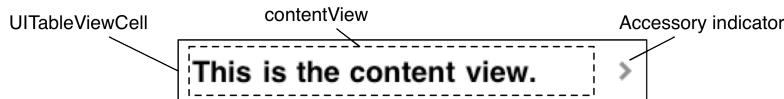
Each row of a table view is a view. These views are instances of **UITableViewCell**. In this section, you will be creating the instances of **UITableViewCell** to fill the table view.

A cell itself has one subview – its **contentView** (Figure 5.8). The **contentView** is the superview for the content of the cell. The cell may also draw an accessory indicator. The accessory indicator shows an action-oriented

icon, such as a checkmark, a disclosure icon, or an information button. These icons are accessed through pre-defined constants for the appearance of the accessory indicator. The default is `UITableViewCellAccessoryNone`, and that is what you are going to use in this chapter. But you will see the accessory indicator again in ???.

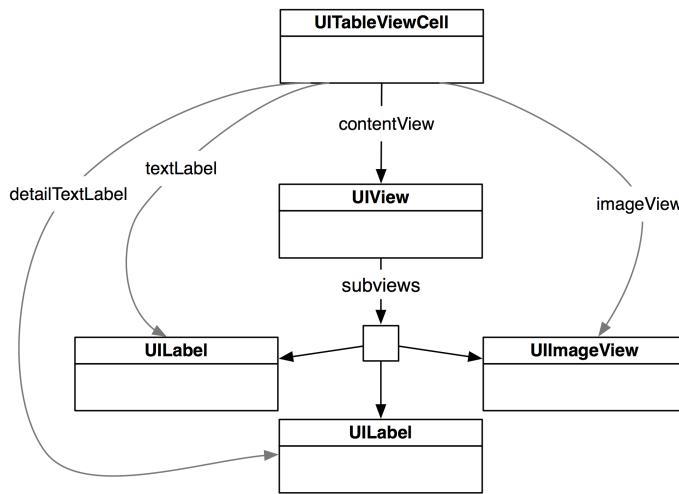
(Curious now? See the reference page for `UITableViewCell` for more details.)

Figure 5.8 `UITableViewCell` layout



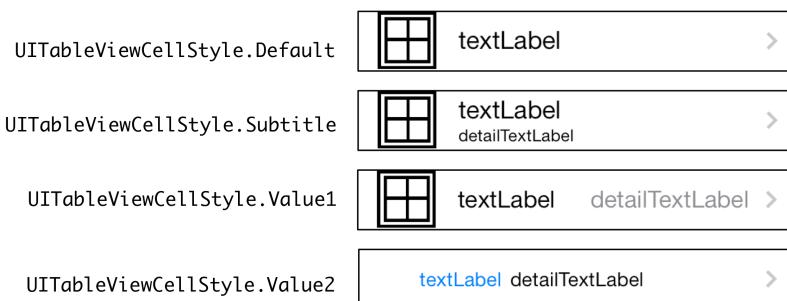
The real meat of a `UITableViewCell` is the three subviews of the `contentView`. Two of those subviews are `UILabel` instances that are properties of `UITableViewCell` named `textLabel` and `detailTextLabel`. The third subview is a `UIImageView` called `imageView` (Figure 5.9). In this chapter, you will only use `textLabel`.

Figure 5.9 `UITableViewCell` hierarchy



Each cell also has a `UITableViewCellStyle` that determines which subviews are used and their position within the `contentView`. Examples of these styles and their constants are shown in Figure 5.10.

Figure 5.10 `UITableViewCellStyle`

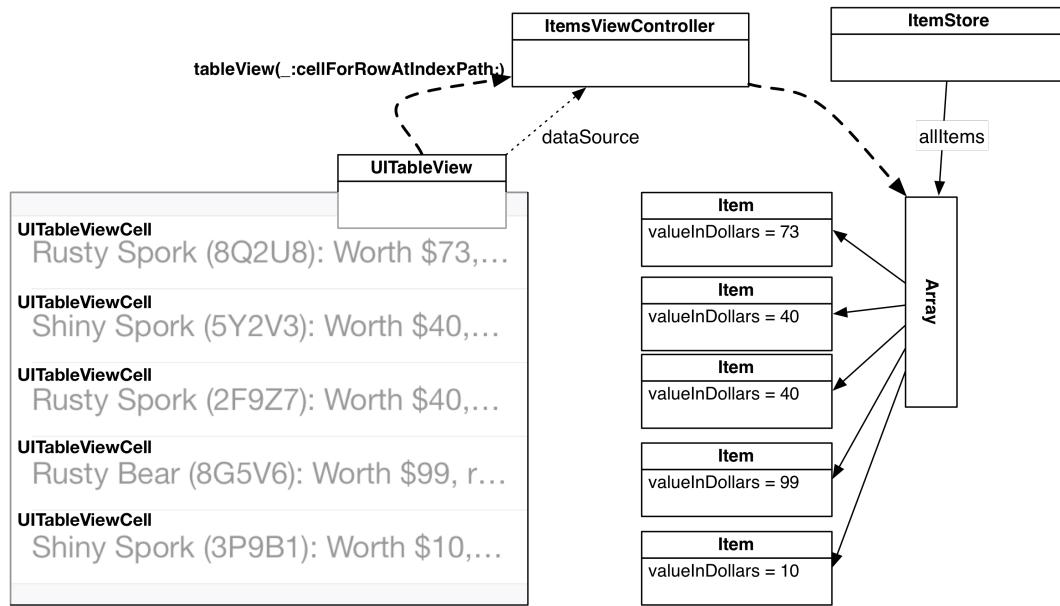


Creating and retrieving `UITableViewCell`s

For now, each cell will display the `description` of an `Item` as its `textLabel`. To make this happen, you need to implement the second required method from the `UITableViewDataSource` protocol,

`tableView(_:cellForRowIndexPath:)`. This method will create a cell, set its `textLabel` to the **description** of the corresponding **Item**, and return it to the **UITableView** (Figure 5.11).

Figure 5.11 **UITableViewCell** retrieval



How do you decide which cell a **Item** corresponds to? One of the parameters sent to `tableView(_:cellForRowIndexPath:)` is an **NSIndexPath**, which has two properties: `section` and `row`. When this message is sent to a data source, the table view is asking, “Can I have a cell to display in section X, row Y?” Because there is only one section in this exercise, your implementation will only be concerned with the row.

In `ItemsViewController.swift`, implement `tableView(_:cellForRowIndexPath:)` so that the *n*th row displays the *n*th entry in the `allItems` array.

```
override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    // Create an instance of UITableViewCell, with default appearance
    let cell = UITableViewCell(style: .Default, reuseIdentifier: "UITableViewCell")

    // Set the text on the cell with the description of the item
    // that is at the nth index of items, where n = row this cell
    // will appear in on the tableview
    let item = itemStore.allItems[indexPath.row]

    cell.textLabel?.text = item.name

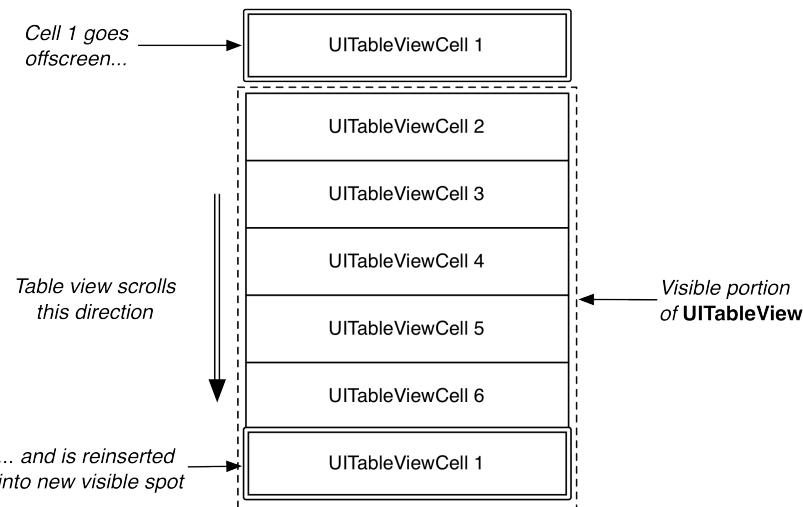
    return cell
}
```

Build and run the application now, and you will see a **UITableView** populated with a list of random items.

Reusing UITableViewCells

iOS devices have a limited amount of memory. If you were displaying a list with thousands of entries in a **UITableView**, you would have thousands of instances of **UITableViewCell**. And your long-suffering iPhone would sputter and die. In its dying breath, it would say “You only needed enough cells to fill the screen!” It would be right.

To preserve the lives of iOS devices everywhere, you can reuse table view cells. When the user scrolls the table, some cells move offscreen. Offscreen cells are put into a pool of cells available for reuse. Then, instead of creating a brand new cell for every request, the data source first checks the pool. If there is an unused cell, the data source configures it with new data and returns it to the table view.

Figure 5.12 Reusable instances of **UITableViewCell**

There is one problem: sometimes a **UITableView** has different types of cells. Occasionally, you have to subclass **UITableViewCell** to create a special look or behavior. However, different subclasses floating around the pool of reusable cells create the possibility of getting back a cell of the wrong type. You must be sure of the type of the cell returned to you so that you can be sure of what properties and methods it has.

Note that you do not care about getting any specific cell out of the pool because you are going to change the cell content anyway. What you need is a cell of a specific type. The good news is that every cell has a `reuseIdentifier` property of type **String**. When a data source asks the table view for a reusable cell, it passes a string and says, “I need a cell with this reuse identifier.” By convention, the reuse identifier is typically the name of the cell class.

To reuse cells, you need to register either a class or a NIB with the table view for a specific reuse identifier. Right now, you will register the default **UITableViewCell** class. You tell the table view “Hey, any time I ask for a cell with *this reuse identifier*, give me back a cell that is *this specific class*.” The table view will oblige and either give you a cell from the reuse pool, or instantiate a new cell if there are no cells of that type in the reuse pool.

A good place to register with the table view is in `viewDidLoad()`. In `ItemsViewController.swift`, override `viewDidLoad()` to register the **UITableViewCell** class with the table view.

```
override func viewDidLoad() {
    super.viewDidLoad()

    tableView.registerClass(UITableViewCell.self,
        forCellReuseIdentifier: "UITableViewCell")
}
```

Next, update `tableView(_:cellForRowIndexPath:)` to reuse cells:

```
override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let cell = UITableViewCell(style: .Default, reuseIdentifier: nil)

    // Get a new or recycled cell
    let cell = tableView.dequeueReusableCellWithIdentifier("UITableViewCell",
        forIndexPath: indexPath) as UITableViewCell

    let item = itemStore.allItems[indexPath.row]
    cell.textLabel?.text = item.name

    return cell
}
```

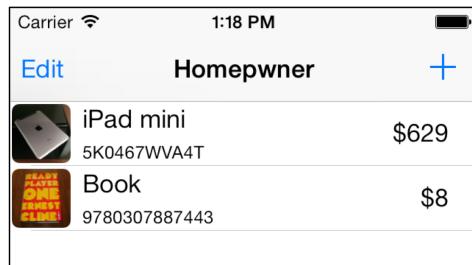
Reusing cells means that you only have to create a handful of cells, which puts fewer demands on memory. Your application’s users (and their devices) will thank you. Build and run the application. The behavior of the application should remain the same.

Subclassing UITableViewCell

For many applications, the basic cell with its `textLabel`, `detailTextLabel`, and `imageView` is sufficient. However, when you need a cell with more detail or a different layout, you subclass `UITableViewCell`.

In this section, you will create a custom subclass of `UITableViewCell` named `ItemCell` that will display `Item` instances more effectively. Each of these cells will show a `Item`'s name, its value in dollars, and its serial number (Figure 5.13).

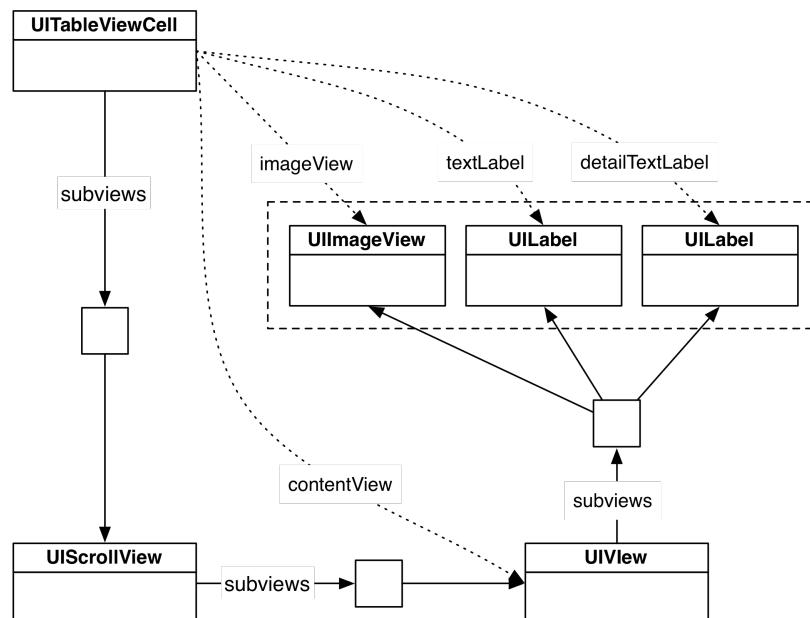
Figure 5.13 Homeowner with subclassed table view cells



`UITableViewCell` is a `UIView` subclass. When subclassing `UIView` (or any of its subclasses), you often override its `drawRect:` method to customize the view's appearance. However, when subclassing `UITableViewCell`, you usually customize its appearance by adding subviews to the cell. You do not add them directly to the cell though; instead you add them to the cell's *content view*.

Each cell has a subview named `contentView`, which is a container for the view objects that make up the layout of a cell subclass (Figure 5.14). When you subclass `UITableViewCell`, you often change its look and behavior by changing the subviews of the cell's `contentView`. For instance, you could create instances of the classes `UITextField`, `UILabel`, and `UIButton` and add them to the `contentView`.

Figure 5.14 UITableViewCell hierarchy

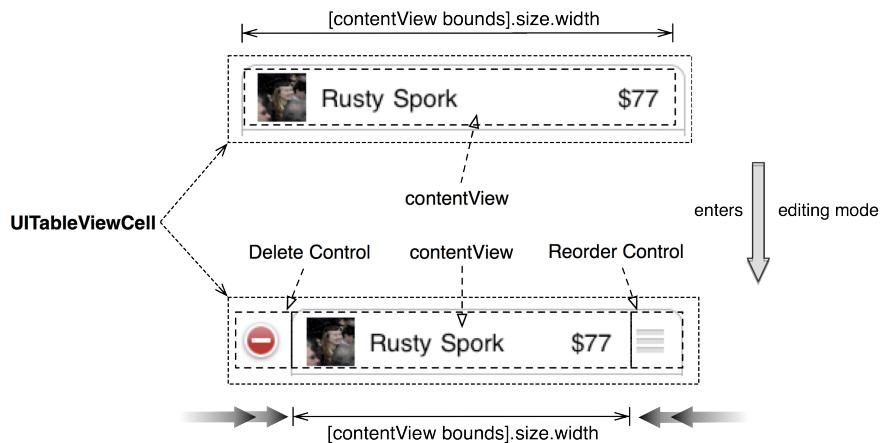


Adding subviews to the `contentView` instead of directly to the cell itself is important because the cell will resize its `contentView` at certain times. For example, when a table view enters editing mode the `contentView` resizes itself to make room for the editing controls (Figure 5.15). If you were to add subviews directly to the `UITableViewCell`, these editing controls would obscure the subviews. The cell cannot adjust its size when entering edit mode (it must remain the width of the table view), but the `contentView` can resize, and it does.

(By the way, notice the `UIScrollView` in the cell hierarchy? That is how iOS moves the contents of the cell to the left when it enters editing mode. You can also use a right-to-left swipe on a cell to show the delete control,

and this uses that same scroll view to get the job done. It makes sense then that the contentView is a subview of the scroll view.)

Figure 5.15 Table view cell layout in standard and editing mode



Creating ItemCell

Create a new Cocoa Touch Class file that is a subclass of `NSObject`. Name it **ItemCell**.

In `ItemCell.swift`, change the superclass to `UITableViewCell`.

```
class ItemCell: NSObject {
    class ItemCell: UITableViewCell {
```

}

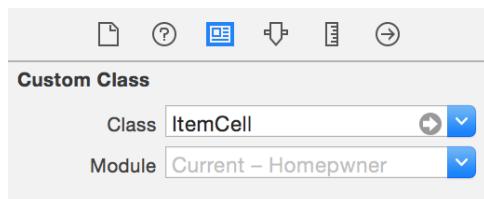
The easiest way to configure a `UITableViewCell` subclass is with a XIB file. Create a new file. From the iOS section, select User Interface, choose the Empty template, and click Next. Name this file `ItemCell.xib`.

This file will contain a single instance of **ItemCell**. When the table view needs a new cell, it will create one from this XIB file.

In `ItemCell.xib`, drag a `UITableViewCell` instance from the object library to the canvas. (Make sure you choose `UITableViewCell`, not `UITableView` or `UITableViewController`.)

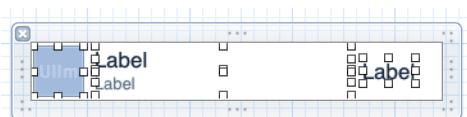
Select the Table View Cell in the outline view and then the identity inspector (the `Identity` tab). Change the Class to **ItemCell** (Figure 5.16).

Figure 5.16 Changing the cell class



An **ItemCell** will display three text elements, so drag three `UILabel` objects onto the cell. Configure them as shown in Figure 5.17. Make the text of the bottom label a slightly smaller font with a dark shade of gray text color.

Figure 5.17 **ItemCell**'s layout



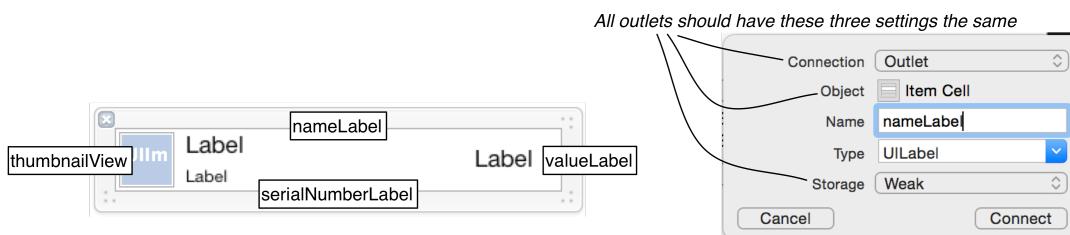
Exposing the properties of ItemCell

In order for `ItemsViewController` to configure the content of an `ItemCell` in `tableView(_:cellForRowAtIndexPath:)`, the cell must have properties that expose the three labels. These properties will be set through outlet connections in `ItemCell.xib`.

The next step, then, is to create and connect outlets on `ItemCell` for each of its subviews.

Option-click on `ItemCell.swift` while `ItemCell.xib` is open. Control-drag from each subview to the method declaration area. Name each outlet and configure the other attributes of the connection as shown in Figure 5.18.

Figure 5.18 `ItemCell` connections



Double-check that `ItemCell.swift` looks like this:

```
import UIKit

class ItemCell: UITableViewCell {

    @IBOutlet weak var nameLabel: UILabel!
    @IBOutlet weak var serialNumberLabel: UILabel!
    @IBOutlet weak var valueLabel: UILabel!

}
```

Note that you did not specify the File's Owner class or make any connections with it. This is a little different than your usual XIB files where all of the connections happen between the File's Owner and the archived objects. To see why, let's see how cells are loaded into the application.

Using ItemCell

In `ItemsViewController`'s `tableView(_:cellForRowAtIndexPath:)` method, you will dequeue an instance of `ItemCell` for every row in the table.

Previously, you registered a class with the table view to inform it which class should be instantiated whenever it needs a new table view cell. Now that you are using a custom NIB file to load a `UITableViewCell` subclass, you will register that NIB instead.

In `ItemsViewController.swift`, modify `viewDidLoad()` to register `ItemCell.xib` for the "ItemCell" reuse identifier.

```
override func viewDidLoad() {
    super.viewDidLoad()

    tableView.registerClass(UITableViewCell.self,
        forCellReuseIdentifier: "UITableViewCell")

    // Load the NIB file
    let nib = UINib(nibName: "ItemCell", bundle: nil)

    // Register this NIB, which contains the cell
    tableView.registerNib(nib,
        forCellReuseIdentifier: "ItemCell")
}
```

The registration of a NIB for a table view is not anything fancy: the table view simply stores the `UINib` instance in a `Dictionary` for the key "ItemCell". A `UINib` contains all of the data stored in its XIB file, and when asked, can create new instances of the objects it contains.

Once a **UINib** has been registered with a table view, the table view can be asked to load the instance of **ItemCell** when given the reuse identifier "ItemCell".

In `ItemsViewController.swift`, modify `tableView(_:cellForRowAtIndexPath:)`.

```
override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("UITableViewCell",
        forIndexPath: indexPath) as UITableViewCell

    // Get a new or recycled cell
    let cell = tableView.dequeueReusableCellWithIdentifier("ItemCell",
        forIndexPath: indexPath) as ItemCell

    let item = itemStore.allItems[indexPath.row]

    cell.textLabel.text = item.name

    // Configure the cell with the Item
    cell.nameLabel.text = item.name
    cell.serialNumberLabel.text = item.serialNumber
    if let value = item.valueInDollars {
        cell.valueLabel.text = "$\$(value)"
    }
    else {
        cell.valueLabel.text = ""
    }

    return cell
}
```

First, the reuse identifier is updated to reflect your new subclass. The code at the end of this method is fairly obvious – for each label on the cell, set its `text` to some property from the appropriate **Item**.

Build and run the application. The new cells now load with their labels populated with the values from each **Item**.

Bronze Challenge: Sections

Have the **UITableView** display two sections – one for items worth more than \$50 and one for the rest. Before you start this challenge, copy the folder containing the project and all of its source files in Finder. Then tackle the challenge in the copied project; you will need the original to build on in the coming chapters.

Silver Challenge: Constant Rows

Make it so the last row of the **UITableView** always has the text No more items!. Make sure this row appears regardless of the number of items in the store (including 0 items).

Gold Challenge: Customizing the Table

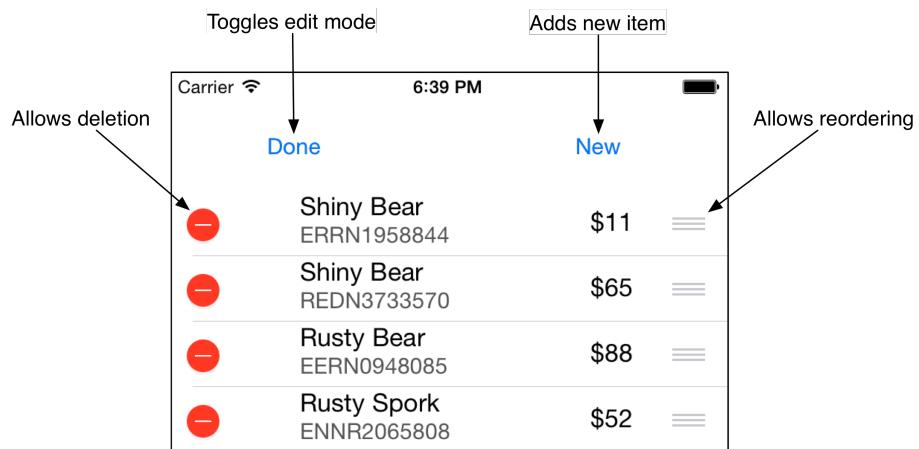
Make each row's height 60 points except for the last row from the silver challenge, which should remain 44 points. Then, change the font size of every row except the last to 20 points. Finally, make the background of the **UITableView** display an image. (To make this pixel-perfect, you will need an image of the correct size depending on your device. Refer to the chart in Chapter 1.)

6

Editing UITableView

In the last chapter, you created an application that displays a list of **Item** instances in a **UITableView**. The next step for Homepwner is allowing the user to interact with the table – to add, delete, and move rows. Figure 6.1 shows what Homepwner will look like by the end of this chapter.

Figure 6.1 Homepwner in editing mode



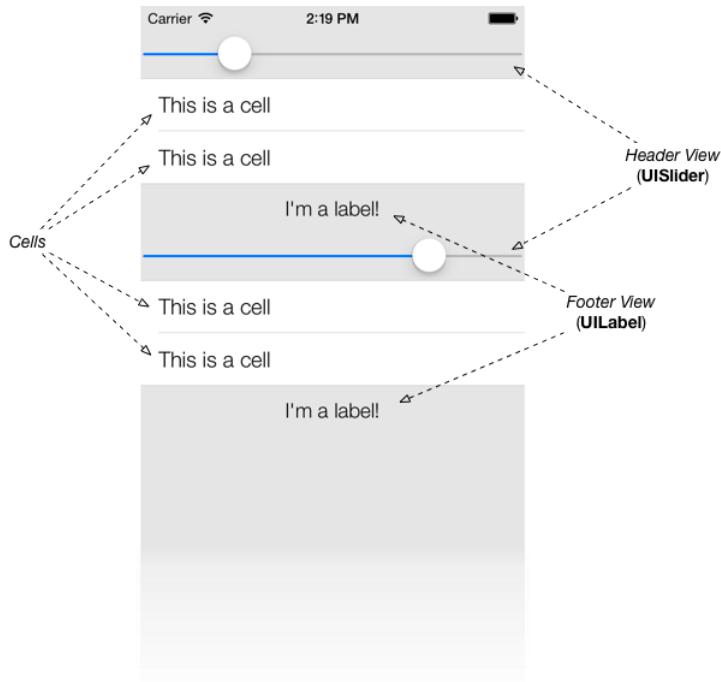
Editing Mode

UITableView has an `editing` property, and when this property is set to `true`, the **UITableView** enters editing mode. Once the table view is in editing mode, the rows of the table can be manipulated by the user. Depending on how the table view is configured, the user can change the order of the rows, add rows, or remove rows. Editing mode does not allow the user to edit the *content* of a row.

But first, the user needs a way to put the **UITableView** in editing mode. For now, you are going to include a button that toggles editing mode in the *header view* of the table. A header view appears at the top of a table and is useful for adding section-wide or table-wide titles and controls. It can be any **UIView** instance.

Note that the table view uses the word “header” in two different ways: There can be a table header and there can be section headers. Likewise, there can be a table footer and section footers.

Figure 6.2 Section headers and footers



You are creating a table header view. It will have two subviews that are instances of **UIButton**: one to toggle editing mode and the other to add a new **Item** to the table. You could create this view programmatically, but in this case you will create the view and its subviews in a XIB file, and **ItemsViewController** will unarchive that XIB file when it needs to display the header view.

First, let's set up the necessary code. Reopen `Homeowner.xcodeproj`. In `ItemsViewController.swift`, add a property for the header view. Also stub out two methods in the implementation.

```
class ItemsViewController: UITableViewController {

    let itemStore: ItemStore
    @IBOutlet var headerView: UIView!

    // Other methods here

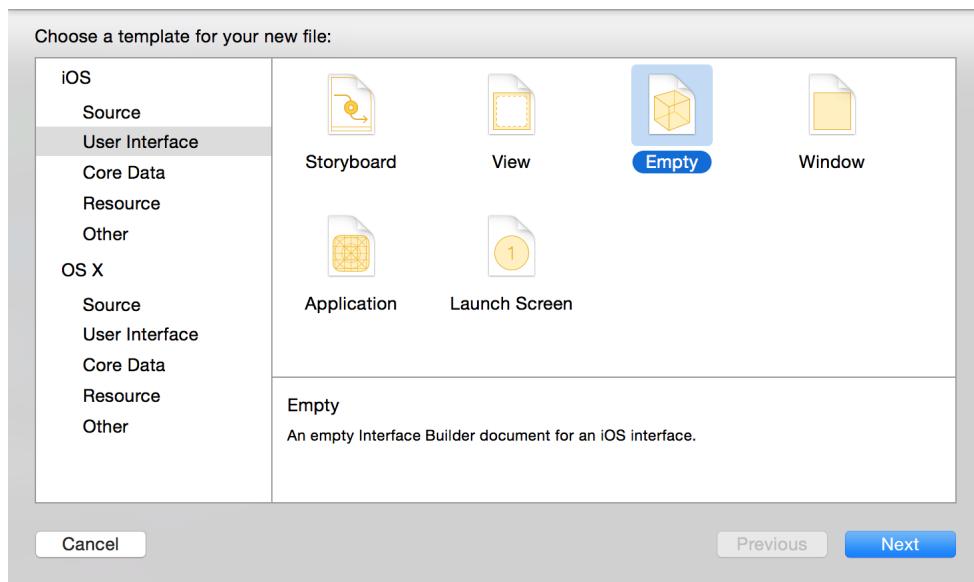
    @IBAction func addNewItem(sender: AnyObject) {
    }

    @IBAction func toggleEditMode(sender: AnyObject) {
    }
}
```

Notice that `headerView` is a strong property. This is because it will be a top-level object in the XIB file; you use weak references for objects that are owned (directly or indirectly) by the top-level objects.

Create a new file (Command-N). From the iOS section, select User Interface, choose the Empty template, and click Next (Figure 6.3).

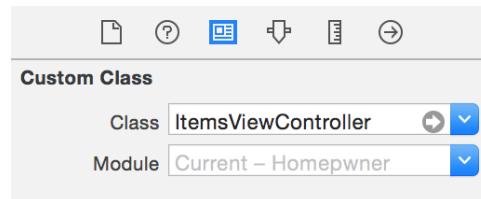
Figure 6.3 Creating a new XIB file



Save this file as `HeaderView`.

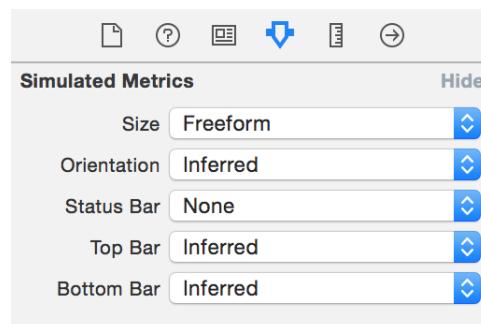
In `HeaderView.xib`, select the File's Owner object and change its Class to **ItemsViewController** in the identity inspector (Figure 6.4).

Figure 6.4 Changing the File's Owner



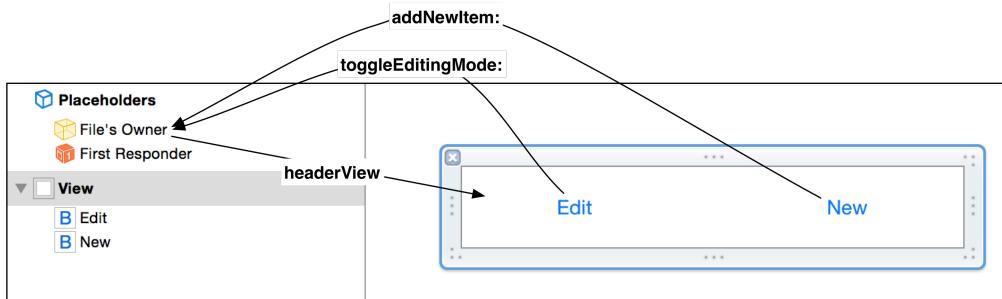
Drag a **UIView** onto the canvas. Then drag two instances of **UIButton** onto that view. You will then want to resize the **UIView** so that it just fits the buttons; however, Xcode will not let you: the size is locked. To unlock the size, select the **UIView** on the canvas and open the attributes inspector. Under the Simulated Metrics section, select Freeform for the Size option. Also, select None for Status Bar (Figure 6.5).

Figure 6.5 Unlocking a view's size



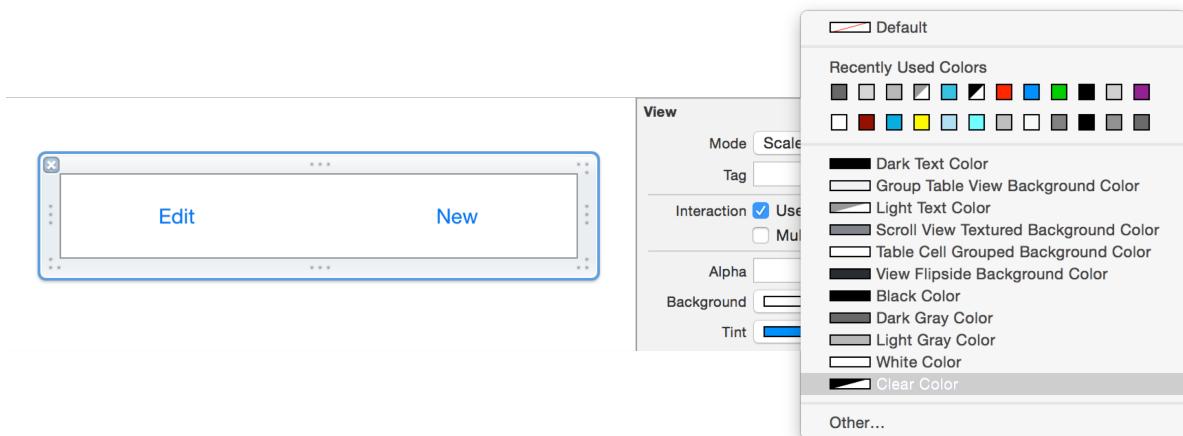
Now that the view can be resized, resize it and make the three connections shown in Figure 6.6.

Figure 6.6 HeaderView.xib layout



Also, change the background color of the **UIView** instance to be completely transparent. To do this, select the view and show the attributes inspector. In the pop-up labeled Background, choose Clear Color (Figure 6.7).

Figure 6.7 Setting background color to clear



So far, your XIB files have been loaded automatically, either by the **UIViewController** (for the view controller's view) or the **UITableView** (for the **UITableViewCell** instances). For HeaderView.xib, you are going to write the code to have the **ItemsViewController** load this XIB file manually.

To load a XIB file manually, you use **NSBundle**. This class is the interface between an application and the application bundle it lives in. When you want to access a file in the application bundle, you ask **NSBundle** for it. An instance of **NSBundle** is created when your application launches, and you can get a pointer to this instance by sending the method **mainBundle()** to **NSBundle**.

Once you have a reference to the main bundle object, you can ask it to load a XIB file. In **ItemsViewController.swift**, load the header view in **viewDidLoad()**.

```
override func viewDidLoad() {
    super.viewDidLoad()

    let nib = UINib(nibName: "ItemCell", bundle: nil)
    tableView.registerNib(nib,
        forCellReuseIdentifier: "ItemCell")

    // Load in the header view
    NSBundle.mainBundle().loadNibNamed("HeaderView", owner: self, options: nil)
}
```

You do not have to specify the suffix of the filename; **NSBundle** will figure it out. Also, notice that you passed **self** as the owner of the XIB file. This ensures that when the main **NSBundle** is parsing the resultant NIB file at runtime, any connections to the File's Owner placeholder will be made to that **ItemsViewController** instance.

Now you just need to tell the table view about its header view. In **ItemsViewController.swift**, add this to the **viewDidLoad()** method:

```
// Load in the header view
NSBundle.mainBundle().loadNibNamed("HeaderView", owner: self, options: nil)

// Set the table view's header view
tableView.tableHeaderView = headerView
```

Build and run the application to see the interface.

While XIB files are often used to create the view for a view controller (for example, `ReminderViewController.xib`), you have now seen that a XIB file can be used any time you wish to archive view objects. In addition, any object can load a XIB file manually by sending the message `loadNibName(_:owner:options:)` to the application bundle.

`UIViewController`'s default XIB loading behavior uses the same code. The only difference is that it connects its view outlet to the view object in the XIB file. Imagine what the default implementation of `loadView()` for `UIViewController` probably looks like:

```
func loadView() {
    // Which bundle is the NIB in?
    // Was a bundle passed to init(nibName:bundle:)?
    var bundle = self.nibBundle
    if bundle == nil {
        // Use the default
        bundle = NSBundle.mainBundle()
    }

    // What is the NIB named?
    // Was a name passed to init(nibName:bundle:)?
    var nibName = self.nibName
    if nibName == nil {
        // Use the default
        nibName = NSStringFromClass(self.dynamicType)
    }

    // Try to find the NIB in the bundle
    var nibPath = bundle.pathForResource(nibName, ofType: "nib")

    // Does it exist?
    if nibPath != nil {
        // Load it (this will set the view outlet as a side-effect)
        bundle.loadNibNamed(nibName, owner: self, options: nil)
    }
    else {
        // If there is no NIB, just create a blank UIView
        view = UIView()
    }
}
```

Now let's implement the `toggleEditMode(_:)` method. You could toggle the `editing` property of `UITableView` directly. However, `UIViewController` also has an `editing` property. A `UIViewController` instance automatically sets the `editing` property of its table view to match its own `editing` property.

To set the `editing` property for a view controller, you send it the message `setEditing(_:animated:)`. In `ItemsViewController.swift`, implement `toggleEditMode(_:)`.

```

@IBAction func toggleEditingStyle(sender: AnyObject) {
    // If you are currently in editing mode...
    if editing {
        // Change text of button to inform user of state
        sender.setTitle("Edit", forState: .Normal)

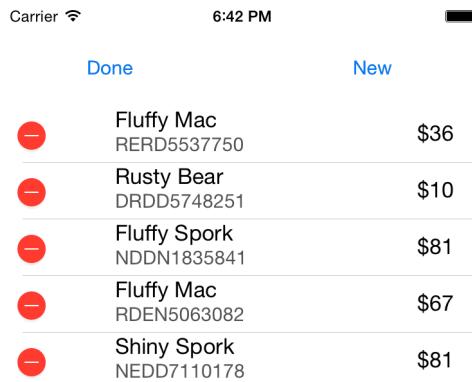
        // Turn off editing mode
        setEditing(false, animated: true)
    }
    else {
        // Change text of button to inform user of state
        sender.setTitle("Done", forState: .Normal)

        // Enter editing mode
        setEditing(true, animated: true)
    }
}

```

Build and run your application, tap the Edit button, and the **UITableView** will enter editing mode (Figure 6.8).

Figure 6.8 **UITableView** in editing mode



	Done	New
–	Fluffy Mac RERD5537750	\$36
–	Rusty Bear DRDD5748251	\$10
–	Fluffy Spork NDDN1835841	\$81
–	Fluffy Mac RDEN5063082	\$67
–	Shiny Spork NEDD7110178	\$81

Adding Rows

There are two common interfaces for adding rows to a table view at runtime.

- A button above the cells of the table view. This is usually for adding a record for which there is a detail view. For example, in the Contacts app, you tap a button when you meet a new person and want to take down all their information.
- A cell with a green plus sign. This is usually for adding a new field to a record, such as when you want to add a birthday to a person's record in the Contacts app. In edit mode, you tap the green plus sign next to "add birthday".

In this exercise, you are using the New button in the header view instead. When this button is tapped, a new row will be added to the **UITableView**.

In **ItemsViewController.swift**, implement **addNewItem(_:)**.

```

@IBAction func addNewItem(sender: AnyObject) {
    // Make a new index path for the 0th section, last row
    let lastRow = tableView.numberOfRowsInSection(0)
    let indexPath = NSIndexPath(forRow: lastRow, inSection: 0)

    // Insert this new row into the table.
    tableView.insertRowsAtIndexPaths([indexPath], withRowAnimation: .Top)
}

```

Build and run the application. Tap the New button and... the application crashes. The console tells you that the table view has an internal inconsistency exception.

Remember that, ultimately, it is the **dataSource** of the **UITableView** that determines the number of rows the table view should display. After inserting a new row, the table view has six rows (the original five plus the

new one). Then, it runs back to its `dataSource` and asks it for the number of rows it should be displaying. `ItemsViewController` consults the store and returns that there should be five rows. The `UITableView` then says, “Hey, that is not right!” and throws an exception.

You must make sure that the `UITableView` and its `dataSource` agree on the number of rows. Thus, you must add a new `Item` to the `ItemStore` before you insert the new row.

In `ItemsViewController.swift`, update `addNewItem(_:)`.

```
@IBAction func addNewItem(sender: AnyObject) {
    let lastRow = tableView.numberOfRowsInSection(0)
    let indexPath = NSIndexPath(forRow: lastRow, inSection: 0)

    // Create a new Item and add it to the store
    let newItem = itemStore.createItem()

    // Figure out where that item is in the array
    if let index = find(itemStore.allItems, newItem) {
        let indexPath = NSIndexPath(forRow: index, inSection: 0)

        // Insert this new row into the table.
        tableView.insertRowsAtIndexPaths([indexPath], withRowAnimation: .Top)
    }
}
```

Build and run the application. Tap the New button and watch the new row slide into the bottom position of the table. Remember that the role of a view object is to present model objects to the user; updating views without updating the model objects is not very useful.

Also, notice that you are accessing the `tableView` property on the `ItemsViewController` to get at the table view. This property is inherited from `UITableViewController`, and it returns the controller’s table view. While you can access the `view` to an instance of `UITableViewController` and get a reference to the same object, using `tableView` tells the compiler that the object returned will be an instance of class `UITableView`. Thus, calling a method that is specific to `UITableView`, like `insertRowsAtIndexPaths(_:withRowAnimation:)`, will not generate an error.

Now that you have the ability to add rows and items, remove the code in the `init(itemStore:)` method in `ItemsViewController.swift` that puts five random items into the store.

```
init(itemStore: ItemStore) {
    self.itemStore = itemStore
    super.init(nibName: nil, bundle: nil)
    for _ in 0...5 {
        self.itemStore.createItem()
    }
}
```

Build and run the application. There will not be any rows when you first fire up the application, but you can add some by tapping the New button.

Deleting Rows

In editing mode, the red circles with the minus sign (shown in Figure 6.8) are deletion controls, and touching one should delete that row. However, at this point, touching a deletion control does not do anything. (Try it and see.) Before the table view will delete a row, it calls a method on its about the proposed deletion and waits for a confirmation before pulling the trigger.

When deleting a cell, you must do two things: remove the row from the `UITableView` and remove the `Item` associated with it from the `ItemStore`. To pull this off, the `ItemStore` must know how to remove objects from itself.

In `ItemStore.swift`, implement a new method to remove a specific item.

```
func removeItem(item: Item) {
    if let index = find(allItems, item) {
        allItems.removeAtIndex(index)
    }
}
```

Now you will implement `tableView(_:commitEditingStyle:forRowAtIndexPath:)`, a method from the `UITableViewDataSource` protocol. (This method is called on the `ItemsViewController`. Keep in mind that while the `ItemStore` is where the data is kept, the `ItemsViewController` is the table view's `dataSource`.)

When `tableView(_:commitEditingStyle:forRowAtIndexPath:)` is called on the data source, two extra arguments are passed along with it. The first is the `UITableViewCellEditingStyle`, which, in this case, is `UITableViewCellEditingStyle.Delete`. The other argument is the `NSIndexPath` of the row in the table.

In `ItemsViewController.swift`, implement this method to have the `ItemStore` remove the right object and to confirm the row deletion by calling the method `deleteRowsAtIndexPaths(_:withRowAnimation:)` back to the table view.

```
override func tableView(tableView: UITableView,
    commitEditingStyle editingStyle: UITableViewCellEditingStyle,
    forRowAtIndexPath indexPath: NSIndexPath) {
    // If the table view is asking to commit a delete command...
    if editingStyle == .Delete {
        let item = itemStore.allItems[indexPath.row]
        // Remove the item from the store
        itemStore.removeItem(item)

        // Also remove that row from the table view with an animation
        tableView.deleteRowsAtIndexPaths([indexPath], withRowAnimation: .Fade)
    }
}
```

Build and run your application, create some rows, and then delete a row. It will disappear. Notice that swipe-to-delete works also.

Moving Rows

To change the order of rows in a `UITableView`, you will use another method from the `UITableViewDataSource` protocol – `tableView(_:moveRowAtIndexPath:toIndexPath:)`.

To delete a row, you had to call the method `deleteRowsAtIndexPaths:withRowAnimation:` on the `UITableView` to confirm the deletion. Moving a row, however, does not require confirmation; the table view moves the row on its own authority and reports the move to its the data source by calling the method `tableView(_:moveRowAtIndexPath:toIndexPath:)`. You just have to implement this method to update your data source to match the new order.

But before you can implement the data source method, you need to give the `ItemStore` a method to change the order of items in its `allItems` array.

In `ItemStore.swift`, implement this new method.

```
func moveItemAtIndex(fromIndex: Int, toIndex: Int) {
    if fromIndex == toIndex {
        return;
    }

    // Get pointer to object being moved so you can re-insert it
    let movedItem = allItems[fromIndex]

    // Remove item from array
    allItems.removeAtIndex(fromIndex)

    // Insert item in array at new location
    allItems.insert(movedItem, atIndex: toIndex)
}
```

In `ItemsViewController.swift`, implement `tableView(_:moveRowAtIndexPath:toIndexPath:)` to update the store.

```
override func tableView(tableView: UITableView,
    moveRowAtIndexPath sourceIndexPath: NSIndexPath,
    toIndexPath destinationIndexPath: NSIndexPath) {
    // Update the model
    itemStore.moveItemAtIndex(sourceIndexPath.row, toIndex: destinationIndexPath.row)
}
```

Build and run your application. Check out the new reordering controls (the three horizontal lines) on the side of each row. Touch and hold a reordering control and move the row to a new position (Figure 6.9).

Figure 6.9 Moving a row

		Done	New
–	Shiny Mac DDRR6358010	\$45	≡
–	Shiny Bear PDNR5125430	\$46	≡
–	Fluffy Spork REER1253801	\$80	≡
–	Fluffy Bear RRDE5602747	\$3	≡
–	Rusty Mac NDRD3097727	\$95	≡

Note that simply implementing `tableView(_:moveRowAtIndexPath:toIndexPath:)` caused the reordering controls to appear. The `UITableView` can ask its data source at runtime whether it implements `tableView(_:moveRowAtIndexPath:toIndexPath:)`. If it does, the table view says, “Good, you can handle moving rows. I’ll add the re-ordering controls.” If not, it says, “If you aren’t implementing that method, then I won’t put controls there.”

Bronze Challenge: Renaming the Delete Button

When deleting a row, a confirmation button appears labeled Delete. Change the label of this button to Remove.

Silver Challenge: Preventing Reordering

Make it so the table view always shows a final row that says No more items! (this part is the same as a challenge from the last chapter. If you have already done it, great!). Then make it so that this row cannot be moved.

Gold Challenge: Really Preventing Reordering

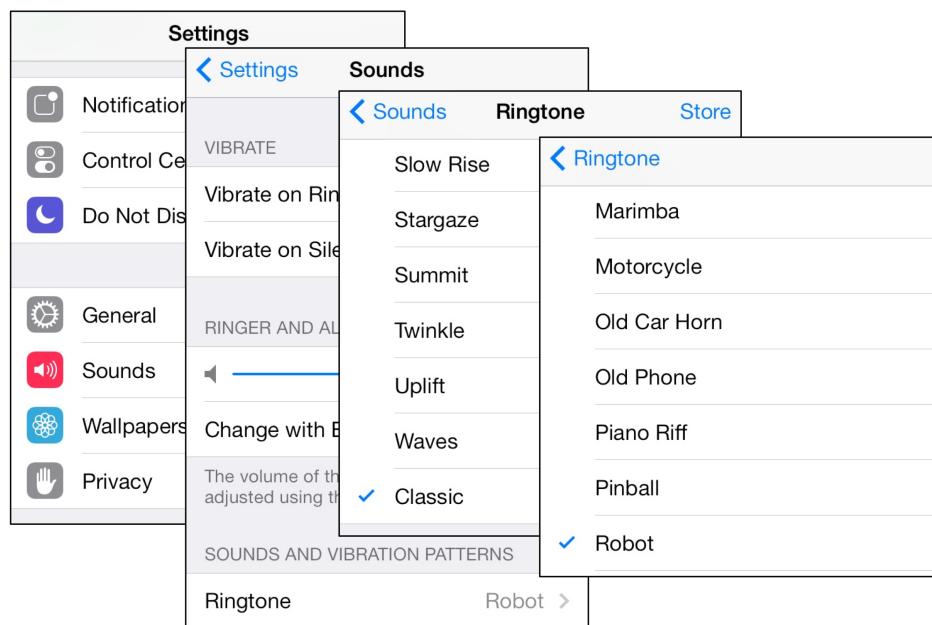
After completing the silver challenge, you may notice that even though you cannot move the No more items! row itself, you can still drag other rows underneath it. Make it so that no matter what, the No more items! row can never be knocked out of the last position. Finally, make it undeletable.

UINavigationController

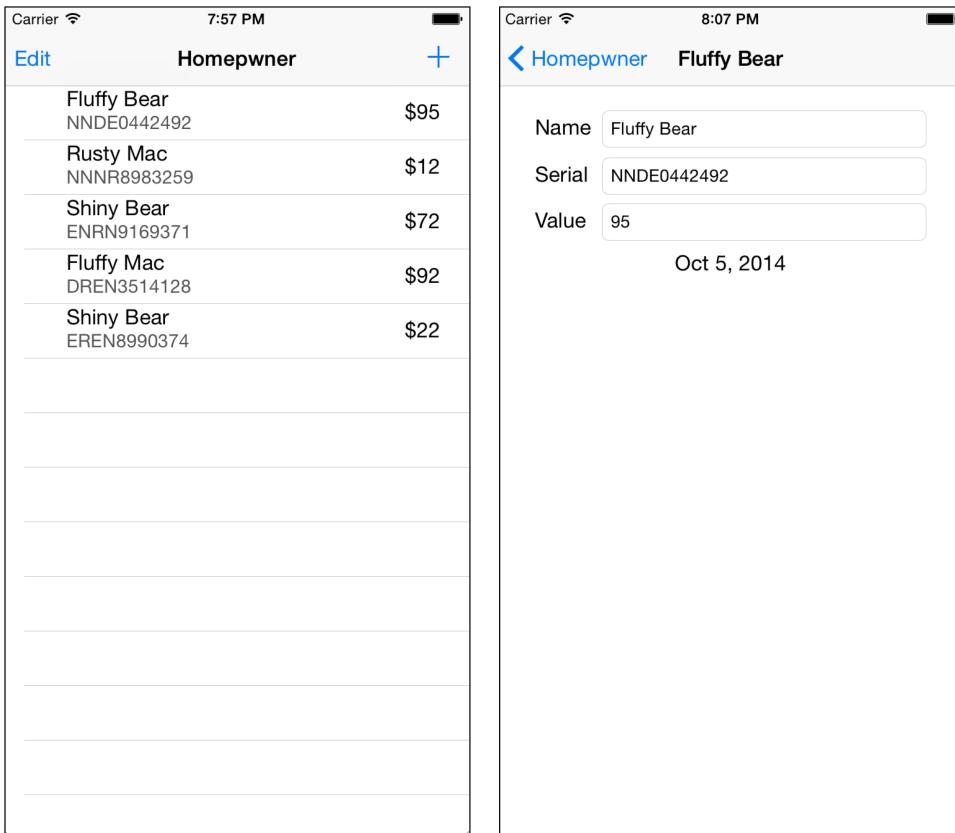
In Chapter 4, you learned about **UITabBarController** and how it allows a user to access different screens. A tab bar controller is great when you have screens that do not rely on each other, but what if you want to move between related screens?

For example, the **Settings** application has multiple related screens of information: a list of settings (like Sounds), a detailed page for each setting, and a selection page for each detail. This type of interface is called a *drill-down interface*.

Figure 7.1 Settings has a drill-down interface



In this chapter, you will use a **UINavigationController** to add a drill-down interface to Homeowner that lets the user view and edit the details of an **Item** (Figure 7.2).

Figure 7.2 Homeowner with **UINavigationController**

UINavigationController

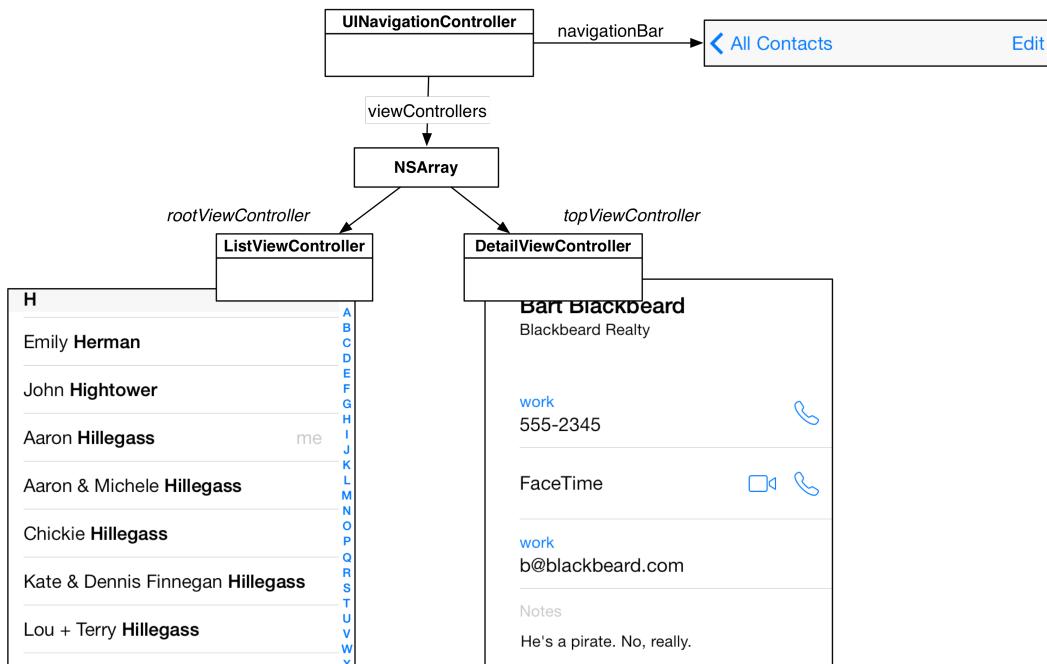
When your application presents multiple screens of information, a **UINavigationController** maintains a stack of those screens. Each screen is the view of a **UIViewController**, and the stack is an array of view controllers. When a **UIViewController** is on top of the stack, its view is visible.

When you initialize an instance of **UINavigationController**, you give it one **UIViewController**. This **UIViewController** is the navigation controller's *root view controller*. The root view controller is always on the bottom of the stack. More view controllers can be pushed on top of the **UINavigationController**'s stack while the application is running.

When a **UIViewController** is pushed onto the stack, its view slides onto the screen from the right. When the stack is popped, the top view controller is removed from the stack and its view slides off to the right, exposing the view of next view controller on the stack.

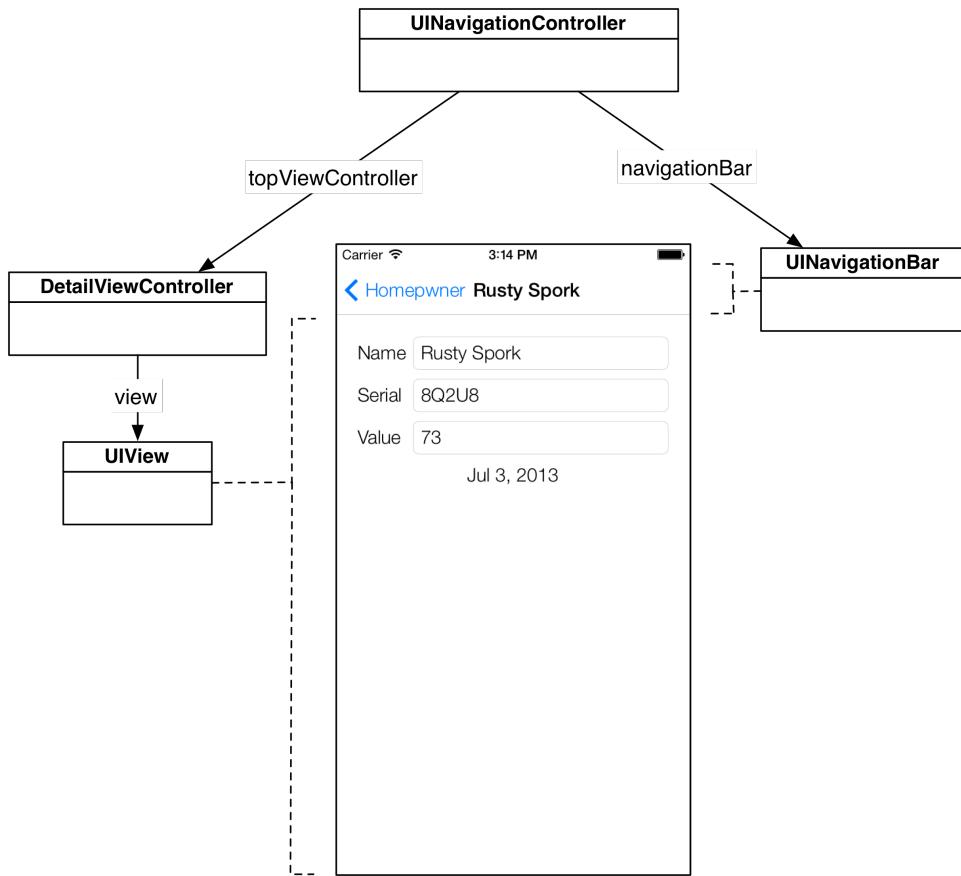
Figure 7.3 shows a navigation controller with two view controllers: a root view controller and an additional view controller above it at the top of the stack. The view of the additional view controller is what the user sees because that view controller is at the top of the stack.

Figure 7.3 UINavigationController's stack



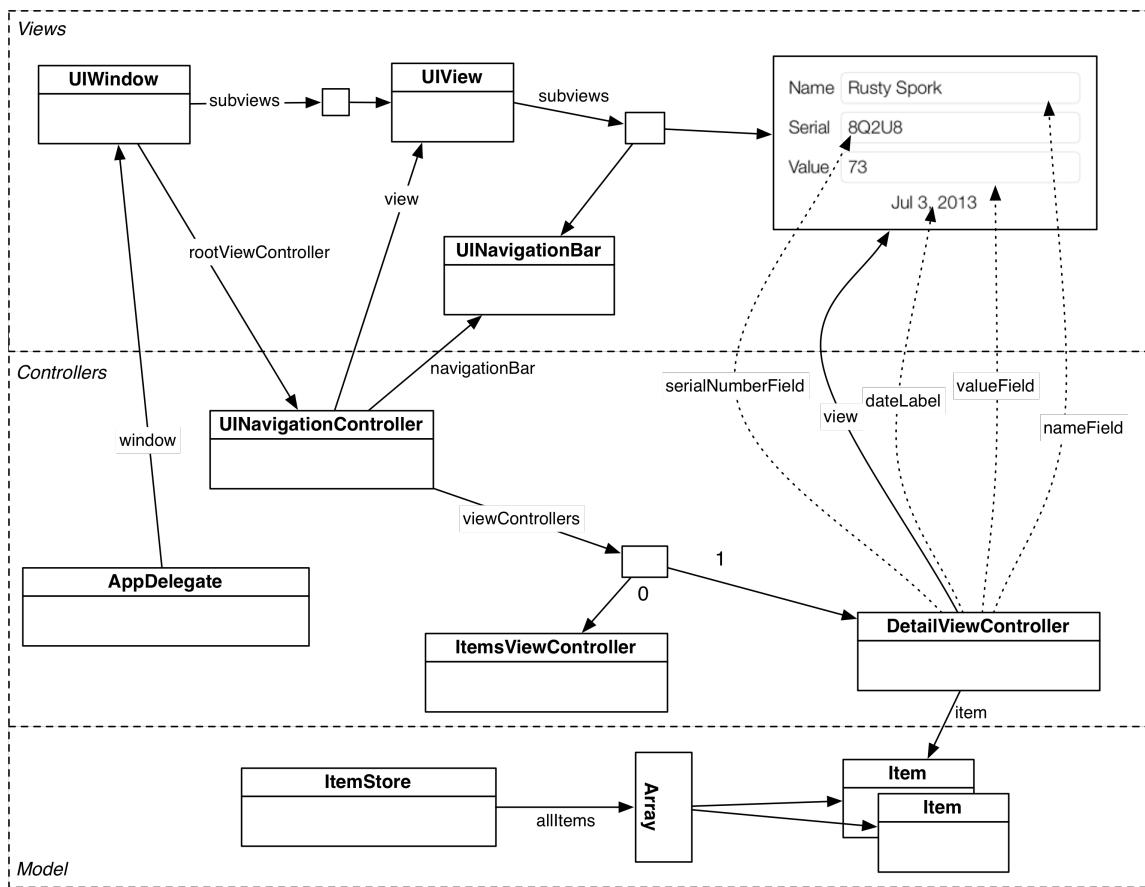
Like **UITabBarController**, **UINavigationController** has a `viewControllers` array. The root view controller is the first object in the array. As more view controllers are pushed onto the stack, they are added to the end of this array. Thus, the last view controller in the array is the top of the stack. **UINavigationController**'s `topViewController` property keeps a pointer to the top of the stack.

UINavigationController is a subclass of **UIViewController**, so it has a `view` of its own. Its `view` always has two subviews: a **UINavigationBar** and the `view` of `topViewController` (Figure 7.4). You can set a navigation controller as the `rootViewController` of the window to make its `view` a subview of the window.

Figure 7.4 A **UINavigationController**'s view

In this chapter, you will add a **UINavigationController** to the Homeowner application and make the **ItemsViewController** the **UINavigationController**'s **rootViewController**. Then, you will create another subclass of **UIViewController** that can be pushed onto the **UINavigationController**'s stack. When a user selects one of the rows, the new **UIViewController**'s view will slide onto the screen. This view controller will allow the user to view and edit the properties of the selected **Item**. The object diagram for the updated Homeowner application is shown in Figure 7.5.

Figure 7.5 Homepwner object diagram



This application is getting fairly large, as you can see in the massive object diagram. Fortunately, view controllers and **UINavigationController** know how to deal with this type of complicated object diagram. When writing iOS applications, it is important to treat each **UIViewController** as its own little world. The stuff that has already been implemented in Cocoa Touch will do the heavy lifting.

Now let's give Homepwner a navigation controller. Reopen the Homepwner project and then open **AppDelegate.swift**. The only requirements for using a **UINavigationController** are that you give it a root view controller and add its view to the window.

In **AppDelegate.swift**, create the **UINavigationController** in **application(_:didFinishLaunchingWithOptions:)**, give it a root view controller of its own, and set the **UINavigationController** as the root view controller of the window.

```

func application(application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
    // Override point for customization after application launch.
    window = UIWindow(frame: UIScreen.mainScreen().bounds)

    // Create an ItemStore
    let itemStore = ItemStore()

    // Create an ItemsViewController and inject ItemStore
    let ivc = ItemsViewController(itemStore: itemStore)

    // Create an instance of a UINavigationController
    // its stack contains only ivc
    let navController = UINavigationController(rootViewController: ivc)

window!.rootViewController = ivc
    // Place navigation controller's view in the window hierarchy
    window!.rootViewController = navController

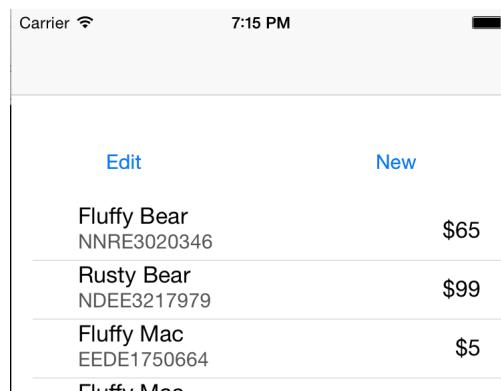
    window!.backgroundColor = UIColor.whiteColor()
    window!.makeKeyAndVisible()

    return true
}

```

Build and run the application. Homepwner will look the same as it did before – except now it has a **UINavigationBar** at the top of the screen (Figure 7.6). Notice how **ItemsViewController**'s view was resized to fit the screen with a navigation bar. **UINavigationController** did this for you.

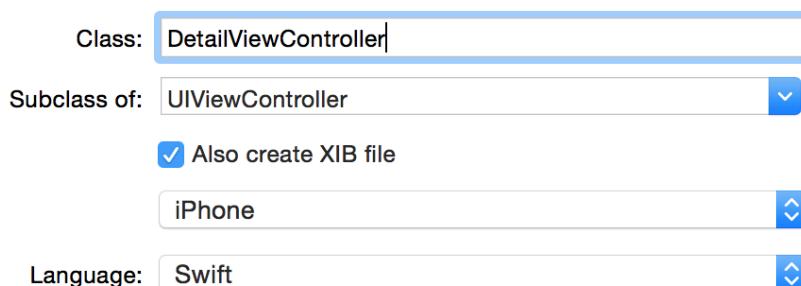
Figure 7.6 Homepwner with an empty navigation bar



An Additional UIViewController

To see the real power of **UINavigationController**, you need another **UIViewController** to put on the navigation controller's stack. Create a new Cocoa Touch Class (File → New → File...). Name this class **DetailViewController** and choose **UIViewController** as the superclass. Check the **Also create XIB file** box (Figure 7.7).

Figure 7.7 Create **UIViewController** subclass with XIB



In `DetailViewController.swift`, delete all of the code between the curly braces so that the file looks like this:

```
import UIKit

class DetailViewController: UIViewController {
```

In `Homepwner`, you want the user to be able to tap an item to get another screen with editable text fields for each property of that `Item`. This view will be controlled by an instance of `DetailViewController`.

The detail view needs four subviews – one for each property of a `Item` instance. And because you need to be able to access these subviews during runtime, `DetailViewController` needs outlets for these subviews. The plan is to add four new outlets to `DetailViewController`, drag the subviews onto the view in the XIB file, and then make the connections.

In previous exercises, these were three distinct steps: you added the outlets in the interface file, then you configured the interface in the XIB file, and then you made connections. You can combine these steps using a shortcut in Xcode. First, open `DetailViewController.xib` by selecting it in the project navigator.

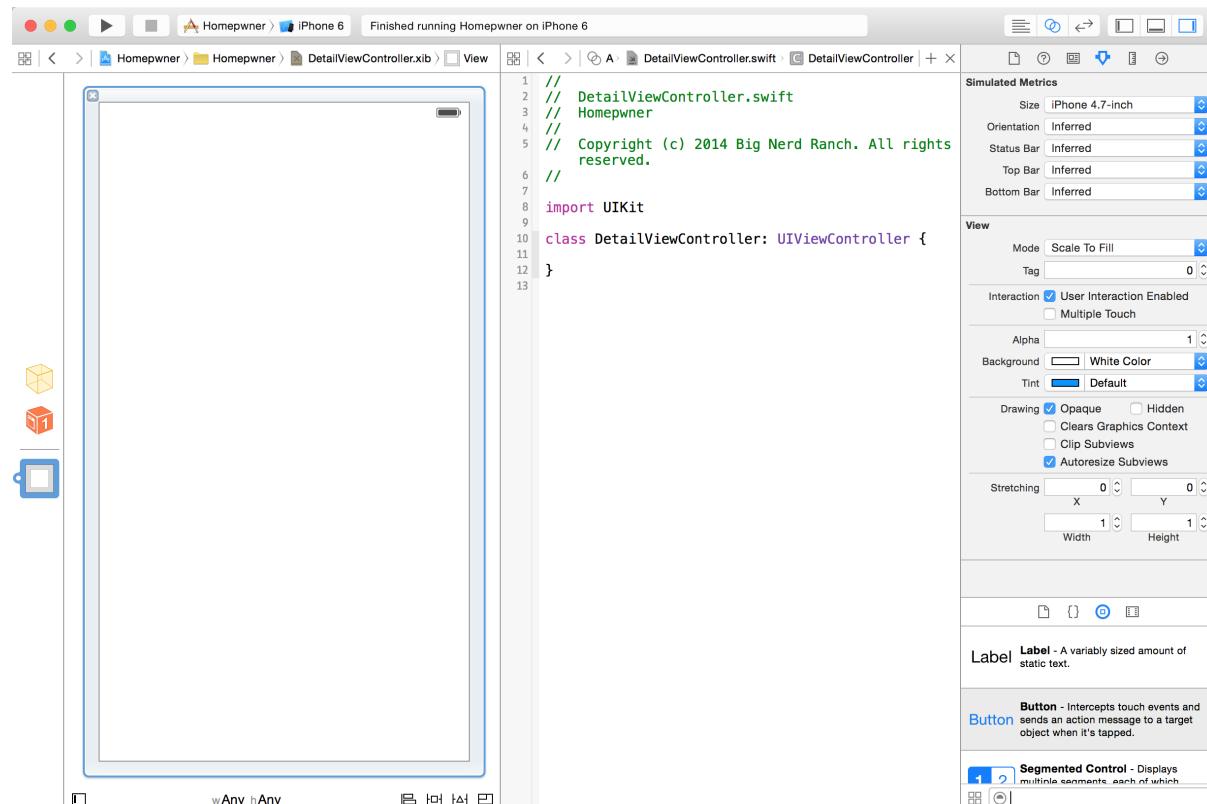
Select the View and open its Attributes inspector. Under Simulated Metrics, change the Size to iPhone 4.7-inch.

Now, Option-click on `DetailViewController.swift` in the project navigator. This shortcut opens the file in the *assistant editor*, right next to `DetailViewController.xib`. (You can toggle the assistant editor by clicking the middle button from the Editor control at the top of the workspace; the shortcut to display the assistant editor is Command-Option-Return; to return to the standard editor, use Command-Return.)

You will also need the object library available so that you can drag the subviews onto the view. Show the utility area by clicking the right button in the View control at the top of the workspace (or Command-Option-0).

Your window is now sufficiently cluttered. Let's make some temporary space. Hide the navigator area by clicking the left button in the View control at the top of the workspace (the shortcut for this is Command-0). Then, change the dock in Interface Builder to show the icon view by clicking the toggle button in the lower left corner of the editor. Your workspace should now look like Figure 7.8.

Figure 7.8 Laying out the workspace



Now, drag four **UILabel** objects and three **UITextField** objects onto the view in the canvas area and configure them to look like Figure 7.9.

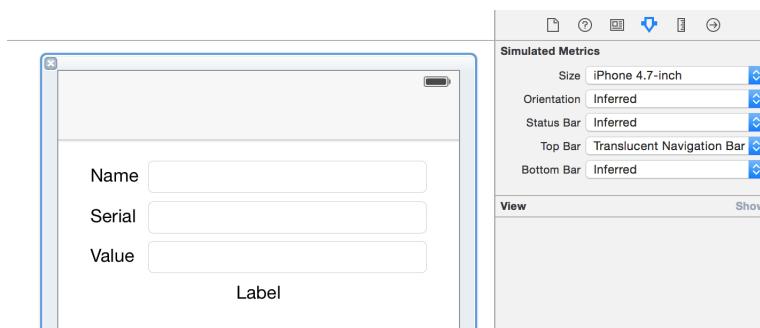
Figure 7.9 Configured **DetailViewController** XIB



It is important that the subviews you just added are not positioned near the very top of the XIB. This is because the view of a **UIViewController** extends beneath the **UINavigationBar** (this is also true for the **UITabBar**). To make configuring interfaces easier, the root level view in a XIB file has *simulated metrics* that will show you what the interface will look like with a navigation bar at the top. You can also preview a tab bar along the bottom and a number of other situations that your interface might find itself in.

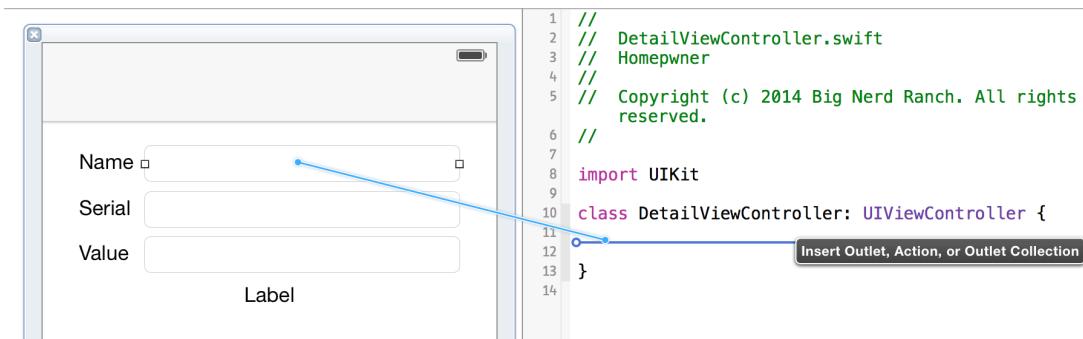
To see the simulated metrics, select the root level view, and open its attributes inspector. At the top, you will see Simulated Metrics. For Top Bar, choose Translucent Navigation Bar (Figure 7.10).

Figure 7.10 Simulated metrics



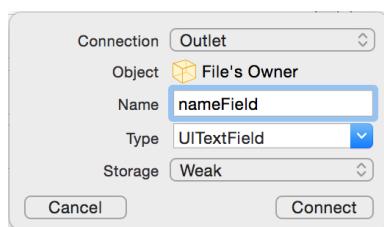
The three instances of **UITextField** and bottom instance of **UILabel** will be outlets in **DetailViewController**. Here comes the exciting part. Control-drag from the **UITextField** next to the Name label to the top of **DetailViewController.swift**, as shown in Figure 7.11.

Figure 7.11 Dragging from XIB to source file



Let go and a pop-up window will appear. Enter `nameField` into the Name field, select Weak from the Storage pop-up menu, and click Connect (Figure 7.12).

Figure 7.12 Auto-generating an outlet and making a connection



This will create an `IBOutlet` property of type `UITextField` named `nameField` in `DetailViewController`. You chose Weak storage for this property because the object it will point to is not a top-level object in the XIB file.

In addition, this `UITextField` is now connected to the `nameField` outlet of the File's Owner in the XIB file. You can verify this by Control-clicking on the File's Owner to see the connections. Also notice that hovering your mouse above the `nameField` connection in the panel that appears will reveal the `UITextField` that you connected. Two birds, one stone.

Create the other three outlets the same way and name them as shown in Figure 7.13.

Figure 7.13 Connection diagram



After making the connections, `DetailViewController.swift` should look like this:

```

import UIKit

class DetailViewController: UIViewController {

    @IBOutlet weak var nameField: UITextField!
    @IBOutlet weak var serialNumberField: UITextField!
    @IBOutlet weak var valueField: UITextField!
    @IBOutlet weak var dateLabel: UILabel!

}

```

If your file looks different, then your outlets are not connected right.

Fix any disparities between your file and the code shown above in three steps: First, go through the Control-drag process and make connections again until you have the four lines shown above in your `DetailViewController.swift`. Second, remove any wrong code (like non-property method declarations or instance variables) that got created. Finally, check for any bad connections in the XIB file. In `DetailViewController.xib`, Control-click on the File's Owner. If there are yellow warning signs next to any connection, click the x icon next to those connections to disconnect them.

It is important to ensure there are no bad connections in a XIB file. A bad connection typically happens when you change the name of an instance variable but do not update the connection in the XIB file. Or, you completely remove an instance variable but do not remove it from the XIB file. Either way, a bad connection will cause your application to crash when the XIB file is loaded.

Now let's make more connections. For each instance of `UITextField` in the XIB file, connect the `delegate` property to the File's Owner. (Remember, Control-drag from the `UITextField` to the File's Owner and select `delegate` from the list.)

Finally, add the initializers for `DetailViewController`. The designated initializer will be `init(itemStore:)` that will set an `ItemStore` property. Additionally, you'll implement the required `init(coder:)` to have it fail if someone tries to use it.

```
class DetailViewController: UIViewController {

    @IBOutlet weak var nameField: UITextField!
    @IBOutlet weak var serialNumberField: UITextField!
    @IBOutlet weak var valueField: UITextField!
    @IBOutlet weak var dateLabel: UILabel!

    let itemStore: ItemStore

    init(itemStore: ItemStore) {
        self.itemStore = itemStore
        super.init(nibName: "DetailViewController", bundle: nil)
    }

    required init(coder aDecoder: NSCoder) {
        fatalError("User init(itemStore:)")
    }
}
```

Now that this project has a good number of source files, you will be switching between them fairly regularly. One way to speed up switching between commonly accessed files is to use Xcode tabs. If you double-click on a file in the project navigator, the file will open in a new tab. You can also open up a blank tab with the shortcut Command-T. The keyboard shortcuts for cycling through tabs are Command-Shift-} and Command-Shift-{. (You can see the other shortcuts for project organization by selecting the General tab from Xcode's preferences.)

Navigating with UINavigationController

Now you have a navigation controller and two view controller subclasses. Time to put the pieces together. The user should be able to tap a row in `ItemsViewController`'s table view and have the `DetailViewController`'s view slide onto the screen and display the properties of the selected `Item` instance.

Pushing view controllers

Of course, you need to create an instance of `DetailViewController`. Where should this object be created? Think back to previous exercises where you instantiated all of your controllers in the method `application(_:didFinishLaunchingWithOptions:)`. For example, in Chapter 4, you created both view controllers and immediately added them to the tab bar controller's `viewControllers` array.

However, when using a `UINavigationController`, you cannot simply store all of the possible view controllers in its stack. The `viewControllers` array of a navigation controller is dynamic – you start with a root view

controller and push view controllers depending on user input. Therefore, some object other than the navigation controller needs to create the instance of **DetailViewController** and be responsible for adding it to the stack.

This object needs to know when to push a **DetailViewController** onto the stack.

ItemsViewController fills this requirement. It knows when a row is tapped in a table view because, as the table view's delegate, it receives the message `tableView(_:didSelectRowAt IndexPath:)` when this event occurs.

Therefore, **ItemsViewController** will be responsible for creating the instance of **DetailViewController** and adding it to the stack.

When a row is tapped in a table view, its delegate is sent `tableView(_:didSelectRowAt IndexPath:)`, which contains the index path of the selected row.

In `ItemsViewController.swift`, implement this method to create a **DetailViewController** and push it on top of the navigation controller's stack.

```
override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath) {
    // Create a DetailViewController
    let dvc = DetailViewController(itemStore: itemStore)

    // Push it on to the navigation controller's stack
    showViewController(dvc, sender: self)
}
```

Build and run the application. Create a new item and select that row from the **UITableView**. Not only are you taken to **DetailViewController**'s view, but you also get a free animation and a back button in the **UINavigationBar**. Tap this button to get back to **ItemsViewController**.

Since the **UINavigationController**'s stack is an array, it will take ownership of any view controller added to it. Thus, the **DetailViewController** is owned only by the **UINavigationController** after `tableView(_:didSelectRowAt IndexPath:)` finishes. When the stack is popped, the **DetailViewController** is destroyed. The next time a row is tapped, a new instance of **DetailViewController** is created.

Having a view controller push the next view controller is a common pattern. The root view controller typically creates the next view controller, and the next view controller creates the one after that, and so on. Some applications may have view controllers that can push different view controllers depending on user input. For example, the Photos app pushes a video view controller or an image view controller onto the navigation stack depending on what type of media was selected.

Passing data between view controllers

Of course, the text fields on the screen are currently empty. To fill these fields, you need a way to pass the selected **Item** from the **ItemsViewController** to the **DetailViewController**.

To pull this off, you will give **DetailViewController** a property to hold a **Item**. When a row is tapped, **ItemsViewController** will give the corresponding **Item** to the instance of **DetailViewController** that is being pushed onto the stack. The **DetailViewController** will populate its text fields with the properties of that **Item**. Editing the text in the text fields on **DetailViewController**'s view will change the properties of that **Item**.

In `DetailViewController.swift`, add this property. Also, update the initializer to take in an **Item** instance and set it as the **item** of the **DetailViewController**.

```
class DetailViewController: UIViewController {

    @IBOutlet weak var nameField: UITextField!
    @IBOutlet weak var serialNumberField: UITextField!
    @IBOutlet weak var valueField: UITextField!
    @IBOutlet weak var dateLabel: UILabel!

    var itemStore: ItemStore
    let item: Item

    init(item: Item, itemStore: ItemStore) {
        self.item = item
        self.itemStore = itemStore
        super.init(nibName: "DetailViewController", bundle: nil)
    }
}
```

When the **DetailViewController**'s view appears on the screen, it needs to set up its subviews to show the properties of the item. In **DetailViewController.swift**, override **viewWillAppear(_:**) to transfer the item's properties to the various instances of **UITextField**.

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)

    nameField.text = item?.name
    serialNumberField.text = item?.serialNumber
    valueField.text = "\((item?.valueInDollars ?? 0))"

    let date = item.dateCreated {
        let dateFormatter = NSDateFormatter()
        dateFormatter.dateStyle = .MediumStyle
        dateFormatter.timeStyle = .NoStyle

        dateLabel.text = dateFormatter.stringFromDate(date)
    }
}
```

In **ItemsViewController.swift**, add the following code to **tableView(_:didSelectRowAtIndexPath:)** so that **DetailViewController** has its item when the **DetailViewController** is created.

```
override func tableView(tableView: UITableView,
    didSelectRowAtIndexPath indexPath: NSIndexPath) {

    // Get the selected item
    let item = itemStore.allItems[indexPath.row]

    // Create a DetailViewController
    let dvc = DetailViewController(item: item, itemStore: itemStore)

    // Push it on to the navigation controller's stack
    showViewController(dvc, sender: self)
}
```

Many programmers new to iOS struggle with how data is passed between view controllers. Having all of the data in the root view controller and passing subsets of that data to the next **UIViewController** (like you just did) is a clean and efficient way of performing this task.

Build and run your application. Create a new item and select that row in the **UITableView**. The view that appears will contain the information for the selected **Item**. While you can edit this data, the **UITableView** will not reflect those changes when you return to it. To fix this problem, you need to implement code to update the properties of the **Item** being edited. In the next section, you will see when to do this.

Appearing and disappearing views

Whenever a **UINavigationController** is about to swap views, it calls two methods: **viewWillDisappear(_:**) and **viewWillAppear(_:**). The **UIViewController** that is about to be popped off the stack has **viewWillDisappear(_:**) called. The **UIViewController** that will then be on top of the stack has **viewWillAppear(_:**) called.

When a **DetailViewController** is popped off the stack, you will set the properties of its **item** to the contents of the text fields. When implementing these methods for views appearing and disappearing, it is important to call the superclass's implementation – it might have some work to do and needs to be given the chance to do it. In **DetailViewController.swift**, implement **viewWillDisappear(_:**).

```
override func viewWillDisappear(animated: Bool) {
    super.viewWillDisappear(animated)

    // Clear first responder
    view.endEditing(true)

    // "Save" changes to item
    item.name = nameField.text
    item.serialNumber = serialNumberField.text
    item.valueInDollars = valueField.text.toInt()
}
```

Notice the use of `endEditing(_:)`. When the message `endEditing(_:)` is sent to a view, if it or any of its subviews is currently the first responder, it will resign its first responder status, and the keyboard will be dismissed. (The argument passed determines whether the first responder should be forced into retirement. Some first responders might refuse to resign, and passing `true` ignores that refusal.)

Now the values of the `Item` will be updated when the user taps the Back button on the `UINavigationBar`. When `ItemsViewController` appears back on the screen, it is sent the message `viewWillAppear(_:)`. Take this opportunity to reload the `UITableView` so the user can immediately see the changes. In `ItemsViewController.swift`, override `viewWillAppear(_:)`.

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)

    tableView.reloadData()
}
```

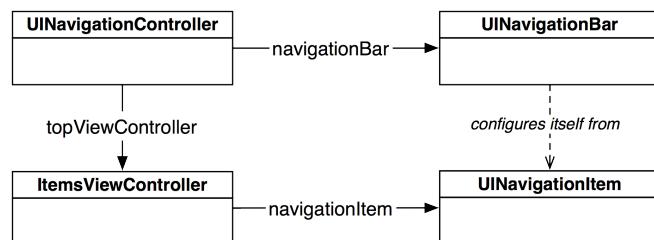
Build and run your application now. Now you can move back and forth between the view controllers that you created and change the data with ease.

UINavigationBar

The `UINavigationBar` is not very interesting right now. A `UINavigationBar` should display a descriptive title for the `UIViewController` that is currently on top of the `UINavigationController`'s stack.

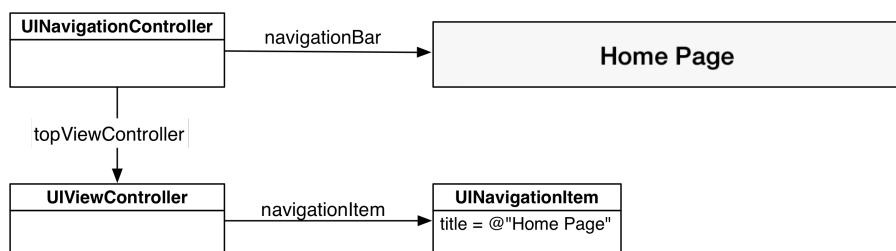
Every `UIViewController` has a `navigationItem` property of type `UINavigationItem`. However, unlike `UINavigationBar`, `UINavigationItem` is not a subclass of `UIView`, so it cannot appear on the screen. Instead, the navigation item supplies the navigation bar with the content it needs to draw. When a `UIViewController` comes to the top of a `UINavigationController`'s stack, the `UINavigationBar` uses the `UIViewController`'s `navigationItem` to configure itself, as shown in Figure 7.14.

Figure 7.14 `UINavigationItem`



By default, a `UINavigationItem` is empty. At the most basic level, a `UINavigationItem` has a simple title string. When a `UIViewController` is moved to the top of the navigation stack and its `navigationItem` has a valid string for its `title` property, the navigation bar will display that string (Figure 7.15).

Figure 7.15 `UINavigationItem` with title



In `ItemsViewController.swift`, modify `init(itemStore:)` to set the `navigationItem`'s `title` to read `Homeowner`.

```
init(itemStore: ItemStore) {
    self.itemStore = itemStore
    super.init(nibName: nil, bundle: nil)

    navigationItem.title = "Homepwner"
}
```

Build and run the application. Notice the string Homepwner on the navigation bar. Create and tap on a row and notice that the navigation bar no longer has a title. You need to give the **DetailViewController** a title, too. It would be nice to have the **DetailViewController**'s navigation item title be the name of the **Item** it is displaying. Obviously, you cannot do this in **init** because you do not yet know what its **item** will be.

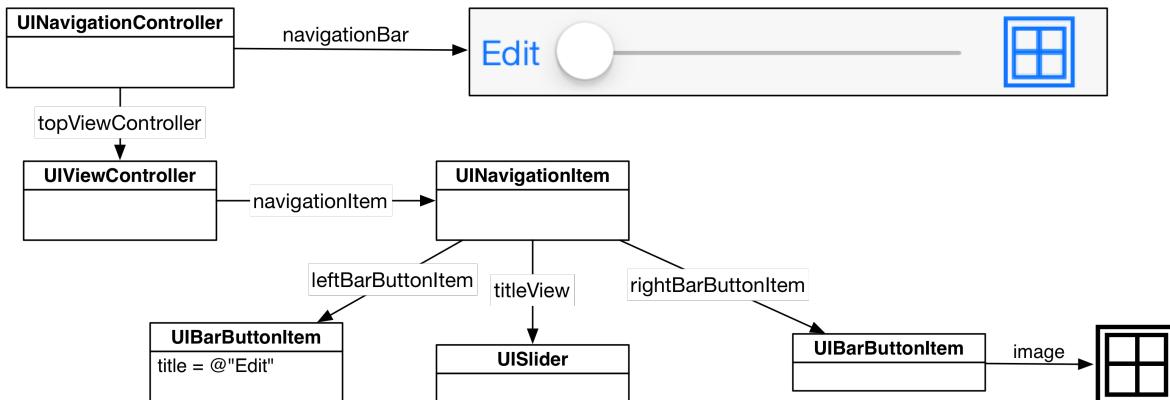
Instead, the **DetailViewController** will set its title when it sets its **item** property. In **DetailViewController.swift**, implement a property setter on **item**.

```
let item: Item {
    didSet {
        navigationItem.title = item?.name
    }
}
```

Build and run the application. Create and tap a row, and you will see that the title of the navigation bar is the name of the **Item** you selected.

A navigation item can hold more than just a title string, as shown in Figure 7.16. There are three customizable areas for each **UINavigationItem**: a **leftBarButtonItem**, a **rightBarButtonItem**, and a **titleView**. The left and right bar button items are pointers to instances of **UIBarButtonItem**, which contains the information for a button that can only be displayed on a **UINavigationBar** or a **UIToolbar**.

Figure 7.16 **UINavigationItem** with everything



Like **UINavigationItem**, **UIBarButtonItem** is not a subclass of **UIView**. Instead, **UINavigationItem** encapsulates information that **UINavigationBar** uses to configure itself. Similarly, **UIBarButtonItem** is not a view, but holds the information about how a single button on the **UINavigationBar** should be displayed. (A **UIToolbar** also uses instances of **UIBarButtonItem** to configure itself.)

The third customizable area of a **UINavigationItem** is its **titleView**. You can either use a basic string as the title or have a subclass of **UIView** sit in the center of the navigation item. You cannot have both. If it suits the context of a specific view controller to have a custom view (like a button, a slider, an image, or even a map), you would set the **titleView** of the navigation item to that custom view. Figure 7.16 shows an example of a **UINavigationItem** with a custom view as its **titleView**. Typically, however, a title string is sufficient, and that is what you will do in this chapter.

Let's add a **UIBarButtonItem** to the **UINavigationBar**. You want this button to sit on the right side of the navigation bar when the **ItemsViewController** is on top of the stack. When tapped, it should add a new **Item** to the list.

A bar button item has a target-action pair that works like **UIControl**'s target-action mechanism: when tapped, it sends the action message to the target. When you set a target-action pair in a XIB file, you Control-drag from

a button to its target and then select a method from the list of @IBActions. To programmatically set up a target-action pair, you pass the target and the action to the button.

In `ItemsViewController.swift`, create a `UIBarButtonItem` instance and give it its target and action.

```
init(itemStore: ItemStore) {
    self.itemStore = itemStore
    super.init(nibName: nil, bundle: nil)

    navigationItem.title = "Homepwner"

    // Create a new bar button item that will send
    // addNewItem(_) to ItemsViewController
    let addItem = UIBarButtonItem(barButtonSystemItem: .Add,
        target: self,
        action: "addNewItem:")

    // Set this bar button item as the right item in the navigationItem
    navigationItem.rightBarButtonItem = addItem
}
```

Build and run the application. Tap the + button, and a new row will appear in the table. (Note that this is not the only way to set up a bar button item; check the documentation for other initialization messages that you can use to create an instance of `UIBarButtonItem`.)

Now let's add another `UIBarButtonItem` to replace the Edit button in the table view header. In `ItemsViewController.swift`, edit the `init(itemStore:)` method.

```
init(itemStore: ItemStore) {
    self.itemStore = itemStore
    super.init(style: .Plain)

    navigationItem.title = "Homepwner"

    // Create a new bar button item that will send
    // addNewItem(_) to ItemsViewController
    let addItem = UIBarButtonItem(barButtonSystemItem: .Add,
        target: self,
        action: "addNewItem:")

    // Set this bar button item as the right item in the navigationItem
    navigationItem.rightBarButtonItem = addItem

    navigationItem.leftBarButtonItem = editButtonItem()
}
```

Surprisingly, that is all the code you need to get an edit button on the navigation bar. Build and run, tap the Edit button, and watch the `UITableView` enter editing mode! Where does `editButtonItem()` come from? `UIViewController` has an `editButtonItem` property, and when sent `editButtonItem()`, the view controller creates a `UIBarButtonItem` with the title Edit. Even better, this button comes with a target-action pair: it calls the method `setEditing(_:animated:)` to its `UIViewController` when tapped.

Now that Homepwner has a fully functional navigation bar, you can get rid of the header view and the associated code. In `ItemsViewController.swift`, delete the following methods.

```
override func viewDidLoad() {
super.viewDidLoad()

let nib = UINib(nibName: "ItemCell", bundle: nil)
tableView.registerNib(nib,
    forCellReuseIdentifier: "ItemCell")

// Load in the header view
// NSBundle.mainBundle().loadNibNamed("HeaderView", owner: self, options: nil)

// Set the table view's header view
// tableView.tableHeaderView = headerView
}
```

Also remove the the `toggleEditMode(_:)` method.

```
@IBAction func toggleEditMode(sender: AnyObject) {
    if editing {
        // Change text of button to inform user of state
        sender.setTitle("Edit", forState: .Normal)

        // Turn off editing mode
        setEditing(false, animated: true)
    } else {
        // Change text of button to inform user of state
        sender.setTitle("Done", forState: .Normal)

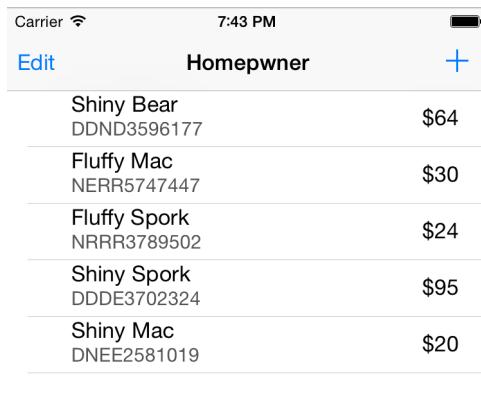
        // Enter editing mode
        setEditing(true, animated: true)
    }
}
```

You can also delete the declaration of the `headerView` property.

Finally, you can also remove the file `HeaderView.xib` from the project navigator.

Build and run again. The old Edit and New buttons are gone, leaving you with a lovely **UINavigationBar** (Figure 7.17).

Figure 7.17 Homeowner with navigation bar



Bronze Challenge: Displaying a Number Pad

The keyboard for the `UITextField` that displays a `Item`'s `valueInDollars` is a QWERTY keyboard. It would be better if it was a number pad. Change the Keyboard Type of that `UITextField` to the Number Pad. (Hint: you can do this in the XIB file using the attributes inspector.)

Silver Challenge: Dismissing a Number Pad

After completing the bronze challenge, you may notice that there is no return key on the number pad. Devise a way for the user to dismiss the number pad from the screen.

Gold Challenge: Pushing More View Controllers

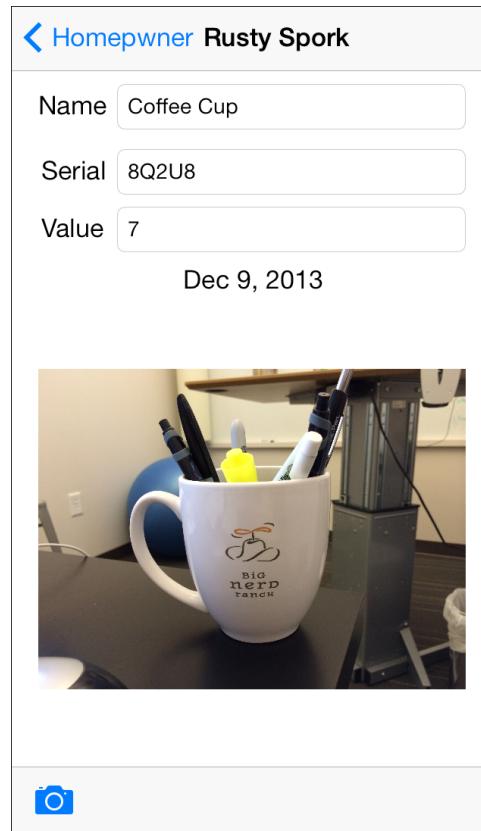
Right now, instances of `Item` cannot have their `dateCreated` property changed. Change `Item` so that they can, and then add a button underneath the `dateLabel` in `DetailViewController` with the title `Change Date`. When this button is tapped, push another view controller instance onto the navigation stack. This view controller should have a `UIDatePicker` instance that modifies the `dateCreated` property of the selected `Item`.

8

Camera

In this chapter, you are going to add photos to the Homepwner application. You will present a **UIImagePickerController** so that the user can take and save a picture of each item. The image will then be associated with an **Item** instance and viewable in the item's detail view.

Figure 8.1 Homepwner with camera addition

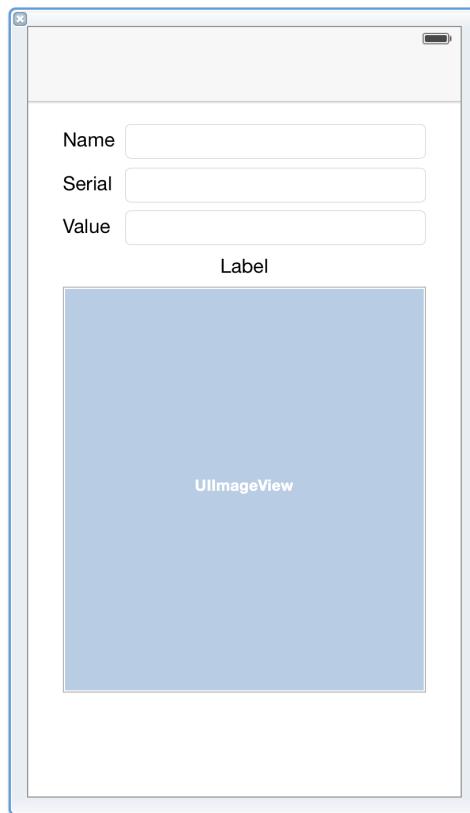


Images tend to be very large, so it is a good idea to store images separately from other data. Thus, in this chapter, you are going to create a second store for images. **ImageStore** will fetch and cache images as they are needed. It will also be able to flush the cache when memory runs low.

Displaying Images and **UIImageView**

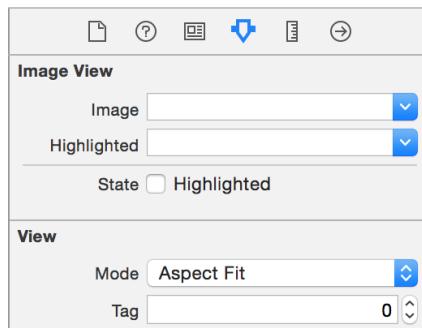
Your first step is to have the **DetailViewController** get and display an image. An easy way to display an image is to put an instance of **UIImageView** on the screen.

Open `Homepwner.xcodeproj` and `DetailViewController.xib`. Then drag an instance of **UIImageView** onto the view and position it below the label. Resize the image view to be almost as wide as the screen but leave some space at the bottom for an eventual toolbar (Figure 8.2).

Figure 8.2 **UIImageView** on **DetailViewController**'s view

A **UIImageView** displays an image according to its `contentMode` property. This property determines where to position and how to resize the content within the image view's frame. **UIImageView**'s default value for `contentMode` is `UIViewContentMode.ScaleToFill`, which will adjust the image to exactly match the bounds of the image view. If you keep the default, an image taken by the camera will be contorted to fit into the square **UIImageView**. You have to change the `contentMode` of the image view so that it resizes the image with the same aspect ratio.

Select the **UIImageView** and open the attributes inspector. Find the `Mode` attribute and change it to `Aspect Fit` (Figure 8.3). This will resize the image to fit within the bounds of the **UIImageView**.

Figure 8.3 Change **UIImageView**'s mode to `Aspect Fit`

Next, Option-click `DetailViewController.swift` in the project navigator to open it in the assistant editor. Control-drag from the **UIImageView** to the top of `DetailViewController.swift`. Name the outlet `imageView` and choose `Weak` as the storage type. Click Connect.

The top of `DetailViewController.swift` should now look like this:

```
class DetailViewController: UIViewController {
    @IBOutlet weak var nameField: UITextField!
    @IBOutlet weak var serialNumberField: UITextField!
    @IBOutlet weak var valueField: UITextField!
    @IBOutlet weak var dateLabel: UILabel!
    @IBOutlet weak var imageView: UIImageView!
```

Adding a camera button

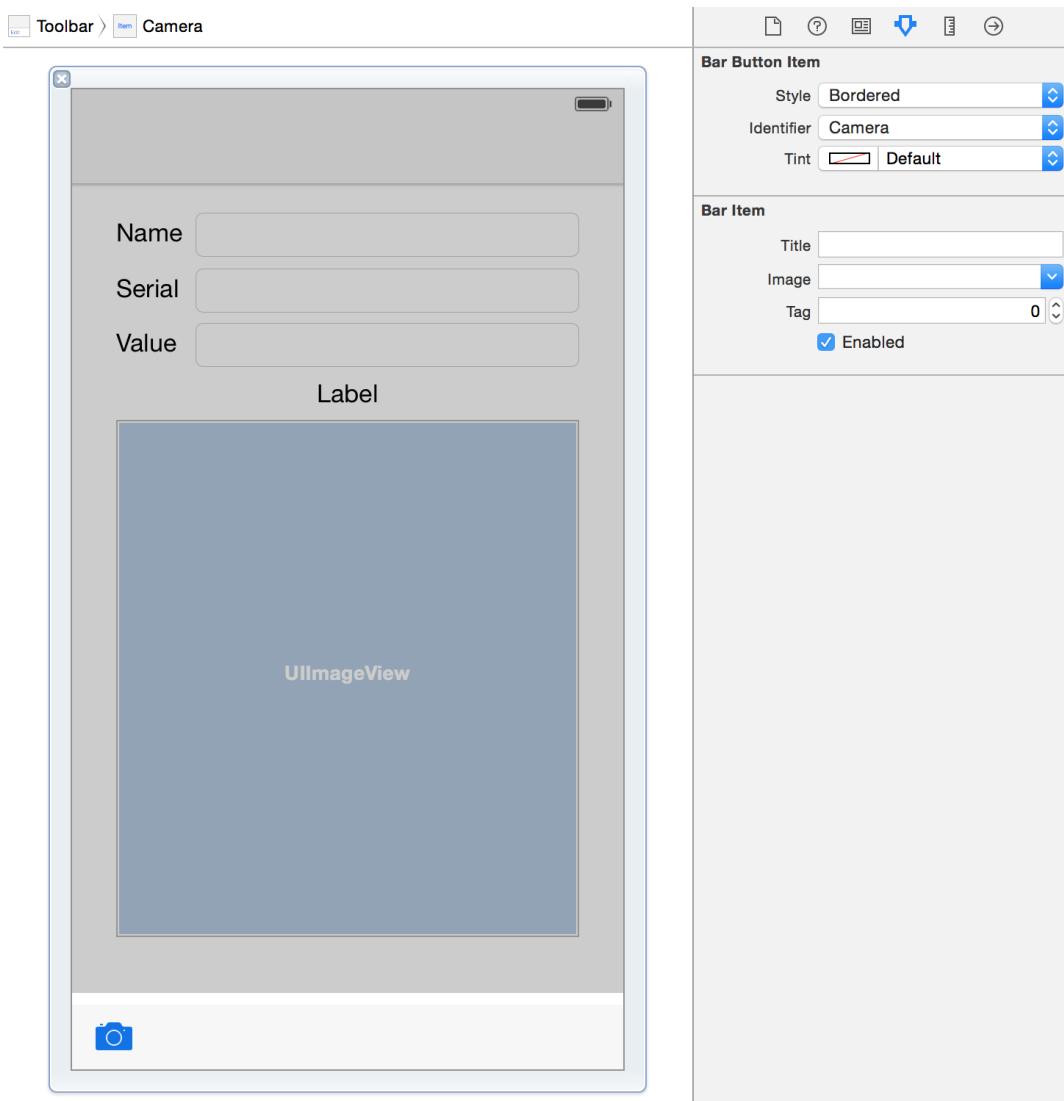
Now you need a button to initiate the photo-taking process. It would be nice to put this button on the navigation bar, but you will need the navigation bar for another button later. Instead, you will create an instance of **UIToolbar** and place it at the bottom of **DetailViewController**'s view.

In **DetailViewController.xib**, drag a **UIToolbar** from the object library onto the bottom of the view.

A **UIToolbar** works a lot like a **UINavigationBar** – you can add instances of **UIBarButtonItem** to it. However, where a navigation bar has two slots for bar button items, a toolbar has an array of bar button items. You can place as many bar button items in a toolbar as can fit on the screen.

By default, a new instance of **UIToolbar** that is created in a XIB file comes with one **UIBarButtonItem**. Select this bar button item and open the attribute inspector. Change the Identifier to Camera, and the item will show a camera icon (Figure 8.4).

Figure 8.4 **UIToolbar** with bar button item

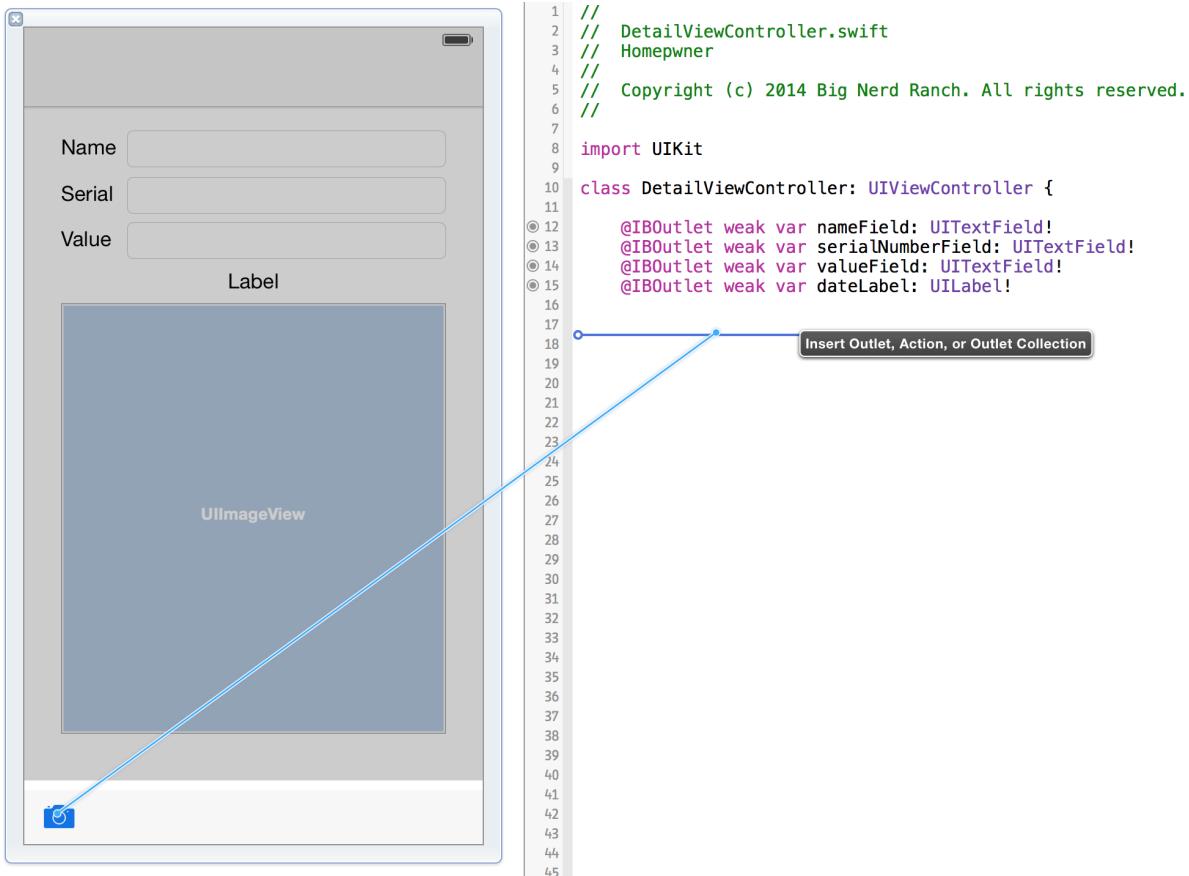


The camera button needs a target and an action. In previous exercises, you connected an action method in two steps: declaring it in code and then making the connection in the XIB file. Just like with outlets, there is a way to do both steps at once.

In the project navigator, Option-click `DetailViewController.swift` to open it in the assistant editor.

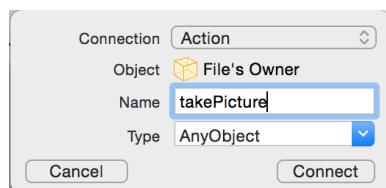
In `DetailViewController.xib`, select the camera button by first clicking on the toolbar and then the button itself. Then Control-drag from the selected button to `DetailViewController.swift` (Figure 8.5).

Figure 8.5 Creating and connecting an action method from a XIB



Let go of the mouse, and a window will appear that allows you to specify the type of connection you are creating. From the Connection pop-up menu, choose Action. Then, name this method `takePicture` and click Connect (Figure 8.6).

Figure 8.6 Creating the action



Now the stub of the action method is in `DetailViewController.swift`, and the `UIBarButtonItem` instance in the XIB is hooked up to send this message to the `DetailViewController` when tapped. The stub should look like this:

```
@IBAction func takePicture(sender: AnyObject) {  
}
```

Xcode is smart enough to know when an action method is connected in the XIB file. In Figure 8.7, notice the little circle within a circle in the gutter area next to `takePicture(_:)`'s method. When this circle is filled in, this action method is connected in a XIB file; an empty circle means that it still needs connecting.

Figure 8.7 Source file connection status

```

17
18 @IBAction func takePicture(sender: AnyObject) {
19
20 }
21

```

In a later chapter, you will need a pointer to the `UIToolbar` itself. Let's set that up now. Select the toolbar (not the Camera button on the toolbar). Then, Control-drag into `DetailViewController.swift`. Name this outlet `toolbar` and ensure that its storage is `Weak`.

The interface for `DetailViewController` now has a `toolbar` outlet:

```

class DetailViewController: UIViewController {

    @IBOutlet weak var nameField: UITextField!
    @IBOutlet weak var serialNumberField: UITextField!
    @IBOutlet weak var valueField: UITextField!
    @IBOutlet weak var dateLabel: UILabel!
    @IBOutlet weak var imageView: UIImageView!
    @IBOutlet weak var toolbar: UIToolbar!
}

```

If you made any mistakes while making these connections, you will need to open `DetailViewController.xib` and disconnect any bad connections. (Look for yellow warning signs in the connections inspector.)

Taking Pictures and UIImagePickerController

In the `takePicture(_:)` method, you will instantiate a `UIImagePickerController` and present it on the screen. When creating an instance of `UIImagePickerController`, you must set its `sourceType` property and assign it a delegate.

Setting the image picker's sourceType

The `sourceType` constant that tells the image picker where to get images. It has three possible values:

`UIImagePickerControllerSourceType.Camera`

The user will take a new picture.

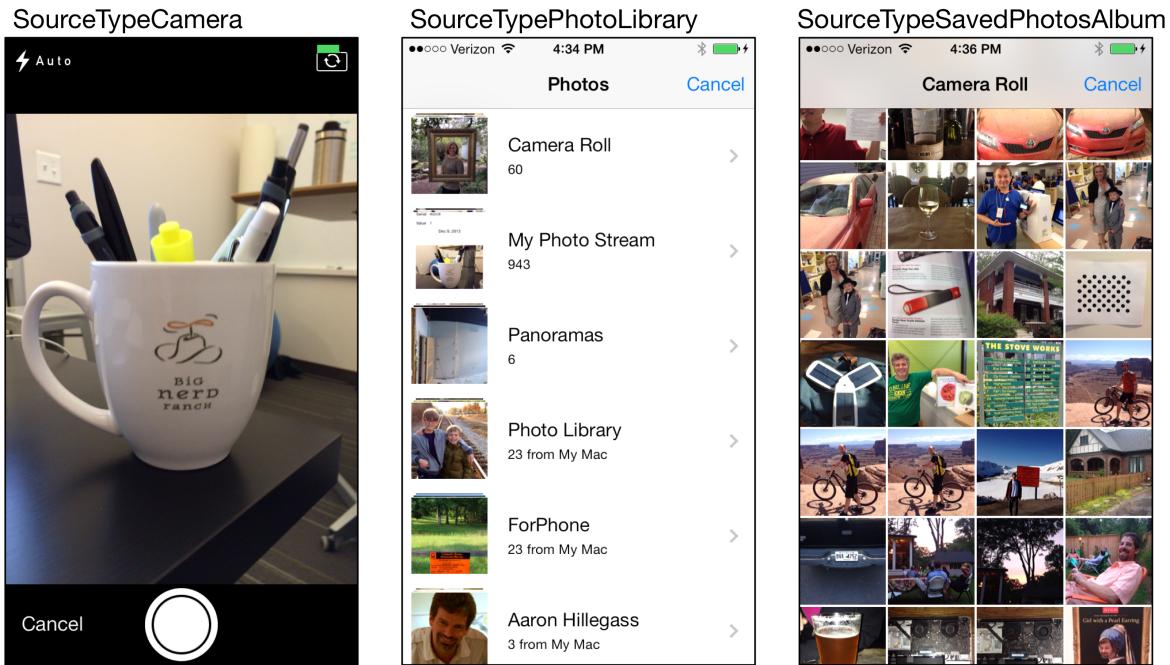
`UIImagePickerControllerSourceType.PhotoLibrary`

The user will be prompted to select an album and then a photo from that album.

`UIImagePickerControllerSourceType.SavedPhotosAlbum`

The user picks from the most recently taken photos.

Figure 8.8 Examples of three source types



The first source type, `.Camera`, will not work on a device that does not have a camera. So, before using this type, you have to check for a camera by calling the method `isSourceTypeAvailable(_:)` on the `UIImagePickerController` class:

```
class func isSourceTypeAvailable(sourceType: UIImagePickerControllerSourceType) -> Bool
```

Calling this method returns a boolean value for whether the device supports the passed-in source type.

In `DetailViewController.swift`, find the stub for `takePicture(_:)`. Add the following code to create the image picker and set its `sourceType`.

```
@IBAction func takePicture(sender: AnyObject) {
    let imagePicker = UIImagePickerController()

    // If the device has a camera, take a picture, otherwise,
    // just pick from photo library
    if UIImagePickerController.isSourceTypeAvailable(.Camera) {
        imagePicker.sourceType = .Camera
    } else {
        imagePicker.sourceType = .PhotoLibrary
    }
}
```

Setting the image picker's delegate

In addition to a source type, the `UIImagePickerController` instance needs a delegate. When the user selects an image from the `UIImagePickerController`'s interface, the delegate is sent the message `imagePickerController(_:didFinishPickingMediaWithInfo:)`. (If the user taps the cancel button, then the delegate receives the message `imagePickerControllerDidCancel(_:)`.)

The image picker's delegate will be the instance of `DetailViewController`. In `DetailViewController.swift`, declare that `DetailViewController` conforms to the `UINavigationControllerDelegate` and the `UIImagePickerControllerDelegate` protocols.

```
class DetailViewController: UIViewController,
    UINavigationControllerDelegate, UIImagePickerControllerDelegate {
```

Why `UINavigationControllerDelegate`? `UIImagePickerController`'s delegate property is actually inherited from its superclass, `UINavigationController`, and while `UIImagePickerController` has its own delegate protocol, its inherited delegate property is declared to point to an object that conforms to `UINavigationControllerDelegate`.

In `DetailViewController.swift`, add the following code to `takePicture(_ :)` to set the instance of `DetailViewController` to be the image picker's delegate.

```
@IBAction func takePicture(sender: AnyObject) {

    let imagePicker = UIImagePickerController()

    // If the device has a camera, take a picture, otherwise,
    // just pick from photo library
    if UIImagePickerController.isSourceTypeAvailable(.Camera) {
        imagePicker.sourceType = .Camera
    }
    else {
        imagePicker.sourceType = .PhotoLibrary
    }

    imagePicker.delegate = self
}
```

Presenting the image picker modally

Once the `UIImagePickerController` has a source type and a delegate, it is time to get its view on the screen. Unlike other `UIViewController` subclasses you have used, an instance of `UIImagePickerController` is presented *modally*. A *modal view controller* takes over the entire screen until it has finished its work.

To present a view controller modally, you send `presentViewController(_ :animated:completion:)` to the `UIViewController` whose view is on the screen. The view controller to be presented is passed to it, and this view controller's view slides up from the bottom of the screen. (You will learn more about the details of presenting modal view controllers in Chapter 13.)

In `DetailViewController.swift`, add code to the end of `takePicture(_ :)` to present the `UIImagePickerController`.

```
imagePicker.delegate = self

// Place image picker on the screen
presentViewController(imagePicker, animated: true, completion: nil)
}
```

(The third argument, `completion:`, expects a closure. You will learn about closures in Chapter 13.)

You can build and run the application now. Select a `Item` to see its details and then tap the camera button on the `UIToolbar`. `UIImagePickerController`'s interface will appear on the screen (Figure 8.9), and you can take a picture or choose an existing image if your device does not have a camera.

(If you are working on the simulator, you can open Safari in the simulator and navigate to a page with an image. Click and hold the image and then choose Save Image to save it in the simulator's photo library. Alternatively, you can drag an image from your computer on to the simulator. Then this image will be shown in the image picker.)

Figure 8.9 **UIImagePickerController** preview interface

Saving the image

Selecting an image dismisses the **UIImagePickerController** and returns you to the detail view. However, you do not have a reference to the photo once the image picker is dismissed. To fix this, you are going to implement the delegate method **imagePickerController(_:didFinishPickingMediaWithInfo:)**. This method is called on the image picker's delegate when a photo has been selected.

In **DetailViewController.swift**, implement this method to put the image into the **UIImageView** and then call the method to dismiss the image picker.

```
func imagePickerController(picker: UIImagePickerController,
    didFinishPickingMediaWithInfo info: [NSObject: AnyObject]) {
    // Get picked image from info dictionary
    let image = info[UIImagePickerControllerOriginalImage] as UIImage

    // Put that image onto the screen in our image view
    imageView.image = image

    // Take image picker off the screen -
    // you must call this dismiss method
    dismissViewControllerAnimated(true, completion: nil)
}
```

Build and run the application again. Take (or select) a photo. The image picker is dismissed, and you are returned to the **DetailViewController**'s view.

You could have hundreds of items, and each one could have a large image associated with it. Keeping hundreds of instances of **Item** in memory is not a big deal. Keeping hundreds of images in memory would be bad: First, you will get a low memory warning. Then, if your app's memory footprint continues to grow, the operating system will terminate it. The solution, which you are going to implement in the next section, is to store images to disk and only fetch them into RAM when they are needed. This fetching will be done by a new class, **ImageStore**. When the **ImageStore** receives a low-memory notification, it will flush its cache to free the memory that the fetched images were occupying.

Creating ImageStore

The image store will hold the pictures the user takes. In Chapter 9, you will have instances of **Item** write out their instance variables to a file, which will then be read in when the application starts. However, because images tend to be very large, it is a good idea to keep them separate from other data. The image store will fetch and cache the images as they are needed. It will also be able to flush the cache if the device runs low on memory.

Create a new **NSObject** subclass called **ImageStore**. Open **ImageStore.swift** and create its interface:

```
class ImageStore: NSObject {

    func setImage(image: UIImage, forKey key: String) {
    }

    func imageForKey(key: String) -> UIImage? {
        return nil
    }

    func deleteImageForKey(key: String) {
    }
}
```

Then, add a property to hang onto the images.

```
class ImageStore: NSObject {

    var imageDictionary = [String: UIImage]()

    func setImage(image: UIImage, forKey key: String) {
    }

    func imageForKey(key: String) -> UIImage? {
        return nil
    }

    func deleteImageForKey(key: String) {
    }
}
```

Finish the implementation of the three methods you stubbed out earlier.

```
func setImage(image: UIImage, forKey key: String) {
    imageDictionary[key] = image
}

func imageForKey(key: String) -> UIImage? {
    return nil
    return imageDictionary[key]
}

func deleteImageForKey(key: String) {
    imageDictionary.removeValue(forKey: key)
}
```

Dictionary

Dictionaries and arrays differ in how they store their objects. An array is an ordered list of pointers to objects that is accessed by an index. When you have an array, you can ask it for the object at the *n*th index:

```
// Put something at the beginning of an array
someArray.insert(something, atIndex: 0)

// Get that same object out
let something = someArray[0]
```

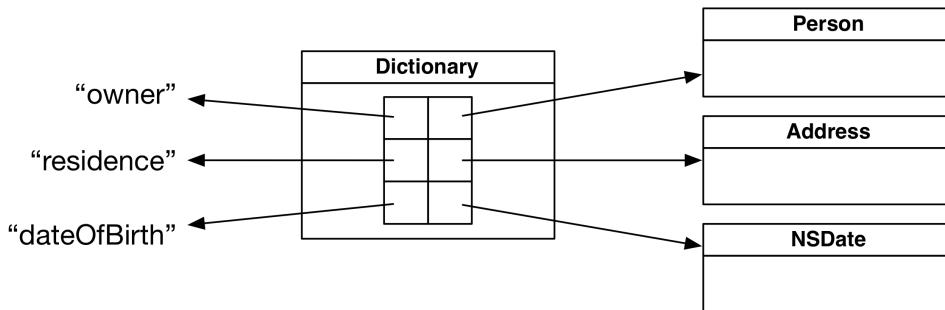
A dictionary's objects are not ordered within the collection. So instead of accessing entries with an index, you use a *key*. The key is usually an instance of **String**.

```
// Add something to a dictionary for the key "MyKey"
someDictionary["MyKey"] = something

// Get that same thing out
let something = someDictionary["MyKey"]
```

We call each entry in a dictionary a *key-value pair*. The *value* is the object being stored in the collection, and the *key* is a unique value (usually a string) that you use to store and retrieve the value later. (In other development environments, a dictionary is called a *hash map*, *hash table*, or *associative array*, but we still use the term key-value pair to talk about the information they store.)

Figure 8.10 **Dictionary** diagram



There are a lot of uses for a **Dictionary**. The two most common are flexible data structures and lookup tables.

First, let's talk about flexible data structures. Typically, when you want to represent a model object, you create a subclass of **NSObject** and give it appropriate instance variables. For example, a **Person** class would have properties like `firstName`, `age`, and other things that you expect a real-life person to have. An instance of **Dictionary** can also be used to represent a model object. In the person example, it would contain values for the keys `firstName`, `age`, and other things that you expect a real-life person to have.

The difference is that the **Person** class requires you to define exactly what a **Person** is and you cannot add, remove, or change the structural make-up of a person. With a **Dictionary**, if you wanted to add an address to the “Person”, you could simply add a value for the address key.

This is not an endorsement to use **Dictionary** to represent every object – most objects need to have a rigid definition, rules for the way they store, save, and load data, and behavior beyond just storing data. That usually means defining a custom class, like **Item**. However, **Dictionary** is commonly used to represent data that is passed into or returned from a method that can have a different structure depending on options you have specified. For example, the **UIImagePickerController**'s delegate method hands you an **Dictionary** that could contain an image or a video depending on how you configured the image picker. The dictionary could also contain metadata related to that image or video.

The other common usage of **Dictionary** is creating lookup tables. Sometime early in your programming career, you probably did something like this:

```
func changeCharacterClass(sender: AnyObject) {
    let enteredText = textField.text
    if enteredText == "Knight" {
        character.characterClass = .Knight
    }
    else if enteredText == "Mage" {
        character.characterClass = .Mage
    }
    else if enteredText == "Thief" {
        character.characterClass = .Thief
    }
}
```

A dictionary can solve the problem of creating giant if-else or switch statements by pre-determining the mapping between two objects. Continuing with perhaps the nerdiest example of all time, an **Dictionary** could be initialized like so:

```
var lookup = [String:CharacterClass]()
lookup["Knight"] = .Knight
lookup["Mage"] = .Mage
lookup["Thief"] = .Thief
```

and then you can change the **changeCharacterClass(_:)** method to something much cleaner:

```
func changeCharacterClass(sender: AnyObject) {
    character.characterClass = lookup[textField.text]
}
```

The added bonus with this approach is that you do not have to hard-code all the possibilities, but could store them in a data file, get them from a server somewhere, or dynamically add them given some input from the user. This is how the **ImageStore** will work: a key will be generated to map to an image and used to lookup that image later.

When using a dictionary, there can only be one value for each key. If you add a value to a dictionary with a key that matches the key of a value already present in the dictionary, the earlier value is removed. If you need to store multiple values under one key, you can put them in an array and add the array to the dictionary as the value.

Finally, note that a dictionary's memory management is like that of an array. Whenever you add an object to a dictionary, the dictionary owns it, and whenever you remove an object from a dictionary, the dictionary releases its ownership.

Another Dependency

The **DetailViewController** needs an instance of **ImageStore** to fetch and store images. You'll use a pass-the-baton approach to dependency injection, just as you did with the **ItemStore**.

In **DetailViewController.swift**, add a property for an **ImageStore**. Also, update the designated initializer to have an additional argument that is an instance of **ImageStore**.

```
let itemStore: ItemStore
let imageStore: ImageStore

init(item: Item, itemStore: ItemStore, imageStore: ImageStore) {
    self.item = item
    self.itemStore = itemStore
    self.imageStore = imageStore
    super.init(nibName: "DetailViewController", bundle: nil)
}
```

Now do the same in **ItemsViewController.swift**. Add a property to hold on to an instance of **ImageStore** and update the designated initializer.

```
class ItemsViewController: UITableViewController {

    let itemStore: ItemStore
    let imageStore: ImageStore

    init(itemStore: ItemStore, imageStore: ImageStore) {
        self.itemStore = itemStore
        self.imageStore = imageStore
        super.init(nibName: nil, bundle: nil)

        navigationItem.title = "Homeowner"

        let addItem = UIBarButtonItem(barButtonSystemItem: .Add,
                                      target: self,
                                      action: "addNewItem:")
        navigationItem.rightBarButtonItem = addItem

        navigationItem.leftBarButtonItem = editButtonItem()
    }
}
```

Then, update `tableView(_:tableView:didSelectRowAt IndexPath:)` to use the new designated initializer for `DetailViewController`.

```
override func tableView(tableView: UITableView,
    didSelectRowAtIndexPath indexPath: NSIndexPath) {
    let item = itemStore.allItems[indexPath.row]
    let dvc = DetailViewController(item: item, itemStore: itemStore, imageStore: imageStore)
```

Finally, update `AppDelegate` to create and inject the `ImageStore`.

```
func application(application: UIApplication!,
    didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
    // Override point for customization after application launch.
    window = UIWindow(frame: UIScreen.mainScreen().bounds)

    // Create an ItemStore
    let itemStore = ItemStore()

    // Create an ImageStore
    let imageStore = ImageStore()

    // Create an ItemsViewController and inject ItemStore
    // as well as ImageStore
    let ivc = ItemsViewController(itemStore: store, imageStore: imageStore)
```

Creating and Using Keys

When an image is added to the store, it will be put into a dictionary under a unique key, and the associated `Item` object will be given that key. When the `DetailViewController` wants an image from the store, it will ask its `item` for the key and search the dictionary for the image. Add a property to `Item.swift` to store the key.

```
let dateCreated: NSDate
let itemKey: String
```

The image keys need to be unique in order for your dictionary to work. While there are many ways to hack together a unique string, you are going to use the Cocoa Touch mechanism for creating universally unique identifiers (UUIDs), also known as globally unique identifiers (GUIDs). Objects of type `NSUUID` represent a UUID and are generated using the time, a counter, and a hardware identifier, which is usually the MAC address of the WiFi card. When represented as a string, UUIDs look something like this:

4A73B5D2-A6F4-4B40-9F82-EA1E34C1DC04

In `Item.swift`, modify the designated initializer to generate a UUID and set it as the `itemKey`.

```
init(name: String, valueInDollars: Int?, serialNumber: String?) {
    // Give the properties initial values
    self.name = name
    self.valueInDollars = valueInDollars
    self.serialNumber = serialNumber
    dateCreated = NSDate()

    // Create an NSUUID object - and get its string representation
    let uuid = NSUUID()
    itemKey = uuid.UUIDString

    // Call the superclass's designated initializer
    super.init()
}
```

Then, in `DetailViewController.swift`, update `imagePickerController(_:didFinishPickingMediaWithInfo:)` to store the image in the `ImageStore`.

```

func imagePickerController(picker: UIImagePickerController,
    didFinishPickingMediaWithInfo info: [NSObject : AnyObject]) {
    // Get picked image from info dictionary
    let image = info[UIImagePickerControllerOriginalImage] as UIImage

    // Store the image in the ImageStore for the item's key
    imageStore.setImage(image, forKey:item.itemKey)

    // Put that image onto the screen in our image view
    imageView.image = image

    // Take image picker off the screen -
    // you must call this dismiss method
    dismissViewControllerAnimated(true, completion: nil)
}

```

Each time an image is captured, it will be added to the store. Both the **ImageStore** and the **Item** will know the key for the image, so both will be able to access it as needed.

Similarly, when an item is deleted, you need to delete its image from the image store. In `ItemsViewController.swift`, update `tableView(_:commitEditingStyle:forRowAtIndexPath:)` to remove the item's image from the image store.

```

override func tableView(tableView: UITableView, commitEditingStyle editingStyle: UITableViewCellEditingStyle, forRowAtIndexPath indexPath: NSIndexPath) {
    // If the table view is asking to commit a delete command...
    if editingStyle == .Delete {
        let item = itemStore.allItems[indexPath.row]

        // Remove the item's image from the image store
        imageStore.deleteImageForKey(item.itemKey)

        // Remove the item from the store
        itemStore.removeItem(item)

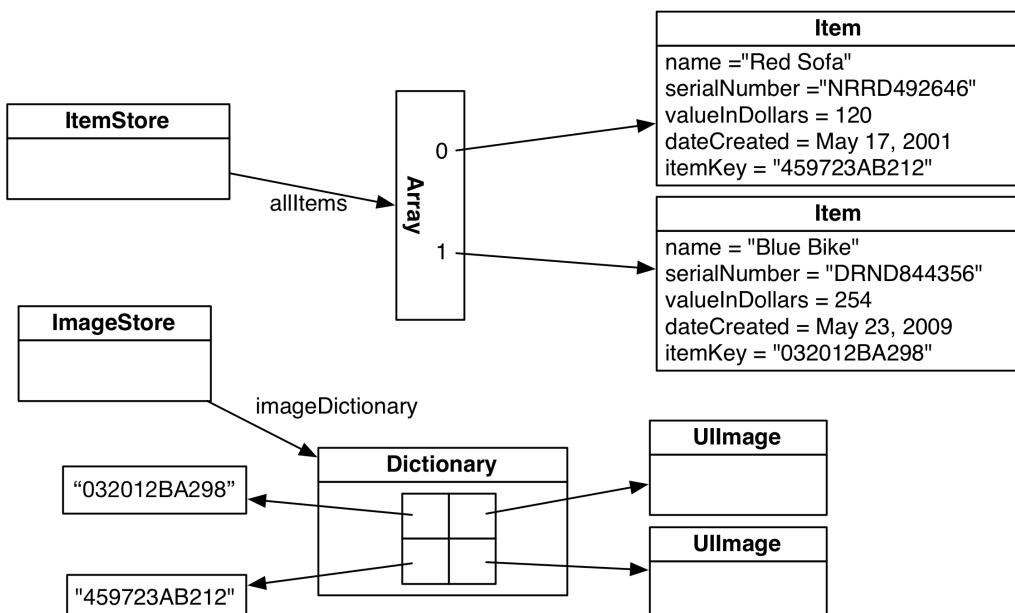
        // Also remove that row from the table view with an animation
        tableView.deleteRowsAtIndexPaths([indexPath], withRowAnimation: .Fade)
    }
}

```

Wrapping up ImageStore

Now that the **ImageStore** can store images and instances of **Item** have a key to get that image (Figure 8.11), you need to teach **DetailViewController** how to grab the image for the selected **Item** and place it in its `imageView`.

Figure 8.11 Cache



The **DetailViewController**'s view will appear at two times: when the user taps a row in **ItemsViewController** and when the **UIImagePickerController** is dismissed. In both of these situations, the **imageView** should be populated with the image of the **Item** being displayed.

In **DetailViewController.swift**, add code to **viewWillAppear(_:)** to do this.

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)

    nameField.text = item?.name
    serialNumberField.text = item?.serialNumber
    valueField.text = "\(item?.valueInDollars)"

    let date = item.dateCreated

    let dateFormatter = NSDateFormatter()
    dateFormatter.dateStyle = .MediumStyle
    dateFormatter.timeStyle = .NoStyle

    dateLabel.text = dateFormatter.stringFromDate(date)

    // Get the item key
    let key = item.itemKey

    // If there is an associated image with the item ...
    if let imageToDisplay = imageStore.imageForKey(key) {
        // ... display it on the image view
        imageView.image = imageToDisplay
    }
}
```

If there is no image associated with the item, then **imageForKey(_:)** will return **nil**. When the image is **nil**, the **UIImageView** will not display an image.

Build and run the application. Create a **Item** and select it from the **UITableView**. Then, tap the camera button and take a picture. The image will appear as it should.

Dismissing the Keyboard

When the keyboard appears on the screen in the item detail view, it obscures **DetailViewController**'s **imageView**. This is annoying when you are trying to see an image, so you are going to implement the delegate method **textFieldShouldReturn(_:)** to have the text field resign its first responder status to dismiss the keyboard when the return key is tapped. (This is why you hooked up the delegate outlets earlier.) But first, in **DetailViewController.swift**, have **DetailViewController** conform to the **UITextFieldDelegate** protocol.

```
class DetailViewController: UIViewController,
    UINavigationControllerDelegate, UIImagePickerControllerDelegate,
    UITextFieldDelegate {
```

Then, implement **textFieldShouldReturn(_:)**.

```
func textFieldShouldReturn(textField: UITextField) -> Bool {
    textField.resignFirstResponder()
    return true
}
```

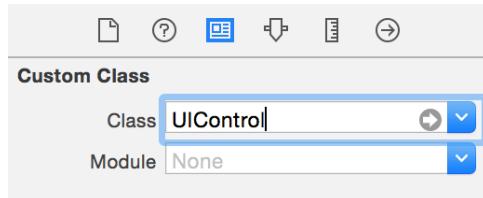
It would be stylish to also dismiss the keyboard if the user taps anywhere else on **DetailViewController**'s view. You can dismiss the keyboard by sending the view the message **endEditing(_:)**, which will cause the text field (as a subview of the view) to resign as first responder. Now let's figure out how to get the view to send a message when tapped.

You have seen how classes like **UIButton** can send an action message to a target when tapped. Buttons inherit this target-action behavior from their superclass, **UIControl**. You are going to change the view of

DetailViewController from an instance of **UIView** to an instance of **UIControl** so that it can handle touch events.

In **DetailViewController.xib**, select the main View object. Open the identity inspector and change the view's class to **UIControl** (Figure 8.12).

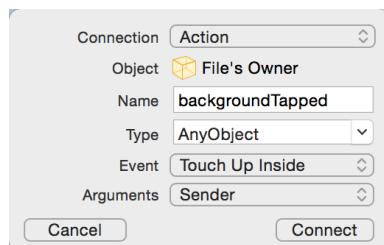
Figure 8.12 Changing the class of **DetailViewController**'s view



Then, open **DetailViewController.swift** in the assistant editor. Control-drag from the view (now a **UIControl**) to the implementation of **DetailViewController**. When the pop-up window appears, select Action from the Connection pop-up menu. Notice that the interface of this pop-up window is slightly different than the one you saw when creating and connecting the **UIBarButtonItem**. A **UIBarButtonItem** is a simplified version of **UIControl** – it only sends its target an action message when it is tapped. A **UIControl**, on the other hand, can send action messages in response to a variety of events.

Therefore, you must choose the appropriate event type to trigger the action message being sent. In this case, you want the action message to be sent when the user taps on the view. Configure this pop-up window to appear exactly as it does in Figure 8.13 and click Connect.

Figure 8.13 Configuring a **UIControl** action



This will create a stub method in **DetailViewController.swift**. Update that method:

```
@IBAction func backgroundTapped(sender: AnyObject) {
    view.endEditing(true)
}
```

Build and run your application and test both ways of dismissing the keyboard.

Bronze Challenge: Editing an Image

UIImagePickerController has a built-in interface for editing an image once it has been selected. Allow the user to edit the image and use the edited image instead of the original image in **DetailViewController**.

Silver Challenge: Removing an Image

Add a button that clears the image for an item.

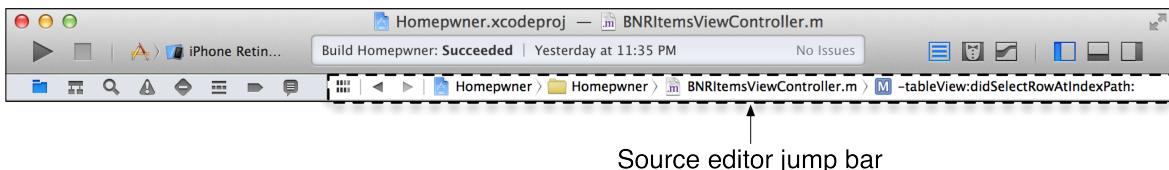
Gold Challenge: Camera Overlay

A **UIImagePickerController** has a **cameraOverlayView** property. Make it so that presenting the **UIImagePickerController** shows a crosshair in the middle of the image capture area.

For the More Curious: Navigating Implementation Files

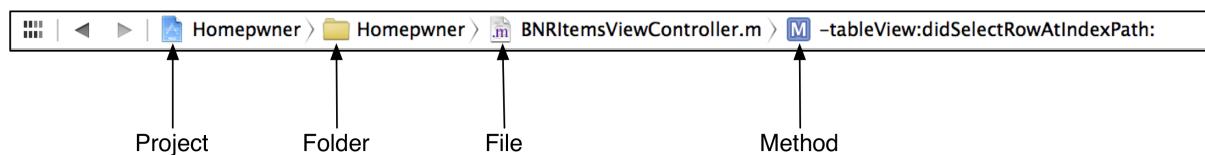
Both of your view controllers have quite a few methods in their implementation files. To be effective iOS developers, you must be able to go to the code you are looking for quickly and easily. The source editor jump bar in Xcode is one tool at your disposal to help out with this (Figure 8.14).

Figure 8.14 Source editor jump bar



The jump bar shows you where exactly you are within the project (and also where the cursor is within a given file). Figure 8.15 breaks down the jump bar details.

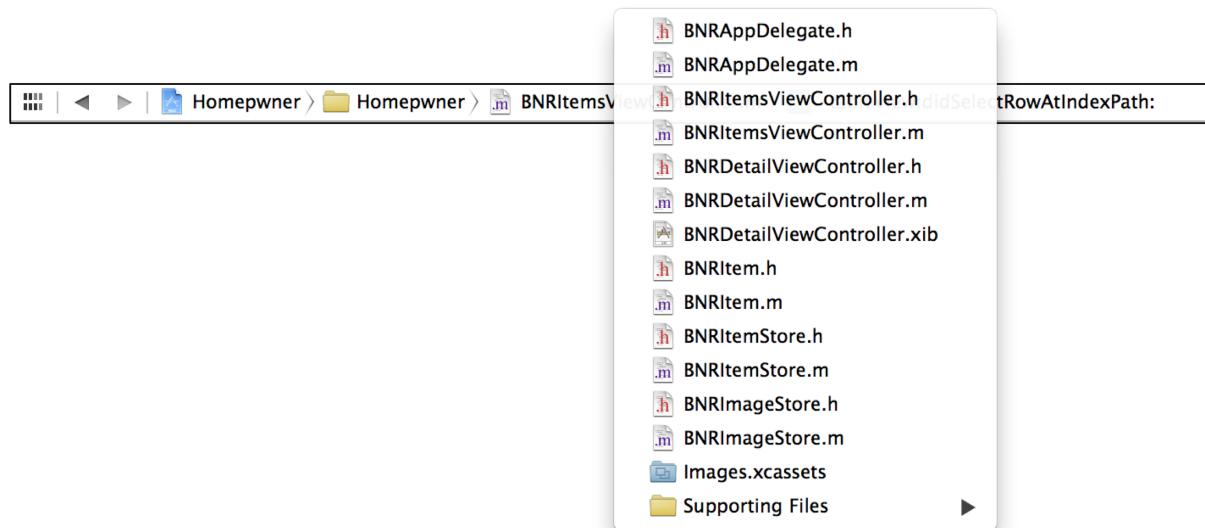
Figure 8.15 Jump bar details



The breadcrumb trail navigation of the jump bar mirrors the project navigation hierarchy. If you click on any of the sections, you will be presented with a popover of that section in the project hierarchy, and from there you can easily navigate to other parts of the project.

Figure 8.16 shows off the file popover in the Homepwner application.

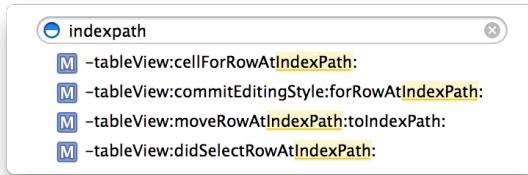
Figure 8.16 File popover



Perhaps most useful is the ability to navigate easily within an implementation file. If you click on the last element in the breadcrumb trail, you will get a popover with the contents of the file including all of the methods implemented within that file.

While the popover is still visible, you can start typing to filter the items in the list. At any point, you can then use the up and down arrow keys and then press the Enter key to jump to that method in the code. Figure 8.17 shows what you get when you search for “indexPath” in `ItemsViewController.swift`.

Figure 8.17 File popover with “indexpath” search



// MARK:

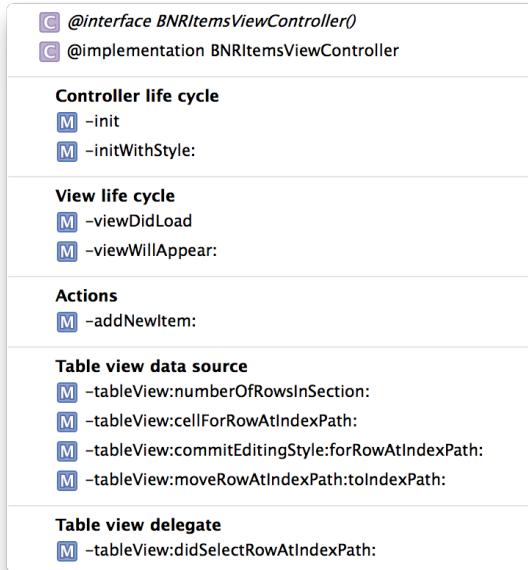
As your classes get longer, it can get more difficult to find the method you are looking for when it is buried in a long list of methods. A good way to organize your methods to help with the mess is by using the // MARK: comment.

```
// MARK: - View life cycle
override func viewDidLoad() { ... }
override func viewWillAppear(animated: Bool) { ... }

// MARK: - Actions
func addNewItem(sender: AnyObject) {...}
```

Adding // MARK:s to your code does not change anything with the code, but instead helps Xcode understand how you would like to visually organize your methods. You can see the results of adding them by opening the current file item in the jump bar. Figure 8.18 shows the results of a well-organized `ItemsViewController.swift`.

Figure 8.18 File popover with // MARK:s



Two useful // MARK:s are the divider and the label.

```
// This is a divider
// MARK: -

// This is a label
// MARK: My Awesome Methods

// They can be combined as well
// MARK: - My Awesome Methods
```

By using the `// MARK:` comment, you force yourself to organize your code. If done well, this will make your code more readable and easier for you to work with when you inevitably need to revisit the code. After doing it repeatedly, you will build habits which will further help you navigate your code base.

Saving, Loading, and Application States

There are many ways to save and load data in an iOS application. This chapter will take you through some of the most common mechanisms as well as the concepts you need for writing to or reading from the filesystem on iOS.

Archiving

Most iOS applications are really doing one thing: providing an interface for a user to manipulate data. Every object in an application has a role in this process. Model objects, as you know, are responsible for holding on to the data that the user manipulates. View objects simply reflect that data, and controllers are responsible for keeping the views and the model objects in sync. Therefore, when talking about saving and loading data, we are almost always talking about saving and loading model objects.

In `Homepwner`, the model objects that a user manipulates are instances of `Item`. `Homepwner` would actually be a useful application if instances of `Item` persisted between runs of the application, and in this chapter, you will use *archiving* to save and load `Item` objects.

Archiving is one of the most common ways of persisting model objects on iOS. *Archiving* an object involves recording all of its properties and saving them to the filesystem. *Unarchiving* recreates the object from that data.

Classes whose instances need to be archived and unarchived must conform to the `NSCoding` protocol and implement its two required methods, `encodeWithCoder(_:)` and `init(coder:)`.

```
protocol NSCoding {
    func encodeWithCoder(aCoder: NSCoder)
    init(coder aDecoder: NSCoder)
}
```

Make `Item` conform to `NSCoding`. Open `Homepwner.xcodeproj` and add this protocol declaration in `Item.swift`.

```
class Item: NSObject, NSCoding {
```

Now you need to implement the required methods. Let's start with `encodeWithCoder(_:)`. When an `Item` is sent the message `encodeWithCoder(_:)`, it will encode all of its properties into the `NSCoder` object that is passed as an argument. While saving, you will use `NSCoder` to write out a stream of data. That stream will be stored on the filesystem. This stream is organized as key-value pairs.

In `Item.swift`, implement `encodeWithCoder(_:)` to add the names and values of the item's properties to the stream.

```
func encodeWithCoder(aCoder: NSCoder) {
    aCoder.encodeObject(name, forKey: "name")
    aCoder.encodeObject(dateCreated, forKey: "dateCreated")
    aCoder.encodeObject(itemKey, forKey: "itemKey")

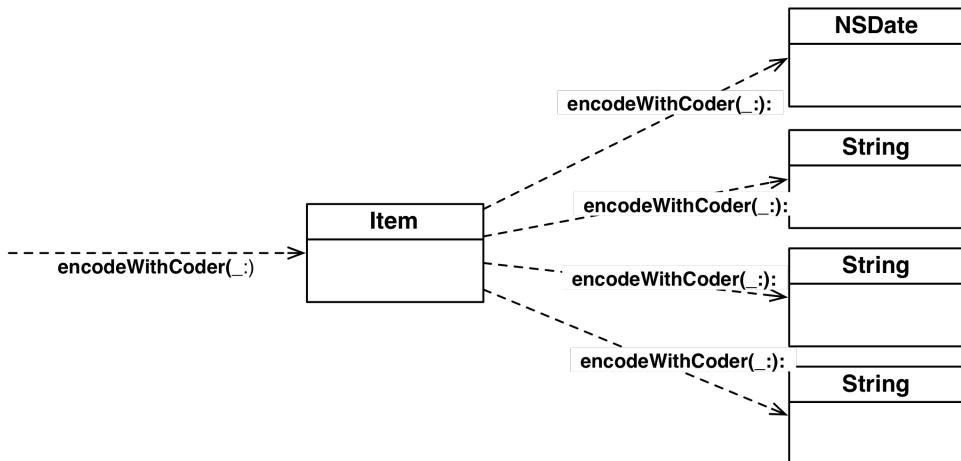
    if let serialNumber = serialNumber {
        aCoder.encodeObject(serialNumber, forKey: "serialNumber")
    }

    if let valueInDollars = valueInDollars {
        aCoder.encodeInteger(valueInDollars, forKey: "valueInDollars")
    }
}
```

Notice that pointers to objects are encoded with `encodeObject(_:forKey:)`, but `valueInDollars` is encoded with `encodeInt(_:forKey:)`. Check the documentation for `NSCoder` to see all of the types you can encode. Regardless of the type of the encoded value, there is always a key, which is a string that identifies which instance variable is being encoded. By convention, this key is the name of the property being encoded.

When an object is encoded (that is, it is the first argument in `encodeObject(_:forKey:)`), that object is sent `encodeWithCoder(_:)`. During the execution of its `encodeWithCoder(_:)` method, it encodes its object instance variables using `encodeObject(_:forKey:)` (Figure 9.1). Thus, encoding an object is a recursive process where each object encodes its “objects”, and they encode their objects, and so on.

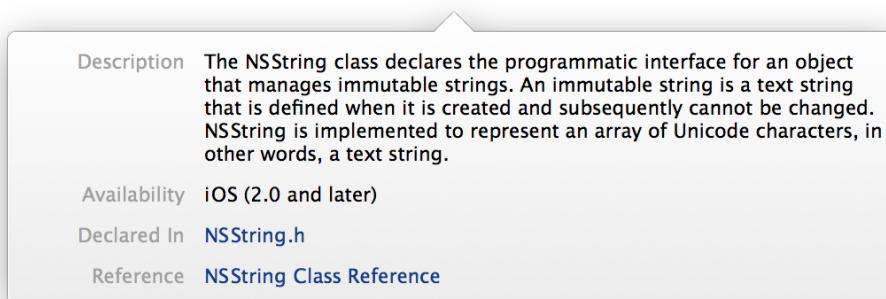
Figure 9.1 Encoding an object



To be encoded, these objects must also conform to `NSCoding`. Let’s confirm this for `String` and `NSDate`. The protocols that a class conforms to are listed in its class reference. Instead of opening up the documentation browser as you have done before, you can take a shortcut to get to the reference directly from your code.

In `Item.swift`, hold down the Option key, mouse over an occurrence of `String` in your code, and click. A pop-up window will appear with a brief description of the class and links to its header file and its reference.

Figure 9.2 Option-clicking `String`



Click the link to be taken to the `String` class reference, and at the top of the reference you will see a list of protocols that the class conforms to. `NSCoding` is not in this list, but if you click the `NSecureCoding` protocol, you will see that this protocol conforms to `NSCoding`. Thus, `String` does, as well.

You can do the same check for the `NSDate` class or take our word for it that `NSDate` is also `NSCoding` compliant.

Option-clicking is not just for classes. You can use the same shortcut for methods, types, protocols, and more. Keep this in mind as you run across items in your code that you want to know more about.

Now back to keys and encoding. The purpose of the key used when encoding is to retrieve the encoded value when this `Item` is loaded from the filesystem later. Objects being loaded from an archive are sent the message `init(coder:)`. This method should grab all of the objects that were encoded in `encodeWithCoder(_:)` and assign them to the appropriate instance variable.

In `Item.swift`, implement `init(coder:)`.

```
required init(coder aDecoder: NSCoder) {
    name = aDecoder.decodeObjectForKey("name") as String
    dateCreated = aDecoder.decodeObjectForKey("dateCreated") as NSDate
    itemKey = aDecoder.decodeObjectForKey("itemKey") as String
    serialNumber = aDecoder.decodeObjectForKey("serialNumber") as String?

    valueInDollars = aDecoder.decodeIntegerForKey("valueInDollars")

    super.init()
}
```

Notice that this method has an `NSCoder` argument, too. In `init(coder:)`, the `NSCoder` is full of data to be consumed by the `Item` being initialized. Also notice that you called `decodeObjectForKey:` to the container to get objects back and `decodeIntegerForKey(_:)` to get the `valueInDollars`.

In ???, we talked about the initializer chain and designated initializers. The `init(coder:)` method is not part of this design pattern; you will keep `Item`'s designated initializer the same, and `init(coder:)` will not call it.

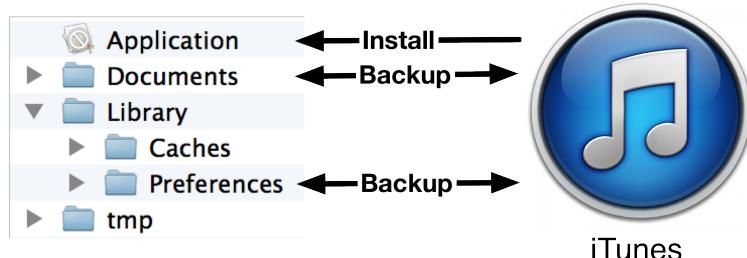
(By the way, archiving is how XIB files are created. `UIView` conforms to `NSCoding`. Instances of `UIView` are created when you drag them onto the canvas area. When the XIB file is saved, these views are archived into the XIB file. When your application launches, it unarchives the views from the XIB file. There are some minor differences between a XIB file and a standard archive, but overall it is the same process.)

Instances of `Item` are now `NSCoding` compliant and can be saved to and loaded from the filesystem using archiving. You can build the application to make sure there are no syntax errors, but you still need a way to kick off the saving and loading. You also need a place on the filesystem to store the saved items.

Application Sandbox

Every iOS application has its own *application sandbox*. An application sandbox is a directory on the filesystem that is barricaded from the rest of the filesystem. Your application must stay in its sandbox, and no other application can access your sandbox.

Figure 9.3 Application sandbox



The application sandbox contains a number of directories:

application bundle

This directory contains the executable and all application resources like NIB files and images. It is read-only.

Documents/

This directory is where you write data that the application generates during runtime and that you want to persist between runs of the application. It is backed up when the device is synchronized with iTunes or iCloud. If something goes wrong with the device, files in this directory can be restored from iTunes or iCloud. For example, in Homepwner, the file that holds the data for all your possessions will be stored here.

Library/Caches/

This directory is where you write data that the application generates during runtime and that you want to persist between runs of the application. However, unlike the `Documents` directory, it does not get backed up when the device is synchronized with iTunes or iCloud. A

major reason for not backing up cached data is that the data can be very large and extend the time it takes to synchronize your device. Data stored somewhere else – like a web server – can be placed in this directory. If the user needs to restore the device, this data can be downloaded from the web server again. If the device is very low on disk space, the system may delete the contents of this directory.

Library/Preferences/

This directory is where any preferences are stored and where the **Settings** application looks for application preferences. **Library/Preferences** is handled automatically by the class **NSUserDefaults** (which you will learn about in ???) and is backed up when the device is synchronized with iTunes or iCloud.

tmp/

This directory is where you write data that you will use temporarily during an application’s runtime. The operating system may purge files in this directory when your application is not running. However, to be tidy you should still explicitly remove files from this directory when you no longer need them. This directory does not get backed up when the device is synchronized with iTunes or iCloud. To get the path to the **tmp** directory in the application sandbox, you can use the convenience function **NSTemporaryDirectory**.

Constructing a file path

The instances of **Item** from **Homepwner** will be saved to a single file in the **Documents** directory. The **ItemStore** will handle writing to and reading from that file. To do this, the **ItemStore** needs to construct a path to this file.

Implement a new property in **ItemStore.swift** to store this path.

```
var allItems: [Item] = []
let itemArchivePath: String = {
    let documentsDirectories =
        NSSearchPathForDirectoriesInDomains(.DocumentDirectory, .UserDomainMask, true)
    let documentDirectory = documentsDirectories.first as String

    return documentDirectory.stringByAppendingPathComponent("items.archive")
}()
```

The function **NSSearchPathForDirectoriesInDomains** searches the filesystem for a path that meets the criteria given by the arguments. On iOS, the last two arguments are always the same. (This function is borrowed from OS X, where there are significantly more options.) The first argument is an **NSSearchPathDirectory** enumeration that specifies the directory in the sandbox you want the path to. For example, searching for **.CachesDirectory** will return the **Caches** directory in the application’s sandbox.

You can search the documentation for **NSSearchPathDirectory** to locate the other options. Remember that these enumeration values are shared by iOS and OS X, so not all of them will work on iOS.

The return value of **NSSearchPathForDirectoriesInDomains** is an array of strings. It is an array of strings because, on OS X, there may be multiple paths that meet the search criteria. On iOS, however, there will only be one (if the directory you searched for is an appropriate sandbox directory). Therefore, the name of the archive file is appended to the first and only path in the array. This will be where the archive of **Item** instances will live.

NSKeyedArchiver and NSKeyedUnarchiver

You now have a place to save data on the filesystem and a model object that can be saved to the filesystem. The final two questions are: how do you kick off the saving and loading processes and when do you do it? To save instances of **Item**, you will use the class **NSKeyedArchiver** when the application “exits.”

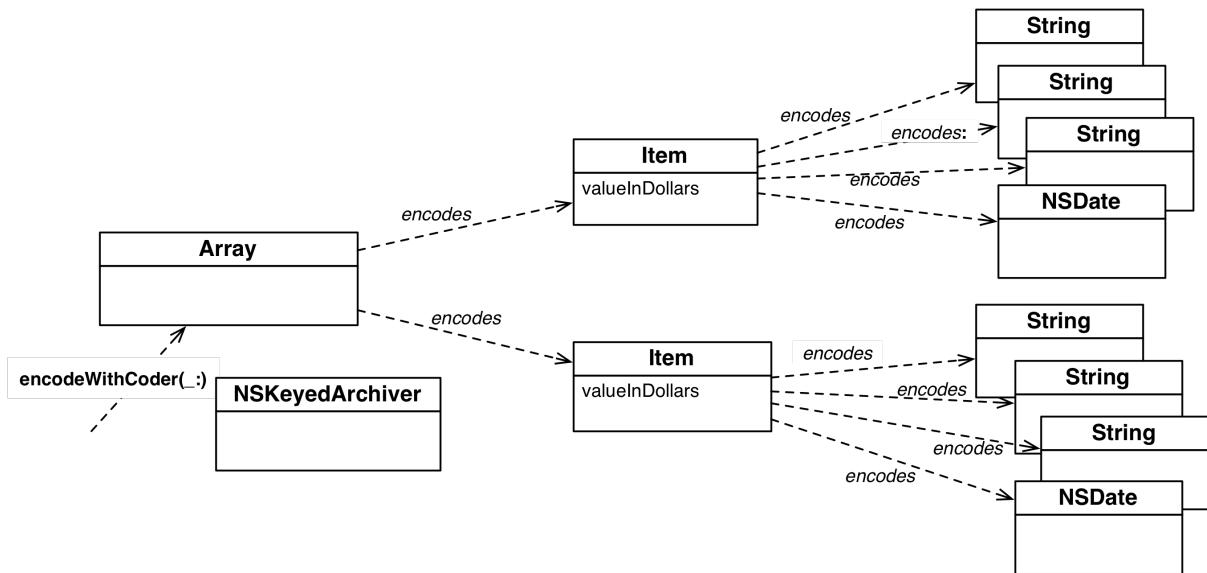
In **ItemStore.swift**, implement a new method that calls **archiveRootObject(_:toFile:)** on the **NSKeyedArchiver** class.

```
func saveChanges() -> Bool {
    return NSKeyedArchiver.archiveRootObject(allItems, toFile: itemArchivePath)
}
```

The `archiveRootObject(_:toFile:)` method takes care of saving every single **Item** in `allItems` to the `itemArchivePath`. Yes, it is that simple. Here is how `archiveRootObject(_:toFile:)` works:

- The method begins by creating an instance of **NSKeyedArchiver**. (**NSKeyedArchiver** is a concrete subclass of the abstract class **NSCoder**.)
- The method `encodeWithCoder:` is called on `allItems` and is passed the instance of **NSKeyedArchiver** as an argument.
- The `allItems` array then calls `encodeWithCoder(_:)` to all of the objects it contains, passing the same **NSKeyedArchiver**. Thus, all your instances of **Item** encode their instance variables into the very same **NSKeyedArchiver** (Figure 9.4).
- The **NSKeyedArchiver** writes the data it collected to the path.

Figure 9.4 Archiving the `allItems` array



NSNotificationCenter

When the user presses the Home button on the device, the application goes into the background. This is when you want to have the **ItemStore** save its changes.

In order for the **ItemStore** to know when the application is going into the background, you will use the notification center. Every application has an instance of **NSNotificationCenter**, which works like a smart bulletin board. An object can register as an observer (“Send me ‘lost dog’ notifications”). When another object posts a notification (“I lost my dog”), the notification center forwards the notification to the registered observers.

Whenever the application goes into the background, the application posts the `UIApplicationDidEnterBackgroundNotification` to the notification center. Objects that want to know when the application is going into the background can register for this notification.

In `ItemStore.swift`, override `init()` to register the item store as an observer of this notification.

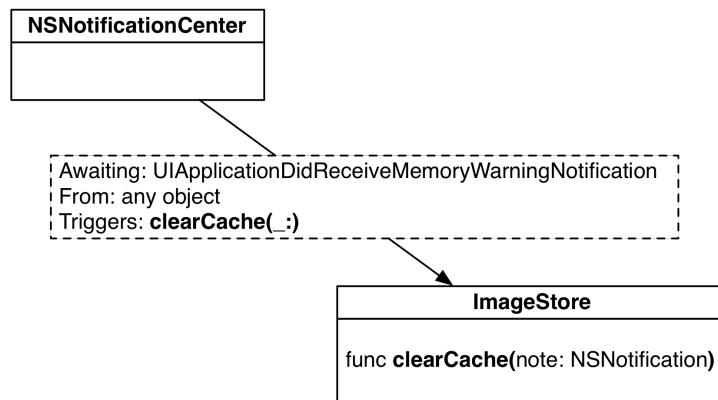
```

override init() {
    super.init()

    let nc = NotificationCenter.defaultCenter()
    nc.addObserver(self,
                   selector: "appDidEnterBackground:",
                   name: UIApplicationDidEnterBackgroundNotification,
                   object: nil)
}
  
```

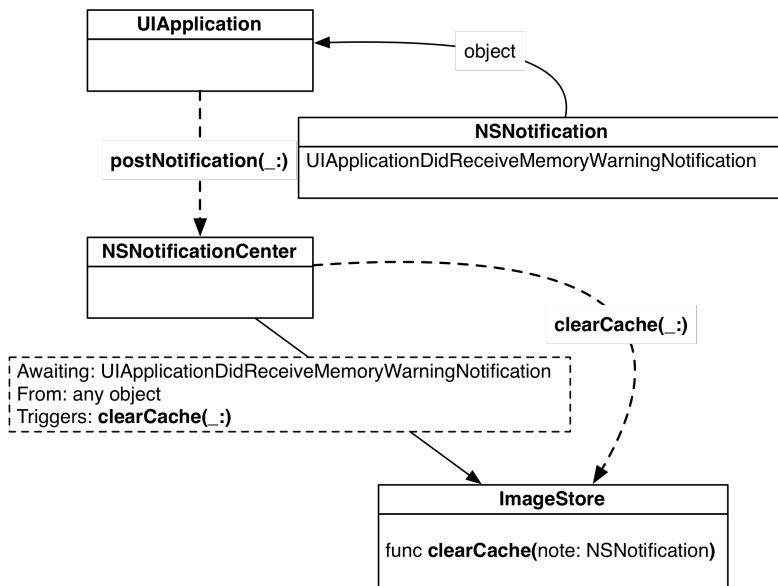
Now your item store is registered as an observer with the notification center (Figure 9.5).

Figure 9.5 Registered as an observer with notification center



Now, when the application goes into the background, the notification center will call the method `appDidEnterBackground(_ :)` on the `ItemStore` instance (Figure 9.6).

Figure 9.6 Receiving the notification



In `ItemStore.swift`, implement `appDidEnterBackground(_ :)` to save the changes to the store.

```

func appDidEnterBackground(note: NSNotification) {
    let success = saveChanges()
    if success {
        println("Saved all of the Items")
    } else {
        println("Could not save the Items")
    }
}
  
```

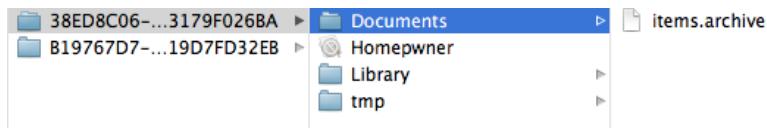
Build and run the application on the simulator. Create a few instances of `Item`. Then, press the Home button to leave the application. Check the console, and you should see a log statement indicating that the items were saved.

While you cannot yet load these instances of `Item` back into the application, you can still verify that *something* was saved. In Finder, press Command-Shift-G. Then, type in `~/Library/Application Support/iPhone Simulator` and press Enter. This is where all of the applications and their sandboxes are stored for the simulator.

Open the directory `8.0` (or, if you are working with another version of iOS, select that directory). Open Applications to see the list of every application that has run on your simulator using iOS 8.0. Unfortunately,

these applications have really unhelpful names. You have to dig into each directory to find the one that contains Homepwner.

Figure 9.7 Homepwner's sandbox



In Homepwner's directory, navigate into the Documents directory (Figure 9.7). You will see the `items.archive` file. Here is a tip: make an alias to the iPhone Simulator directory somewhere convenient to make it easy to check the sandboxes of your applications.

Now let's turn to loading these files. To load instances of `Item` when the application launches, you will use the class `NSKeyedUnarchiver` when the `ItemStore` is created.

In `ItemStore.swift`, add the following code to `init()`.

```
override init() {
    super.init()

    if let archivedItems =
        NSKeyedUnarchiver.unarchiveObjectWithFile(itemArchivePath) as? [Item] {
        allItems += archivedItems
    }

    let nc = NSNotificationCenter.defaultCenter()
    nc.addObserver(self,
                   selector: "appDidEnterBackground:",
                   name: UIApplicationDidEnterBackgroundNotification,
                   object: nil)
}
```

The `unarchiveObjectWithFile(_)` method will create an instance of `NSKeyedUnarchiver` and load the archive located at the `itemArchivePath` into that instance. The `NSKeyedUnarchiver` will then inspect the type of the root object in the archive and create an instance of that type. In this case, the type will be an `Array` because you created this archive with a root object of this type. (If the root object was an `Item` instead, `unarchiveObjectWithFile(_)` would return an instance of `Item`.)

The newly allocated `Array` is then sent `init(coder:)` and, as you may have guessed, the `NSKeyedUnarchiver` is passed as the argument. The array starts decoding its contents (instances of `Item`) from the `NSKeyedUnarchiver` and sends each of these objects the message `init(coder:)`, passing the same `NSKeyedUnarchiver`.

You can now build and run the application. Any items that a user enters will be available until the user explicitly deletes them. One thing to note about testing your saving and loading code: If you kill Homepwner from Xcode, the `UIApplicationDidEnterBackgroundNotification` will not get a chance to be sent and the item array will not be saved. You must press the Home button first and then kill it from Xcode by clicking the Stop button.

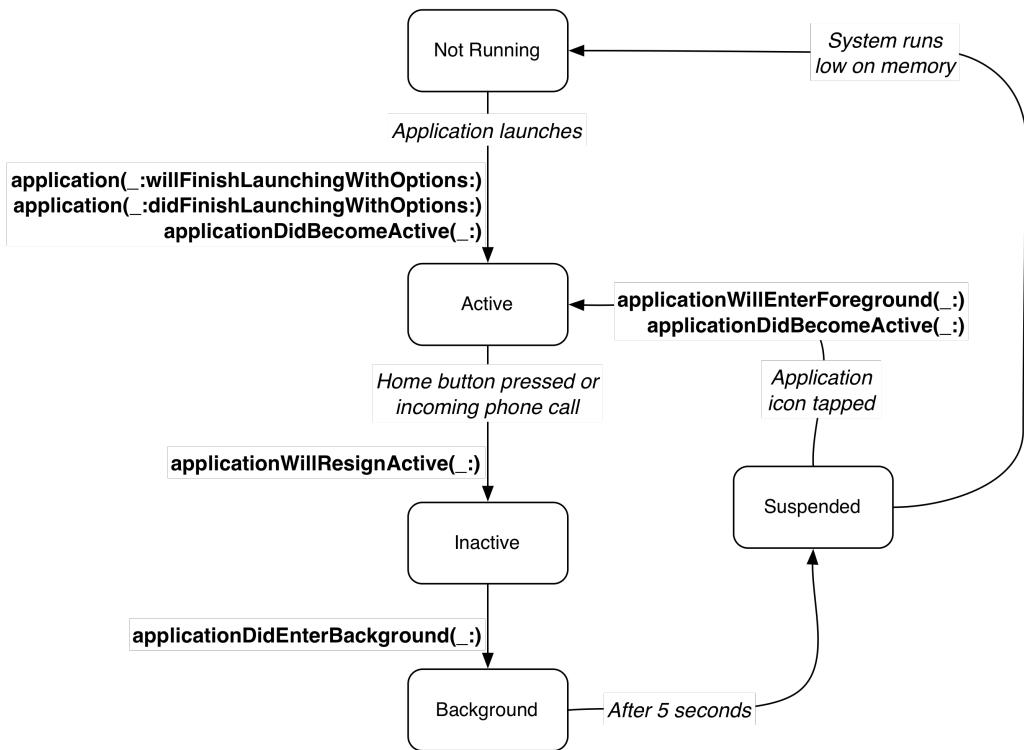
Now that you can save and load items, there is no reason to auto-populate each one with random data. In `ItemStore.swift`, modify the implementation of `createItem()` so that it creates an empty `Item` instead of one with random data.

```
func createItem() -> Item {
    let newItem = Item(random: true)
    let newItem = Item(random: false)
    allItems.append(newItem)
    return newItem
}
```

Application States and Transitions

In Homepwner, the items are archived when the application enters the *background state*. It is useful to understand all of the states an application can be in, what causes them to transition between states, and how your code can be notified of these transitions. This information is summarized in Figure 9.8.

Figure 9.8 States of typical application



When an application is not running, it is in the *not running* state, and it does not execute any code or have any memory reserved in RAM.

After the user launches an application, it enters the *active state*. When in the active state, an application's interface is on the screen, it is accepting events, and its code is handling those events.

While in the active state, an application can be temporarily interrupted by a system event like an SMS message, push notification, phone call, or alarm. An overlay will appear on top of your application to handle this event. This state is known as the *inactive state*. In the inactive state, an application is mostly visible (an alert view will appear and obscure part of the interface) and is executing code, but it is not receiving events. Applications typically spend very little time in the inactive state. You can force an active application into the inactive state by pressing the Lock button at the top of the device. The application will stay inactive until the device is unlocked.

When the user presses the Home button or switches to another application in some other way, the application enters the *background state*. (Actually, it spends a brief moment in the inactive state before transitioning to the background.) In the background, an application's interface is not visible or receiving events, but it can still execute code. By default, an application that enters the background state has about ten seconds before it enters the *suspended state*. Your application should not rely on this number; instead it should save user data and release any shared resources as quickly as possible.

An application in the suspended state cannot execute code, you cannot see its interface, and any resources it does not need while suspended are destroyed. A suspended application is essentially freeze-dried and can be quickly thawed when the user relaunches it. Table 9.1 summarizes the characteristics of the different application states.

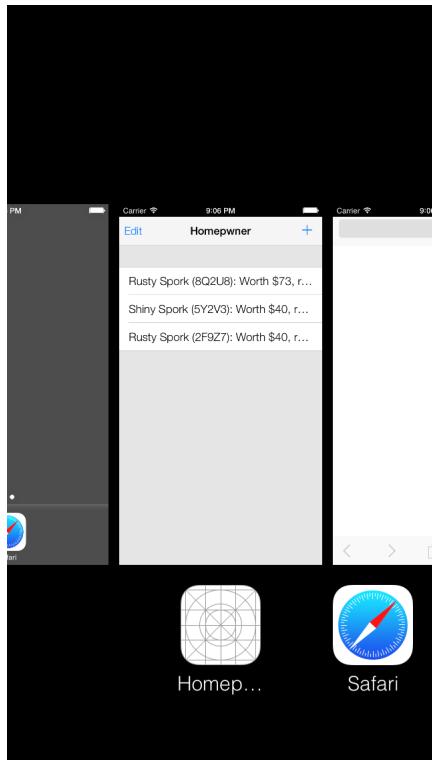
Table 9.1 Application states

State	Visible	Receives Events	Executes Code
Not Running	No	No	No
Active	Yes	Yes	Yes
Inactive	Mostly	No	Yes
Background	No	No	Yes

State	Visible	Receives Events	Executes Code
Suspended	No	No	No

You can see what applications are in the background or suspended by double-tapping the Home button to get to the multitasking display. You can do this in the Simulator by doing Command-Shift-H twice. (Recently run applications that have been terminated may also appear in this display.)

Figure 9.9 Background and suspended applications in the multitasking display



An application in the suspended state will remain in that state as long as there is adequate system memory. When the operating system decides memory is getting low, it terminates suspended applications as needed. A suspended application gets no notification that it is about to be terminated; it is simply removed from memory. (An application may remain in the multitasking display after it has been terminated, but it will have to be relaunched when tapped.)

When an application changes its state, the application delegate is sent a message. Here are some of the messages from the `UIApplicationDelegate` protocol that announce application state transitions. (These are also shown in Figure 9.8.)

```
optional func application(application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [NSObject : AnyObject]?) -> Bool
optional func applicationDidBecomeActive(application: UIApplication)
optional func applicationWillResignActive(application: UIApplication)
optional func applicationDidEnterBackground(application: UIApplication)
optional func applicationWillEnterForeground(application: UIApplication)
```

You can implement code in these methods to take the appropriate actions for your application. Additionally, there is a notification associated with each of these state changes; you observed the `UIApplicationDidEnterBackgroundNotification` earlier in this chapter. Transitioning to the background state is a good place to save any outstanding changes and the state of the application, because it is the last time your application can execute code before it enters the suspended state. Once in the suspended state, an application can be terminated at the whim of the operating system.

Writing to the Filesystem with NSData

Your archiving in `Homepwner` saves and loads the `itemKey` for each `Item`, but what about the images themselves? Let's extend the image store to save images as they are added and fetch them as they are needed.

The images for **Item** instances should also be stored in the Documents directory. You can use the image key generated when the user takes a picture to name the image in the filesystem.

Implement a new method in `ImageStore.swift` named `imagePathForKey(_)` to create a path in the documents directory using a given key.

```
func imagePathForKey(key: String) -> String {
    let documentsDirectories =
        NSSearchPathForDirectoriesInDomains(.DocumentDirectory, .UserDomainMask, true)
    let documentDirectory = documentsDirectories.first as String

    return documentDirectory.stringByAppendingPathComponent(key)
}
```

To save and load an image, you are going to copy the JPEG representation of the image into a buffer in memory. Instead of just malloc'ing a buffer, Objective-C programmers have a handy class to create, maintain, and destroy these sorts of buffers – **NSData**. An **NSData** instance holds some number of bytes of binary data, and you will use **NSData** to store image data.

In `ImageStore.swift`, modify `setImage(_:_forKey:)` to get a path and save the image.

```
func setImage(image: UIImage, forKey key: String) {
    imageDictionary[key] = image

    // Create full path for image
    let imagePath = imagePathForKey(key)

    // Turn image into JPEG data
    let data = UIImageJPEGRepresentation(image, 0.5)

    // Write it to full path
    data.writeToFile(imagePath, atomically: true)
}
```

Let's examine this code more closely. The function `UIImageJPEGRepresentation` takes two parameters: a **UIImage** and a compression quality. The compression quality is a float from 0 to 1, where 1 is the highest quality (least compression). The function returns an instance of **NSData**.

This **NSData** instance can be written to the filesystem by sending it the message `writeToFile(_:_atomically:)`. The bytes held in this **NSData** are then written to the path specified by the first parameter. The second parameter, **atomically**, is a boolean value. If it is `true`, the file is written to a temporary place on the filesystem, and, once the writing operation is complete, that file is renamed to the path of the first parameter, replacing any previously existing file. Writing atomically prevents data corruption should your application crash during the write procedure.

It is worth noting that this way of writing data to the filesystem is *not* archiving. While **NSData** instances can be archived, using the method `writeToFile(_:_atomically:)` copies the bytes in the **NSData** directly to the filesystem.

In `ImageStore.swift`, make sure that when an image is deleted from the store, it is also deleted from the filesystem:

```
func deleteImageForKey(key: String) {
    imageDictionary.removeValueForKey(key)

    let imagePath = imagePathForKey(key)
    NSFileManager.defaultManager().removeItemAtPath(imagePath, error: nil)
}
```

Now that the image is stored in the filesystem, the **ImageStore** will need to load that image when it is requested. The class method `imageWithContentsOfFile(_)` of **UIImage** will read in an image from a file, given a path.

In `ImageStore.swift`, replace the method `imageForKey(_)` so that the **ImageStore** will load the image from the filesystem if it does not already have it.

```

func imageForKey(key: String) -> UIImage? {
    return imageDictionary[key]

    if let existingImage = imageDictionary[key] {
        return existingImage
    }
    else {
        let imagePath = imagePathForKey(key)

        if let imageFromDisk = UIImage(contentsOfFile: imagePath) {
            setImage(imageFromDisk, forKey: key)
            return imageFromDisk
        }
        else {
            return nil
        }
    }
}

```

Build and run the application again. Take a photo for an item, exit the application, and then press the Home button. Launch the application again. Selecting that same item will show all its saved details – including the photo you just took.

Also, notice that the images were saved immediately after being taken, while the instances of **Item** were saved only when the application entered the background. You save the images right away because they are just too big to keep in memory for long.

Low-Memory Warnings

When the system is running low on RAM, it issues a low memory warning to the running application. The application responds by freeing up any resources that it does not need at the moment and can easily recreate. View controllers, during a low memory warning, have the method **didReceiveMemoryWarning()** triggered.

Objects other than view controllers may have data that they are not using and can recreate later. The **ImageStore** is such an object – when a low-memory warning occurs, it can release its ownership of the images by emptying its **imageDictionary**. Then if another object ever asks for a specific image again, that image can be loaded into memory from the filesystem.

In order to have objects that are not view controllers respond to low memory warnings, you must use the notification center. Whenever a low-memory warning occurs, **UIApplicationDidReceiveMemoryWarningNotification** is posted to the notification center. Objects that want to implement their own low-memory warning handlers can register for this notification.

In **ImageStore.swift**, override **init()** to register the image store as an observer of this notification.

```

override init() {
    super.init()

    let nc = NSNotificationCenter.defaultCenter()
    nc.addObserver(self,
                   selector: "clearCache:",
                   name: UIApplicationDidReceiveMemoryWarningNotification,
                   object: nil)
}

```

Now, when a low-memory warning is posted, the notification center will call the method **clearCache(_)** on the **ImageStore** instance (Figure 9.6).

In **ImageStore.swift**, implement **clearCache(_)** to remove all the instances of **UIImage** from the **ImageStore**'s **imageDictionary**.

```

func clearCache(note:NSNotification) {
    println("Flushing \(imageDictionary.count) images out of the cache")
    imageDictionary.removeAll(keepCapacity: false)
}

```

Removing an object from a dictionary relinquishes ownership of the object, so flushing the cache causes all of the images to lose an owner. Images that are not being used by other objects are destroyed, and when they are needed again, they will be reloaded from the filesystem. If an image is currently displayed in the **DetailViewController**'s `imageView`, then it will not be destroyed since it is owned by the `imageView`. When the **DetailViewController**'s `imageView` loses ownership of that image (either because the **DetailViewController** was popped off the stack or a new image was chosen), then it is destroyed. It will be reloaded later if needed.

Build and run the app in the simulator. Create or load in some images. Then select Simulate Memory Warning in the Hardware menu. You should see a log statement indicating that the cache has been flushed out.

More on NSNotificationCenter

Notifications are another form of *callbacks*, like delegation and target-action pairs. However, unlike delegation and target-action pairs, which require that the object responsible for an event send a message directly to its delegate or targets, notifications use a middle-man: the **NSNotificationCenter**.

Notifications are represented by instances of **NSNotification**. Each **NSNotification** has a name (used by the notification center to find observers), an object (the object that is responsible for posting the notification), and an optional user info dictionary that contains additional information that the poster wants observers to know about. For example, if for some reason the status bar's frame changes, **UIApplication** posts a `UIApplicationDidChangeStatusBarFrameNotification` with a user info dictionary. In the dictionary is the new frame of the status bar. If you received the notification, you could get the frame like this:

```
func statusBarMovedOrResized(note: NSNotification) {
    if let userInfo = note.userInfo {
        let wrappedFrame = userInfo[UIApplicationStatusBarFrameUserInfoKey] as NSValue
        let newFrame = wrappedFrame.CGRectValue()

        // ... use frame here ...
        println("new frame: \(newFrame)")
    }
}
```

How can you know what is in the `userInfo` dictionary? Each notification is documented in the class reference. Most say “This notification does not contain a `userInfo` dictionary.” For notifications with `userInfo` dictionaries, all the keys and what they map to will be listed.

The last argument of `addObserver(_:selector:name:object:)` is typically `nil` – which means, no matter what object posted a “Fire!” notification, the observer will get sent its message. You can specify a specific object for this argument and the observer will only get notified if that object posts the notification it has registered for, while ignoring any other object that posts the same notification.

One purpose of the notification center is to allow multiple objects to register a callback for the same event. Any number of objects can register as an observer for the same notification name. When the notification occurs, all of those objects are sent the message they registered with (in no particular order). Thus, notifications are a good solution when more than one object is interested in an event. For example, many objects might want to know about a rotation event, so Apple used notifications for that.

One final point: the **NSNotificationCenter** has nothing to do with inter-app communication, push notifications, or local notifications. It is simply for communication between objects in a single application.

Model-View-Controller-Store Design Pattern

In this exercise, you expanded on the **ItemStore** to allow it to save and load **Item** instances from the filesystem. The controller object asks the **ItemStore** for the model objects it needs, but it does not have to worry about where those objects actually came from. As far as the controller is concerned, if it wants an object, it will get one; the **ItemStore** is responsible for making sure that happens.

The standard Model-View-Controller design pattern calls for the controller to bear the burden of saving and loading model objects. However, in practice, this can become overwhelming – the controller is simply too busy handling the interactions between model and view objects to deal with the details of how objects are fetched and saved. Therefore, it is useful to move the logic that deals with where model objects come from and where they are saved to into another type of object: a *store*.

A store exposes a number of methods that allow a controller object to fetch and save model objects. The details of where these model objects come from or how they get there is left to the store. In this chapter, the store worked with a simple file. However, the store could also access a database, talk to a web service, or use some other method to produce the model objects for the controller.

One benefit of this approach, besides simplified controller classes, is that you can swap out *how* the store works without modifying the controller or the rest of your application. This can be a simple change, like the directory structure of the data, or a much larger change, like the format of the data. Thus, if an application has more than one controller object that needs to save and load data, you only have to change the store object.

Many developers talk about the Model-View-Controller design pattern. In this chapter, we have extended the idea to a *Model-View-Controller-Store* design pattern.

Bronze Challenge: PNG

Instead of saving each image as a JPEG, save it as a PNG.

For the More Curious: Application State Transitions

Let's write some quick code to get a better understanding of the different application state transitions.

In `AppDelegate.swift`, implement the application state transition delegate methods so that they print out the name of the method. You will need to add four more methods. (Check to make sure the template has not already created these methods before writing brand new ones.) Rather than hard-coding the name of the method in the `NSLog`, use the `_cmd` implicit variable you met in Chapter 20. Recall that `_cmd` is the method's selector, and you convert it to an `NSString` using `NSStringFromSelector`.

```
func applicationWillResignActive(application: UIApplication) {
    println("applicationWillResignActive(_:)")}
```

```
func applicationDidEnterBackground(application: UIApplication) {
    println("applicationDidEnterBackground(_:)")}
```

```
func applicationWillEnterForeground(application: UIApplication) {
    println("applicationWillEnterForeground(_:)")}
```

```
func applicationDidBecomeActive(application: UIApplication) {
    println("applicationDidBecomeActive(_:)")}
```

```
func applicationWillTerminate(application: UIApplication) {
    println("applicationWillTerminate(_:)")}
```

Now, add the following `println` statement to the top of `application(_:didFinishLaunchingWithOptions:)`.

```
func application(application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
    println("application(_:didFinishLaunchingWithOptions:)")
    ...}
```

Build and run the application. You will see that the application gets sent `application(_:didFinishLaunchingWithOptions:)` and then `applicationDidBecomeActive(_:)`. Play around some to see what actions cause what transitions.

Press the Home button, and the console will report that the application briefly inactivated and then went to the background state. Relaunch the application by tapping its icon on the Home screen or in the multitasking display. The console will report that the application entered the foreground and then became active.

Press the Home button to exit the application again. Then, double-press the Home button to open the multitasking display. Swipe the Homeowner application up and off this display to quit the application. Note that no message is sent to your application delegate at this point – it is simply terminated.

For the More Curious: Reading and Writing to the Filesystem

In addition to archiving and **NSData**'s binary read and write methods, there are a few more methods for transferring data to and from the filesystem. One of them, Core Data, is coming up in Chapter 14. A couple of others are worth mentioning here.

Using **NSData** works well for binary data. For text data, **String** has two instance methods **writeToFile(_:_:encoding:error:)** and **init?(contentsOfFile:encoding:error:)**. They are used as follows:

```
// A local variable to store an error object if one comes back
var error: NSError?
let success = someString.writeToFile(somePath,
    atomically: true,
    encoding: NSUTF8StringEncoding,
    error: &error)
if !success {
    if let error = error {
        println("Error writing file: \(error.localizedDescription)")
    }
}

let myEssay = String(contentsOfFile: somePath,
    encoding: NSUTF8StringEncoding,
    error: &error)
if myEssay == nil {
    if let error = error {
        println("Error reading file: \(error.localizedDescription)")
    }
}
```

What is that **NSError** object? Some methods might fail for a variety of reasons. For example, writing to the filesystem might fail because the path is invalid or the user does not have permission to write to the specified path. An **NSError** object contains the reason for a failure. You can access the **localizedDescription** property of **NSError** for a human-readable description of the error. This is something you can show to the user or print to a debug console.

The syntax for getting back an **NSError** instance is a little strange. An error object is only created if an error occurred; otherwise, there is no need for the object. When a method can return an error through one of its arguments, you create a local variable that is a pointer to an **NSError** object. Notice that you do not instantiate the error object – that is the job of the method you are calling. Instead, you pass the *address* of your pointer variable (**&error**) to the method that might generate an error. If an error occurs in the implementation of that method, an **NSError** instance is created, and your pointer is set to point at that new object. If you do not care about the error object, you can always pass **nil**.

Note that in many languages, anything unexpected results in an exception being thrown. Among Swift programmers, exceptions are nearly always used to indicate programmer error. When an exception *is* thrown, the information about what went wrong is in an **NSError** object. That information is usually just a hint to the programmer like “You tried to access the 7th object in this array, but there are only two.” The symbols for the call stack (as it appeared when the exception was thrown) are also in the **NSError**.

When do you use **NSError** and when do you use **NSError**? If you are writing a method that should only be called with an odd number as an argument, throw an exception if it is called with an even number – the caller is making an error and you want to help that programmer find the error in his ways. If you are writing a method that wants to read the contents of a particular directory, but does not have the necessary privileges, create an **NSError** and pass it back to the caller to indicate why you were unable to fulfill this very reasonable request.

Property list **serializable** objects can also be written to the file system. The only objects that are *property list* **serializable** are **String**, **NSNumber** (including primitives like **Int**, **Double**, and **Bool**), **NSDate**, **NSData**, **Array**, and **Dictionary**. When an **Array** or **Dictionary** is written to the filesystem with these methods, an *XML property list* is created. An XML property list is a collection of tagged values:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
  <dict>
    <key>firstName</key>
    <string>Christian</string>
    <key>lastName</key>
    <string>Keur</string>
  </dict>
  <dict>
    <key>firstName</key>
    <string>Aaron</string>
    <key>lastName</key>
    <string>Hillegass</string>
  </dict>
</array>
</plist>

```

XML property lists are a convenient way to store data because they can be read on nearly any system. Many web service applications use property lists as input and output. The code for writing and reading a property list looks like this:

```

let authors: AnyObject = [["firstName":"Christian", "lastName":"Keur"],
                         ["firstName":"Aaron", "lastName":"Hillegass"]]

// Write array to disk
if NSPropertyListSerialization.propertyList(authors, isValidForFormat: .XMLFormat_v1_0) {
    var error: NSError?
    if let data = NSPropertyListSerialization.dataWithPropertyList(authors,
        format: .XMLFormat_v1_0,
        options: NSPropertyListWriteOptions.allZeros,
        error: &error) {
        data.writeToFile(path, atomically: true)
    } else {
        if let error = error {
            println("Error writing plist: \(error.localizedDescription)")
        }
    }
}

// Read array from disk
if let data = NSData(contentsOfFile: path) {
    var error: NSError?
    if let authors = NSPropertyListSerialization.propertyListWithData(data,
        options: NSPropertyListReadOptions.allZeros,
        format: nil,
        error: &error) as? [[String:String]] {
        println("Read in authors: \(authors)")
    } else {
        if let error = error {
            println("Error reading plist: \(error.localizedDescription)")
        }
    }
}

```

For the More Curious: The Application Bundle

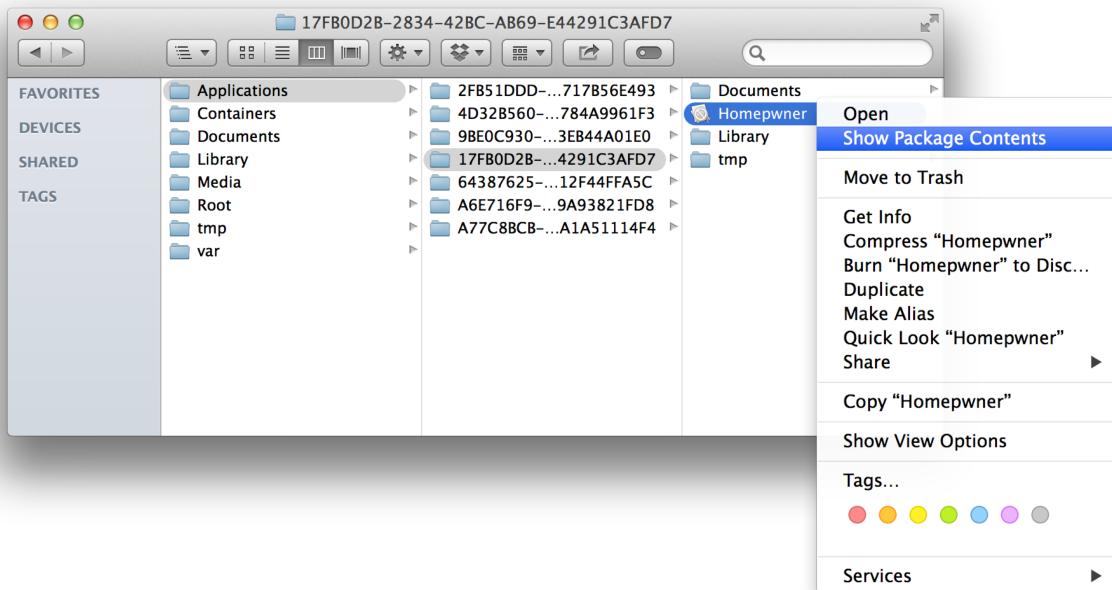
When you build an iOS application project in Xcode, you create an *application bundle*. The application bundle contains the application executable and any resources you have bundled with your application. Resources are things like XIB files, images, audio files – any files that will be used at runtime. When you add a resource file to a project, Xcode is smart enough to realize that it should be bundled with your application.

How can you tell which files are being bundled with your application? Select the Homeowner project from the project navigator. Check out the Build Phases pane in the Homeowner target. Everything under Copy Bundle Resources will be added to the application bundle when it is built.

Each item in the Homepwner target group is one of the phases that occurs when you build a project. The Copy Bundle Resources phase is where all of the resources in your project get copied into the application bundle.

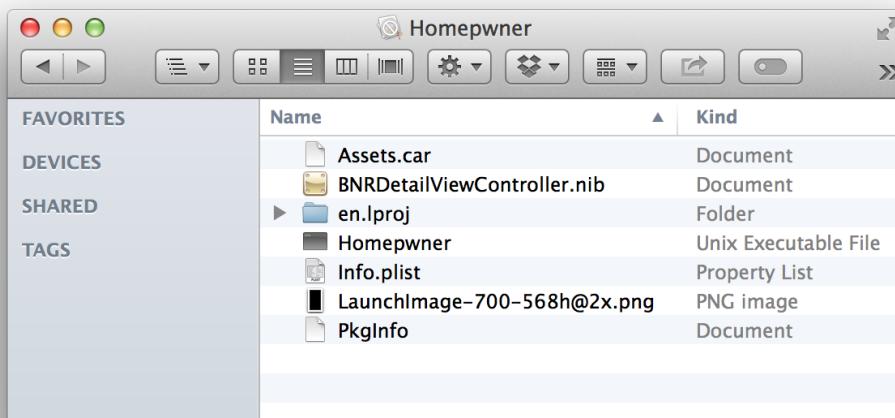
You can check out what an application bundle looks like on the filesystem after you install an application on the simulator. Navigate to `~/Library/Application Support/iPhone Simulator/(version number)/Applications`. The directories within this directory are the application sandboxes for applications installed on your computer's iOS simulator. Opening one of these directories will show you what you expect in an application sandbox: an application bundle and the `Documents`, `tmp`, and `Library` directories. Right- or Command-click the application bundle and choose Show Package Contents from the contextual menu (Figure 9.10).

Figure 9.10 Viewing an application bundle



A Finder window will appear showing you the contents of the application bundle (Figure 9.11). When a user downloads your application from the App Store, these files are copied to their device.

Figure 9.11 The application bundle



You can load files from the application's bundle at runtime. To get the full path for files in the application bundle, you need to get a pointer to the application bundle and then ask it for the path of a resource.

```
// Get a reference to the application bundle
let applicationBundle = NSBundle.mainBundle()

// Ask for the path to a resource named myImage.png in the bundle
if let path = applicationBundle.pathForResource("myImage", ofType: "png") {
    // Do something with path
}
```

If you ask for the path to a file that is not in the application's bundle, this method will return `nil`. If the file does exist, then the full path is returned, and you can use this path to load the file with the appropriate class.

Also, files within the application bundle are read-only. You cannot modify them nor can you dynamically add files to the application bundle at runtime. Files in the application bundle are typically things like button images, interface sound effects, or the initial state of a database you ship with your application. You will use this method in later chapters to load these types of resources at runtime.

10

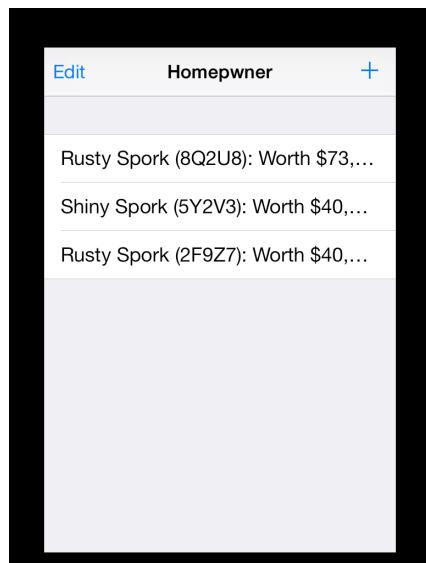
Introduction to Auto Layout

In this chapter, you will return to Homepwner and universalize it so that it will run on an iPad as well as on an iPhone. You will then use the Auto Layout system to ensure that Homepwner's detail interface appears as you want no matter what type of device it is running on.

Universalizing Homepwner

Currently, Homepwner can be run on the iPad simulator, but it will not look right.

Figure 10.1 An iPhone application running on a simulated iPad

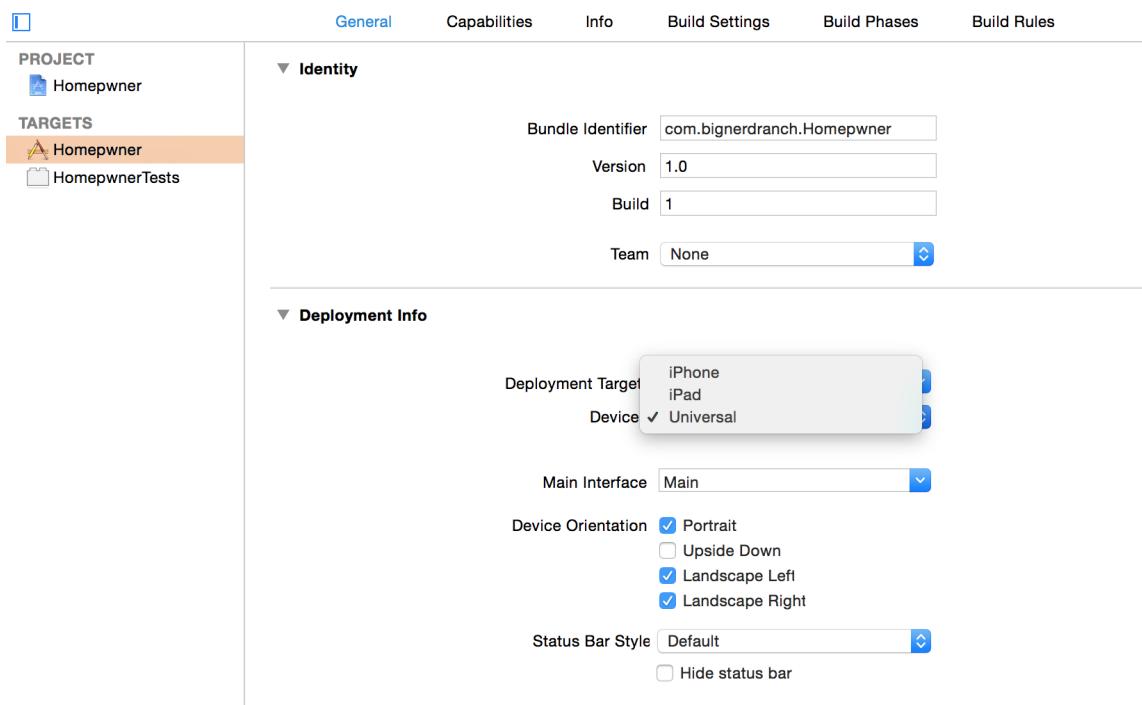


This is not what you want for your iPad users. You want Homepwner to run *natively* on the iPad so that it will look like an iPad app. A single application that runs natively on both the iPad and the iPhone is called a *universal application*.

Reopen your Homepwner project. In the project navigator, select the Homepwner project (the item at the top of the file list). Then select the Homepwner target in the project and targets list and the General tab. This tab presents a convenient interface for editing some of the target's properties.

Locate the Deployment Info section and change the Devices pop-up from iPhone to Universal (Figure 10.2).

Figure 10.2 Universalizing Homepwner



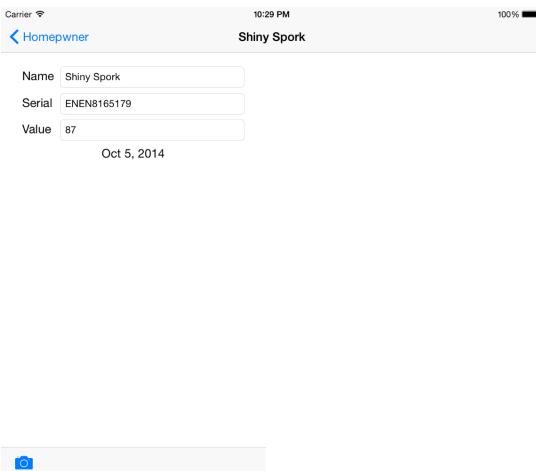
Homepwner is now a universal application. To see the difference, select an iPad simulator from the scheme pop-up menu and then build and run the application. Homepwner looks much better. The table view and its rows are sized appropriately for the iPad screen.

Figure 10.3 A universal application running on the iPad



Now add a new item to get to the detail interface. Here the results are not so good:

Figure 10.4 Detail interface does not automatically scale



The table view interface knew how to resize itself for the iPad sized-screen, but your custom detail interface needs some guidance. You will provide that guidance using Auto Layout.

The Auto Layout System

In Chapter 2, you learned that a view's `frame` specifies its size and position relative to its superview. So far, you have defined the `frames` of your views with absolute coordinates either programmatically or by configuring them in Interface Builder. Absolute coordinates, however, make your layout fragile because they assume that you know the size of the screen ahead of time.

Using Auto Layout, you can describe the layout of your views in a relative way that allows the `frames` to be determined at runtime so that the `frames`' definitions can take into account the screen size of the device that the application is running on.

The screen size for each device is listed in Table 10.1. (Remember that points are used when laying out your interface instead of pixels. A retina device has the same size screen in points as a non-retina device, even though it has many more pixels.)

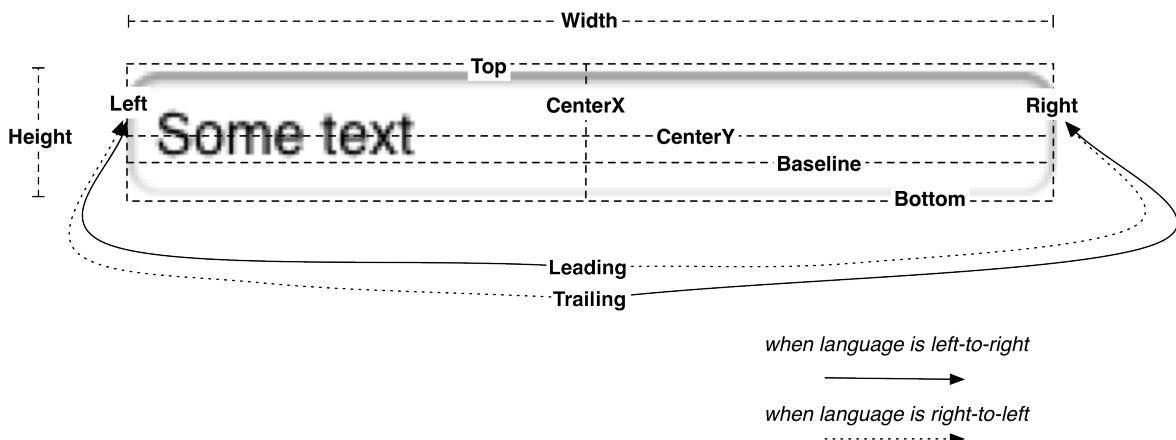
Table 10.1 Device Screen Sizes

Screen Size	Width x Height (points)
3.5-inch	320 x 480
4.0-inch	320 x 568
4.7-inch	375 x 667
5.5-inch	414 x 736
All iPads	768 x 1024

Alignment rectangle and layout attributes

The Auto Layout system works with yet another rectangle for a view – the *alignment rectangle*. This rectangle is defined by several *layout attributes* (Figure 10.5).

Figure 10.5 Layout attributes defining an alignment rectangle of a view



Width/Height	These values determine the alignment rectangle's size.
Top/Bottom/Left/Right	These values determine the spacing between the given edge of the alignment rectangle and the alignment rectangle of another view in the hierarchy.
CenterX/CenterY	These values determine the center point of the alignment rectangle.
Baseline	This value is the same as the bottom attribute for most, but not all, views. For example, UITextField defines its baseline to be the bottom of the text it displays rather than the bottom of the alignment rectangle. This keeps “descenders” (letters like ‘g’ and ‘p’ that descend below the baseline) from being obscured by a view right below the text field.
Leading/Trailing	These values are used with text-based views like UITextField and UILabel . If the language of the device is set to a language that reads left-to-right (e.g., English), then the leading attribute is the same as the left attribute and the trailing attribute is the same as the right attribute. If the language reads right-to-left (e.g., Arabic), then the leading attribute is on the right and the trailing attribute is on the left.

By default, every view in a XIB file has an alignment rectangle, and every view hierarchy uses Auto Layout. But, as you have seen in your detail view, the default will not always work as you want. This is when you need to step in.

You do not define a view's alignment rectangle directly. You do not have enough information (screen size!) to do that. Instead, you provide a set of *constraints*. Taken together, these constraints allow the system to determine the layout attributes, and thus the alignment rectangle, for each view in the view hierarchy.

Constraints

A *constraint* defines a specific relationship in a view hierarchy that can be used to determine a layout attribute for one or more views. For example, you might add a constraint like “the vertical space between these two views should always be 8 points” or “these views must always have the same width.” A constraint can also be used to give a view a fixed size, like “this view's height should always be 44 points.”

You do not need to have a constraint for every layout attribute. Some values may come directly from a constraint; others will be computed by the values of related layout attributes. For example, if a view's constraints set its left edge and its width, then the right edge is already determined ($\text{left edge} + \text{width} = \text{right edge}$, always).

If, after all of the constraints have been considered, there is still an ambiguous or missing value for a layout attribute, then there will be errors and warnings from Auto Layout and your interface will not look as you expect on all devices. Debugging these problems is important, and you will get some practice later in this chapter.

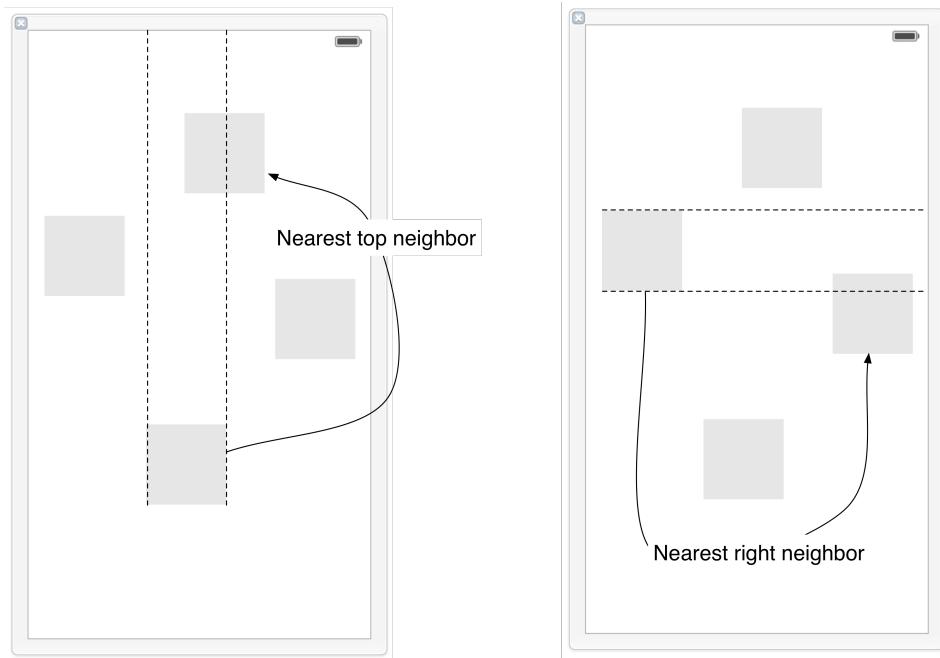
How do you come up with constraints? Let's see how using the view in the **DetailViewController**'s view hierarchy that will be simple to constrain – the toolbar.

First, describe what you want the view to look like regardless of screen size. For the toolbar, you could describe it like this:

- The toolbar should sit at the bottom of the screen.
- The toolbar should be as wide as the screen.
- The toolbar's height should be 44 points. (This is Apple's standard for instances of **UIToolbar**.)

To turn this description into constraints in Interface Builder, it will help to understand how to find a view's *nearest neighbor*. The nearest neighbor is the closest sibling view in the specified direction (Figure 10.6).

Figure 10.6 Nearest neighbor



If a view does not have any siblings in the specified direction, then the nearest neighbor is its superview, also known as its *container*.

Now you can spell out the constraints for the toolbar:

1. The toolbar's bottom edge should be 0 points away from its nearest neighbor (which is its container – the view of the **DetailViewController**).
2. The toolbar's left edge should be 0 points away from its nearest neighbor.
3. The toolbar's right edge should be 0 points away from its nearest neighbor.
4. The toolbar's height should be 44 points.

If you consider the second and third constraints, you can see that there is no need to explicitly constrain the toolbar's width. It will be determined from the constraints on the toolbar's left and right edges. There is also no need to constrain the toolbar's top edge. The constraints on the bottom edge and the height will determine the value of the `top` attribute.

Now that you have a plan for the toolbar, you can add these constraints. Constraints can be added using Interface Builder or in code.

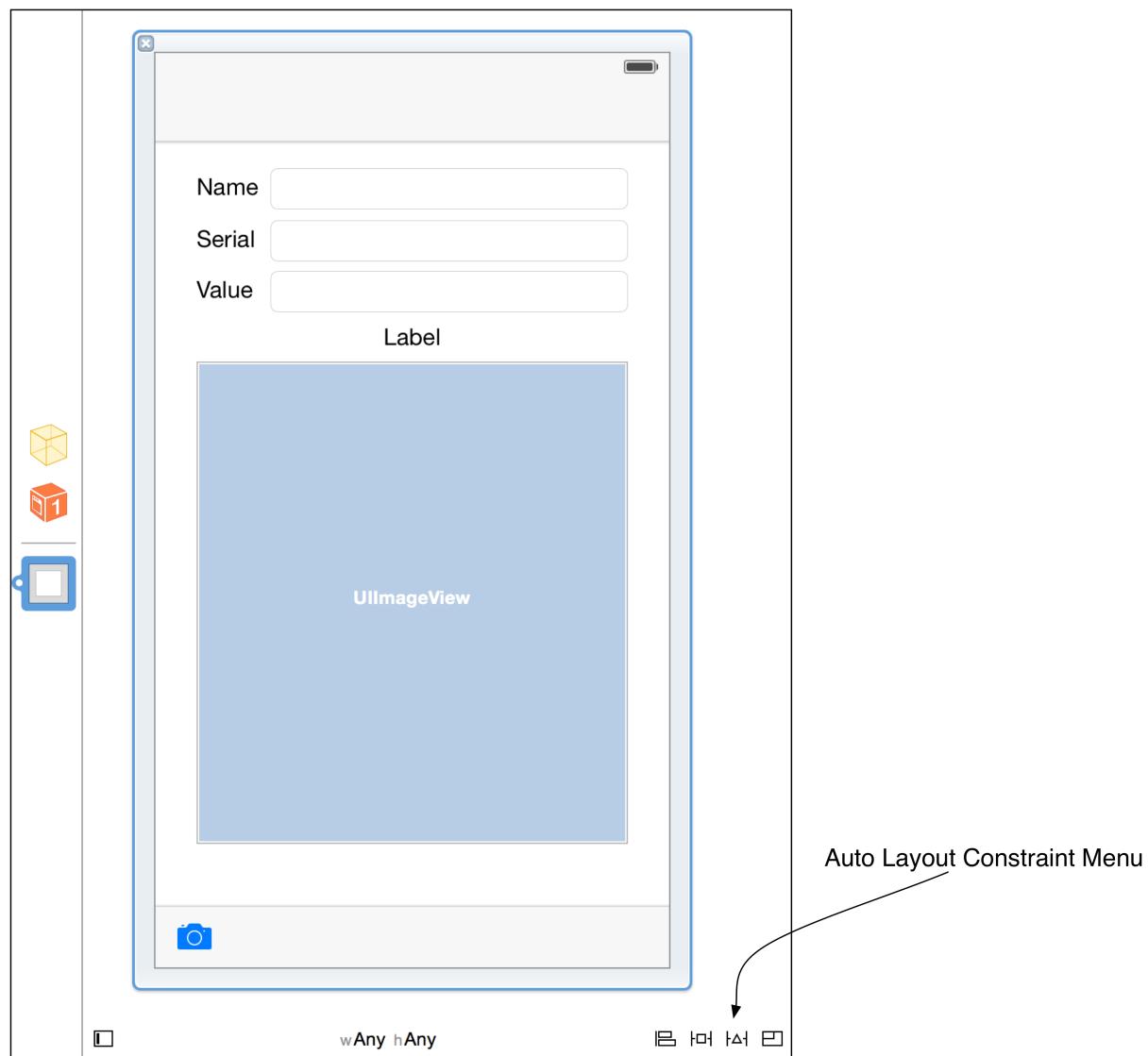
Apple recommends that you add constraints using Interface Builder whenever possible. This is what you will do in this chapter. However, if your views are created and configured programmatically, then you can add constraints in code. In Chapter 11, you will get a chance to practice that approach.

Adding Constraints in Interface Builder

Open `DetailViewController.xib`. First, select the image view in the canvas and delete it from the XIB file. You will recreate the image view and add its constraints programmatically in Chapter 11.

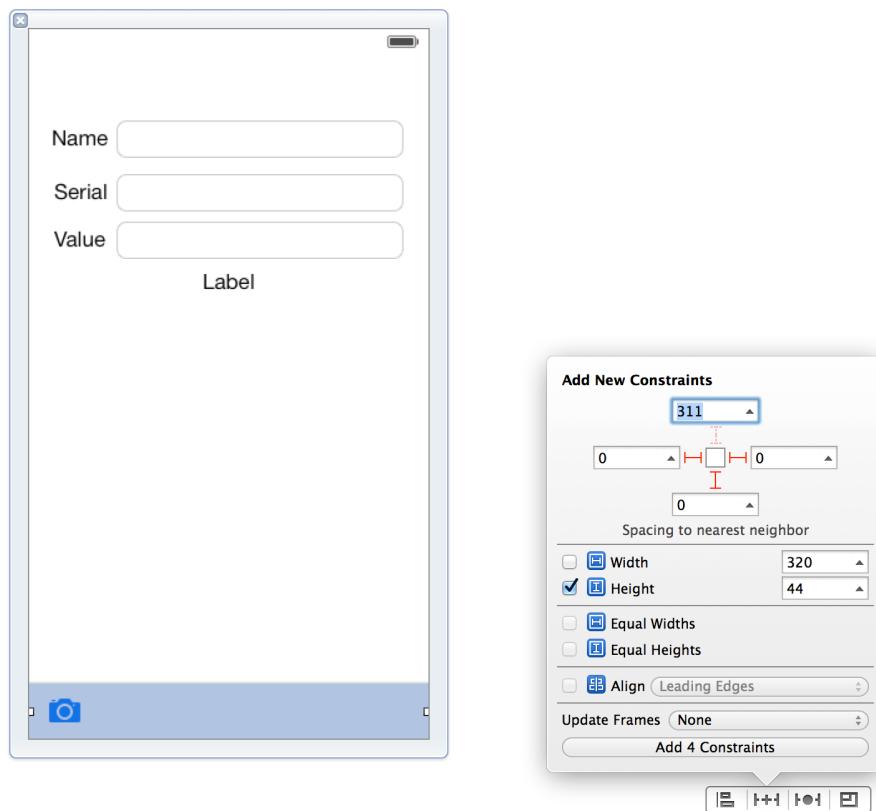
Select the toolbar on the canvas. At the bottom righthand corner of the canvas, find the Auto Layout constraint menu (Figure 10.7).

Figure 10.7 Selecting a constraint



Click the icon (the second from the left) to reveal the Pin menu. This menu shows you the current size and position of the toolbar. You can add all of the necessary constraints for the toolbar in this menu (Figure 10.8).

Figure 10.8 Adding 4 constraints to the toolbar



At the top of the Pin menu are four values that describe the toolbar's current spacing from its nearest neighbor on the canvas. For the toolbar, you are only interested in the bottom, left, and right values. They are all **0**, meaning that these edges of the toolbar are currently **0** points away from the toolbar's nearest neighbor in those directions. The toolbar has no siblings to its bottom, left, or right, so its nearest neighbor in all three directions is its container, the view of the **DetailViewController**.

To turn these values into constraints, click the orange struts separating the values from the square in the middle. The struts will become solid lines.

In the middle of the menu, find the toolbar's **Height**. It is currently **44** points, which is what you want. To constrain the toolbar's height based on this value, check the box next to **Height**. The button at the bottom of the menu reads **Add 4 Constraints**. Click this button.

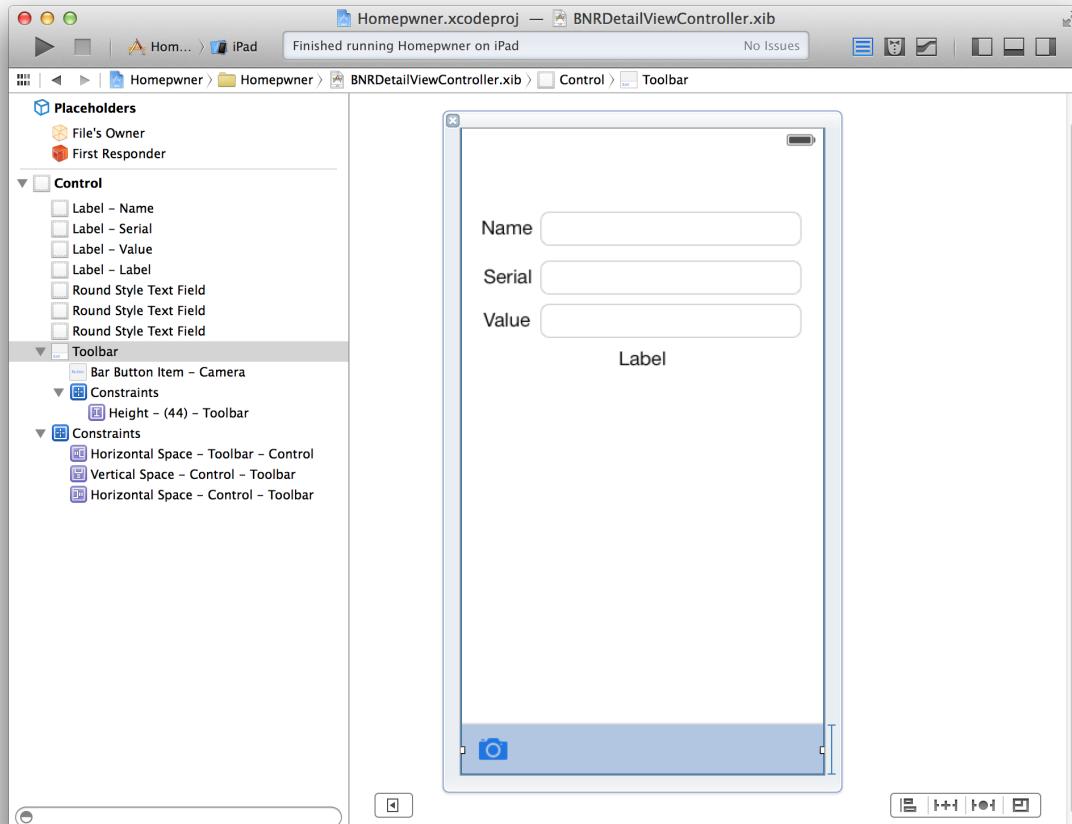
Build and run the application on the iPad simulator. Create an item and select it to navigate to the detail interface. The toolbar will appear at the bottom of the screen and be as wide as the screen (Figure 10.9).

Figure 10.9 Toolbar appearing correctly



You can see the constraints that you just added in the dock to the left of the canvas. Find the Constraints section and reveal its contents. However, you will see only three constraints here. The fourth constraint, the fixed height of the toolbar, is in a separate Constraints section underneath Toolbar. Reveal the contents of this Constraints section (Figure 10.10).

Figure 10.10 Constraints in dock



Why the division? Once a constraint is created, it is added to a particular view object in the view hierarchy. Which view gets a constraint is based on which views that constraint affects. The three edge constraints are added to the Control because they apply to both the toolbar and its superview, the view of the **DetailViewController**. (Recall that you changed the class of this view object from **UIView** to **UIControl** at the end of Chapter 8 to enable tapping on this view to dismiss the keyboard.) The height constraint, on the other hand, is added to the toolbar because it applies only to the toolbar.

In a XIB file, a constraint is added to the appropriate view automatically. For constraints created programmatically, creating and adding are distinct steps. In the next chapter, you will see how to determine which view a constraint should be added to when you create constraints programmatically.

If you select any of these constraints in the dock, a blue line will appear on the canvas representing the constraint. (Some constraints will be harder to see than others.) Selecting the view will show you all the constraints influencing that view.

You can delete any constraint by selecting it in the dock or by selecting its representative line in the canvas and then pressing Delete. Try it out. Delete the height constraint and then select the toolbar on the canvas and use the Pin menu to add it back.

Adding more constraints

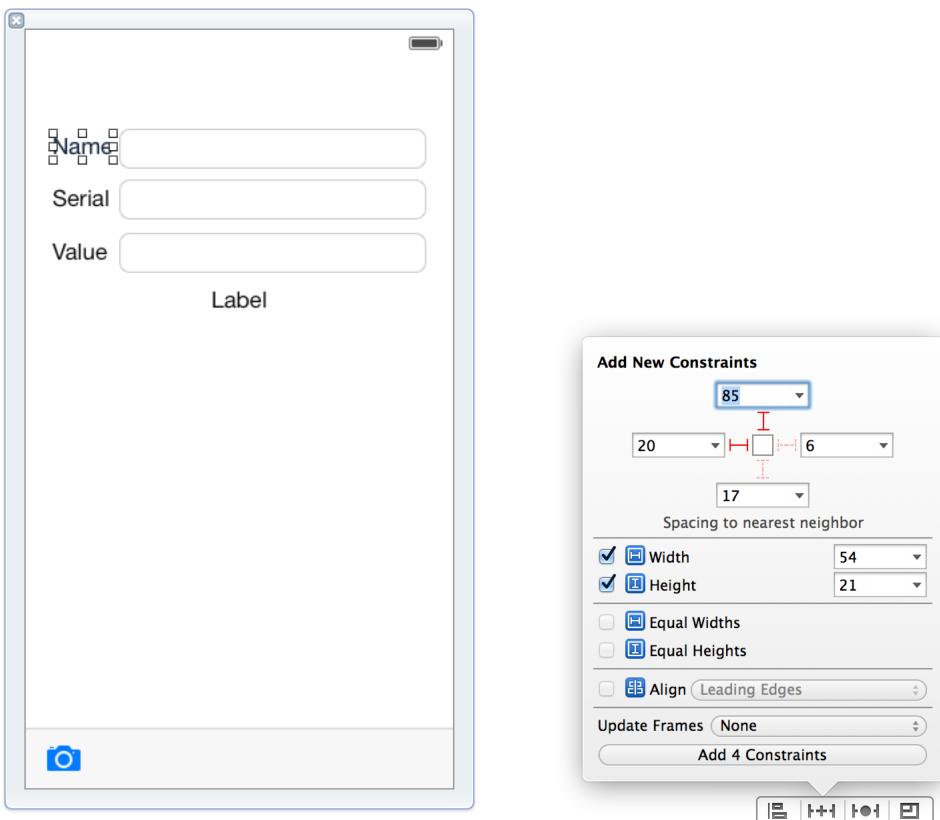
Let's turn now to the Name label. This label's size and position are fine right now when run on the iPad. However, you still need to constrain this view so that it will appear appropriately if it is presented in a different language or font size.

Regardless of language, font size, or screen size, you want the Name label's size to be fixed and its position to be near the top left corner. Select the Name label in the canvas and click the Pin menu button in the constraints menu.

Select the top and left strut at the top of the pin menu. The nearest neighbor in these directions is the Name label's container (also the view of the **DetailViewController**). Also, check the boxes for Width and Height to fix the label at its current size in points.

Your Pin menu should look something like Figure 10.11. Note that your values are unlikely to exactly match this figure, and that is OK. You are creating a constraint based on the position of the view in *your* canvas. If you change your values to match Figure 10.11, then your constraints will not match the position of your views on the canvas, and you will get misplaced view warnings. You will learn about these warnings shortly, but it is best to avoid them for now.

Figure 10.11 Example constraints for the Name label



Click Add 4 Constraints at the bottom of the menu.

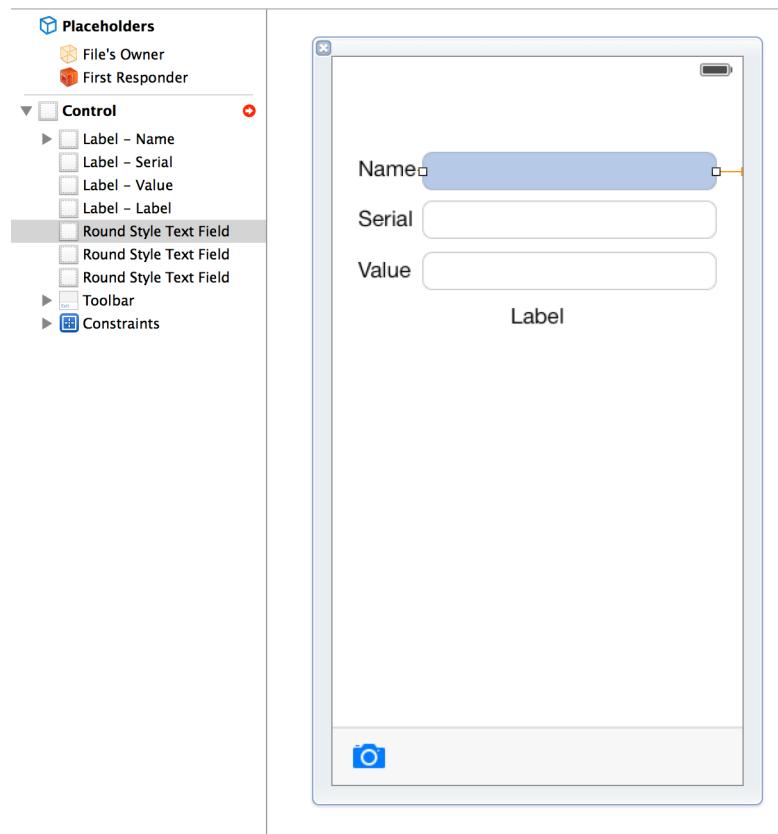
Now consider the text field to the right of the Name label. Regardless of screen size, the text field's width should stretch from its position to the left of the Name label to fill most of the screen.

Select the text field and open the Pin menu. At the top of the menu, select the left and right struts, and then add these two constraints. This pins the leading edge of the text field to the Name label and the trailing edge to 20 points from the container. By pinning the text field's trailing edge to its container, you are ensuring that the text field will always stretch to fill most of the screen regardless of the screen's size.

You now have a problem. Notice that the lines representing the constraints on the text field are orange instead of blue. This color difference means that the text field does not have enough constraints for Auto Layout to unambiguously specify its alignment rectangle.

To get more information about this problem, find and click the red icon in the dock next to Control.

Figure 10.12 Insufficient constraints for text field

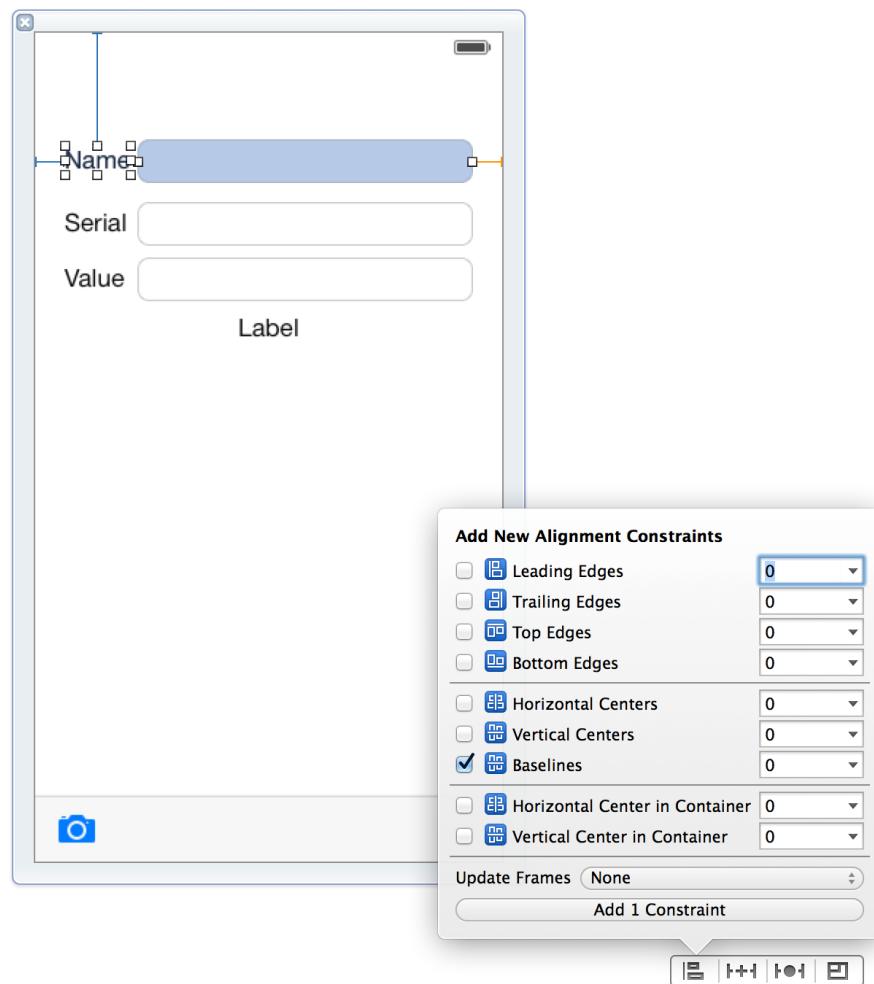


According to Interface Builder, you are missing a constraint. The text field needs its Y (vertical) position constrained. You could fix this problem by opening the Pin menu and selecting the top strut. This would pin the text field some distance from its container.

There is a better design choice: align the text field with the Name label instead. You can align two or more views according to any layout attribute. Here the best choice is to align baselines. That way the text that the user enters in the text field will line up with the text in the Name label.

On the canvas, select the text field and hold the Shift key down to select the Name label at the same time. From the constraints menu, click the icon to reveal the Align menu. Check the box next to **Baselines** and add 1 constraint.

Figure 10.13 Aligning baselines of label and text field

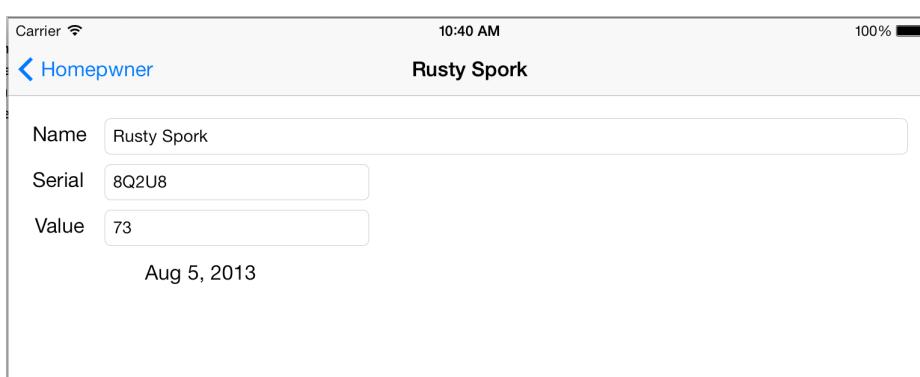


Now the lines representing the constraints are blue again, and the red icon has disappeared; the text field has enough constraints to size and position it unambiguously.

A missing constraint (also called an ambiguous layout) is just one of the problems that can pop up when adding constraints. Two other types are conflicting constraints and constraints not matching the view's size and position on the canvas. Later in the chapter, you will see how to debug these problems.

Build and run the project on iPad. Select an item, and you should see the top text field stretching to fill in the extra space provided by the iPad's screen size (Figure 10.14).

Figure 10.14 Name field stretching



Adding even more constraints

Now that you know about pin and align constraints, you can add constraints for the rest of the views, starting with the Serial label.

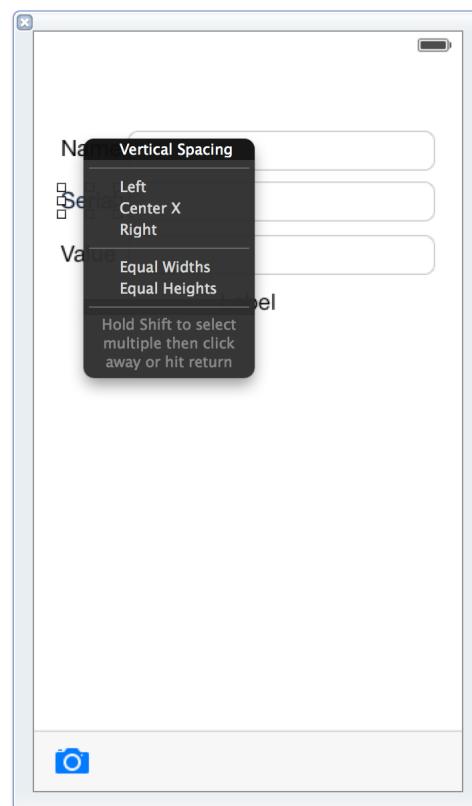
Here are the constraints that you need to add for the Serial label:

- top edge should be pinned at its current distance from the Name label
- leading (left) edge should be aligned with the Name label's leading edge
- height and width should be fixed at their current values

So far, you have added constraints using the Pin and Align menus. You can also add constraints by Control-dragging on the canvas. This dragging is similar to setting up outlets and actions. You drag from one view to another. After you release the mouse button, you get a list of constraints that you can add. The list is context-sensitive; it is populated based on the direction of the drag and what views you are dragging to and from.

Let's see how Control-dragging works. Select the Serial label in the canvas. Then Control-drag from this label to the Name label. When you let go of the mouse button, a menu will appear (Figure 10.15).

Figure 10.15 Adding constraint by Control-dragging between views



Select Vertical Spacing from the menu to fix the vertical distance between these two views at its current value. This is identical to opening the Pin menu and selecting the top strut.

Control-drag to the Name label again. This time, select Left from the menu. This is identical to opening the Align menu and checking Leading Edges.

Finally, you need to fix the label's height and width at their current values. Because these constraints affect only the Serial label, you do not Control-drag to another view. Instead, make a very short diagonal Control-drag within the Serial label.

When the menu appears, hold down the Shift key to select both Width and Height. (If you are only seeing one of these choices, then your Control-drag was vertical or horizontal instead of diagonal. Try again.)

Now you need to constrain the text field to the right of the `Serial` label. This view needs three constraints:

- leading edge should be pinned at its current distance from the `Serial` label
- baseline should be aligned with the `Serial` label's baseline
- trailing (right) edge should be aligned with the `Name` text field's trailing edge

Select the text field and Control-drag to the `Serial` label. Let go and Shift-click **Horizontal Spacing** and **Baseline**. Then Control-drag from the `Serial` text field up to the `Name` text field. Select **Right**.

There are three more views that need constraints: the `Value` label, the text field to its right, and the label that displays the date. Using the constraints menu or Control-dragging, add the following constraints:

For the `Value` label...

- top edge should be pinned at its current distance from the `Serial` label
- leading edge should be aligned with the `Serial` label's leading edge
- height and width should be fixed at their current values

For the text field...

- leading edge should be pinned at its current distance from the `Value` label
- baseline should be aligned with the `Value` label's baseline
- trailing edge should be aligned with the `Serial` text field's trailing edge

For the date label...

- top edge should be pinned at its current distance from the text field that displays the item's value
- leading edge should be pinned to the leading edge of the `Value` label
- trailing edge should be pinned to the trailing edge of `valueField`
- height should be fixed at its current value

Build and run the application on iPad. The `DetailViewController` should look good at this point.

Priorities

Each constraint has a priority level that is used to determine which constraint wins when more than one constraint conflicts. A priority is a value from 1 to 1000, where 1000 is a required constraint. By default, constraints are required, so all of the constraints that you have added are required. This means that the priority level would not help if you had conflicting constraints. Instead, Auto Layout would report an issue regarding unsatisfiable constraints. Typically, you find the constraints that conflict and then either remove one (often, the culprit is an obvious accident) or reduce the priority level of a constraint to resolve the conflict but keep all the constraints in play. You will learn more about debugging this issue in the next section.

Debugging Constraints

You have added constraints to `DetailViewController.xib` that will allow Auto Layout to determine alignment rectangles for every view in the hierarchy and give you the layout that you want on the iPhone and the iPad.

Given the sheer number of constraints, it is easy to introduce problems. You can miss a constraint, you could have constraints that conflict, or you could have a constraint that conflicts with how a view appears on the canvas.

Fortunately, there are several tools you can use to debug Auto Layout constraints. Let's take a look at each of these in turn and see ways to fix them.

Ambiguous layout

An ambiguous layout occurs when there is more than one way to fulfill a set of constraints. Typically, this means that you are missing at least one constraint.

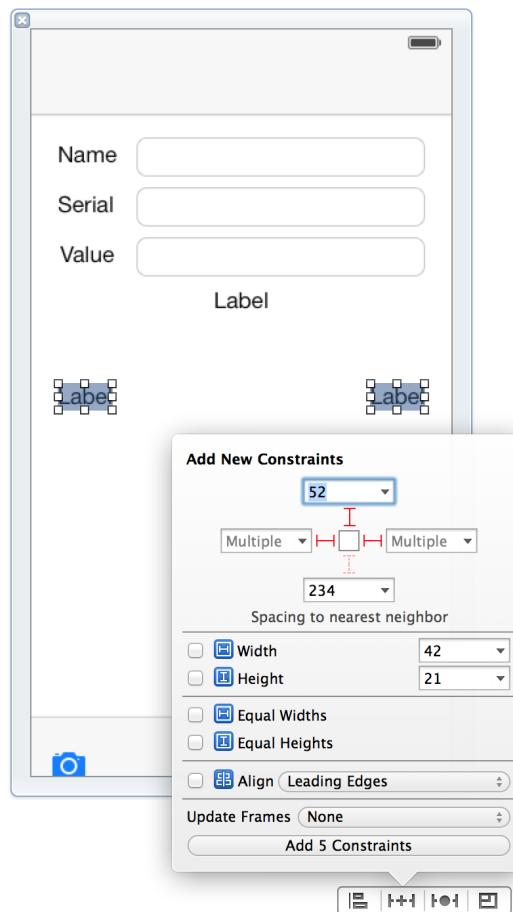
Currently, there are no ambiguous layouts, so let's introduce some. Add two labels to the **UIControl** under the `dateLabel` and position them next to one another. In the attributes inspector, change the background color of the labels to light gray so that you can see their frames. The interface should look like Figure 10.16.

Figure 10.16 New labels



Now let's add some constraints to both labels. Hold down the Shift key and select both labels. Open the Pin Auto Layout menu, select the top, left, and right struts at the top, and then click Add 5 Constraints (Figure 10.17).

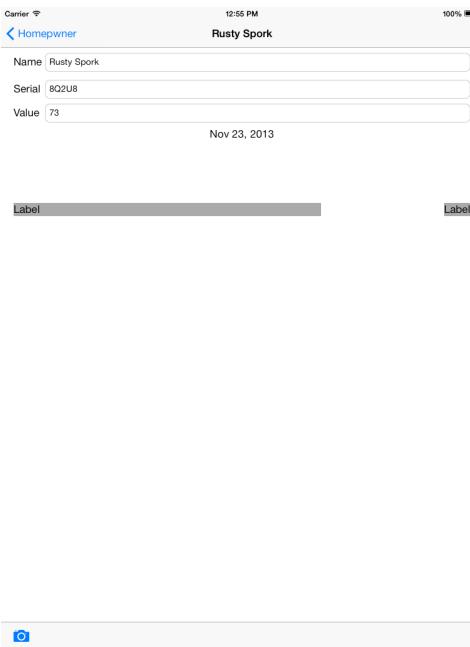
Figure 10.17 Adding constraints to multiple views at once



You have pinned three edges for two labels, so you may be wondering “Why 5 constraints and not 6?” The answer is that the trailing edge constraint of the label on the left and the leading edge constraint of the label on the right are the same constraint. Interface Builder recognizes this and only adds one constraint to satisfy both attributes.

If you build and run on an iPhone, everything will look fine, but running on the iPad is a different story. Build and run the project on an iPad and navigate to the detail interface. One of the two labels is wider than the other (Figure 10.18).

Figure 10.18 Width of labels is surprising on iPad



These labels do not have enough constraints to unambiguously define their frames. Auto Layout takes its best guess at runtime, and it is not what you wanted on the iPad. You are going to use two **UIView** methods, **hasAmbiguousLayout** and **exerciseAmbiguousLayout**, to debug this situation.

Open **DetailViewController.swift**. Override the method **viewDidLayoutSubviews** to check if any of its subviews has an ambiguous layout.

```
override func viewDidLayoutSubviews() {
    for subview in view.subviews as [UIView] {
        if subview.hasAmbiguousLayout() {
            println("AMBIGUOUS: \(subview)")
        }
    }
}
```

viewDidLayoutSubviews gets called any time the view changes in size (for example, when the device is rotated) or when it is first presented on the screen.

Build and run the application on the iPad simulator and navigate to the **DetailViewController**. Then check the console; it will report that the two labels are ambiguous. (Note that the output in the console is often duplicated, resulting in twice as many messages as you would expect.)

You can go a step further to actually see the other way this layout might appear. In **DetailViewController.swift**, edit the **backgroundTapped(_:)** method to call the method **exerciseAmbiguityInLayout** on any ambiguous views.

```
@IBAction func backgroundTapped(sender: AnyObject) {
    view.endEditing(true)

    for subview in view.subviews as [UIView] {
        if subview.hasAmbiguousLayout() {
            subview.exerciseAmbiguityInLayout()
        }
    }
}
```

Build and run the application again. Once on the **DetailViewController**, tap the background view anywhere. The width of the two labels will swap.

Figure 10.19 Tapping in the background demonstrates the other possible layout



Neither of the widths of the labels has been constrained, and so there is more than one solution to the system of Auto Layout equations. Because of this, there is an ambiguous layout and tapping the background switches between the two possible solutions. Due to the other constraints you have specified, as long as one of the labels has its width constrained, the other label's width can be determined. You will get rid of the ambiguous layout by giving the two labels equal widths.

In `DetailViewController.xib`, Control-drag from one label to the other, and then select Equal Widths. Build and run the application on iPad. Your labels have the same width. Check the console to confirm that there are no more ambiguous layouts. Tapping on the background will have no effect on the interface.

Your interface is once again properly set up. All views have enough constraints to fully construct their alignment rectangle, and so there are no more ambiguous views.

In `DetailViewController.xib`, select and delete the two test labels.

The `exerciseAmbiguityInLayout` method is purely a debugging tool that allows Auto Layout to show you where your layouts could potentially end up. You should never leave this code in an application that you are shipping.

In `DetailViewController.swift`, delete `viewDidLayoutSubviews` and delete the code that calls `exerciseAmbiguityInLayout` in `backgroundTapped(_)`.

```
override func viewDidLayoutSubviews() {
    for subview in view.subviews as [UIView] {
        if subview.hasAmbiguousLayout() {
            println("AMBIGUOUS: \(subview)")
        }
    }
}

@IBAction func backgroundTapped(sender: AnyObject) {
    view.endEditing(true)

    for subview in view.subviews as [UIView] {
        if subview.hasAmbiguousLayout() {
            subview.exerciseAmbiguityInLayout()
        }
    }
}
```

Unsatisfiable constraints

The problem of unsatisfiable constraints occurs when two or more constraints conflict. This often means that a view has too many constraints. To illustrate, let's introduce this problem to the **DetailViewController**.

In **DetailViewController.xib**, select the label that displays the date. In the attributes inspector, change its background to light gray so that you can see its frame in the layout. Next, pin the width of this label to its current value.

Just as before, if you were to build and run the application on an iPhone, everything would be fine. Build and run the application on an iPad. The label may look just as it did before, but take a look at the console.

```
Unable to simultaneously satisfy constraints.
Probably at least one of the constraints in the following list is one you don't want.
Try this: (1) look at each constraint and try to figure out which you don't expect;
(2) find the code that added the unwanted constraint or constraints and fix it.
(Note: If you're seeing NSAutoresizingMaskLayoutConstraints that you don't understand,
refer to the documentation for the UIView property
translatesAutoresizingMaskIntoConstraints)
(
    "<>NSLayoutConstraint:0xa333da0 H:[UILabel:0xa333ca0(280)]>",
    "<>NSLayoutConstraint:0xa394500 H:[UILabel:0xa333ca0]-(20)-|
        (Names: '|':UIControl:0xa38cd80 )",
    "<>NSLayoutConstraint:0xa394530 H:|-(20)-[UILabel:0xa333ca0]
        (Names: '|':UIControl:0xa38cd80 )",
    "<>NSAutoresizingMaskLayoutConstraint:0xa3a1a70 h=-&-
        v=-&- UIControl:0xa38cd80.width == _UIParallaxDimmingView:0xa37b140.width>",
    "<>NSAutoresizingMaskLayoutConstraint:0xa3a21d0 h=-&
        v=-& H:[_UIParallaxDimmingView:0xa37b140(768)]>"
)
Will attempt to recover by breaking constraint
<NSLayoutConstraint:0xa333da0 H:[UILabel:0xa333ca0(280)]>

Break on objc_exception_throw to catch this in the debugger.
The methods in the UIConstraintBasedLayoutDebugging category on UIView listed
in <UIKit/UIView.h> may also be helpful.
```

First, Xcode informs you that it is “unable to simultaneously satisfy constraints” and gives you some hints on what to look for. The console then lists all of the constraints that are related to the issue. Finally, you are told that one of the constraints will be ignored so that the label will have a valid frame. In this case, the constraint to pin the width will be ignored.

You can also just think through the problem. You constrained this label’s leading and trailing edges to resize with the superview. Then you constrained its width to a fixed value. These are conflicting constraints, and the solution is to remove one.

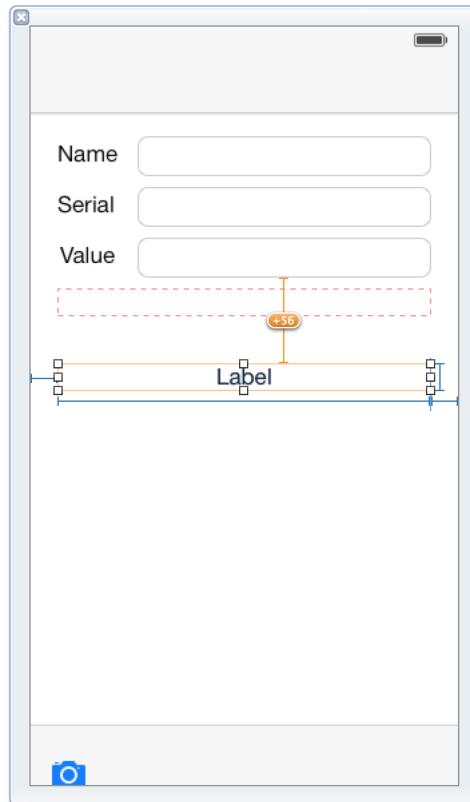
In **DetailViewController.xib**, delete the width constraint that you just added to the label and set its background color back to clear.

Misplaced views

If a view’s frame in a XIB does not match its constraints, then you have a misplaced view problem. This means that the frame of that view at runtime will not match how it currently appears on the canvas. Let’s cause a misplaced view problem.

Select the label that displays the date and drag it down a little bit so that the interface looks like Figure 10.20.

Figure 10.20 A misplaced view

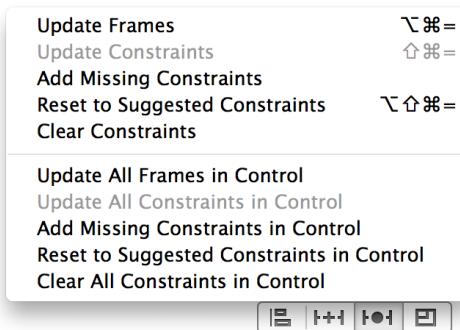


A rectangle with an orange, dashed border will appear where the label used to be. This is the runtime frame; at runtime, the existing constraints will position the label where this rectangle is and not where you just dragged it to.

How you fix the problem depends on whether the view's size and position on the canvas are what you want. If so, then you change the constraints to work with this new position. If not, then you change the view's size or position to match the constraints. Let's say that moving the label was an accident and that you want the view's position to match the existing constraints.

Select the date label. Then, in the Auto Layout constraints menu, select the icon to reveal the Resolve Auto Layout Issues menu (Figure 10.21).

Figure 10.21 Resolve Auto Layout Issues menu



Select **Update Frames** at the top. This will reposition the label to match its constraints.

On the other hand, if you wanted the constraints to change to match the new position of the view, you would choose to **Update Constraints**.

The **Resolve Auto Layout Issues** menu is very powerful. Here is a description of the items in the top half of the menu.

Update Frames	Adjusts the frame of the view to match its constraints.
Update Constraints	Adjusts the constraints of the view to match its frame.
Add Missing Constraints	For views with an ambiguous layout, this will add the necessary constraints to remove the ambiguity. However, the new constraints might not be what you want, so make sure to double-check and test this resolution.
Reset to Suggested Constraints	This will remove any existing constraints from the view and add new constraints. These suggested constraints are sensitive to the context of the view. For example, if the view is near the top of its superview, the suggested constraints will probably pin it to the top, whereas if the view is near the bottom of its superview, it will probably be pinned to the bottom.
Clear Constraints	All constraints are removed. If no explicit constraints are added to this view, it will have the default fixed position and size constraints added to it.

The bottom section repeats these options but applies them to all of the subviews instead of only the selected view(s).

Bronze Challenge: Practice Makes Perfect

Open the Resolve Auto Layout Issues menu and select Clear All Constraints in Control. Review Figure 10.4 or run the application on the iPad simulator and navigate to the detail interface to remind yourself of the initial problems. Add the constraints back on your own to achieve a reasonable-looking detail interface on the iPad.

Play with different ways of adding constraints (menus vs. Control-dragging), adding multiple constraints at once, and constraining multiple views at once. Use the debugging tools and the warnings and errors that Interface Builder provides to ensure that your constraints are sufficient.

Silver Challenge: Universalize Quiz

Make Quiz a universal application. If you run the universalized app on the iPad simulator without adding any constraints, the interface will look like this:

Figure 10.22 Universalized Quiz running on the iPad



Decide how the interface should look on the iPad and then add constraints in `Main.storyboard` to ensure that it will appear as you want on any device.

11

Auto Layout: Programmatic Constraints

In this chapter, you are going to interact with Auto Layout in code. Apple recommends that you create and constrain your views in a XIB file whenever possible. However, if your views are created in code, then you will need to constrain them programmatically.

To have a view to work with, you are going to recreate the image view programmatically and then constrain it in the `UIViewController` method `viewDidLoad`. This method will be called after the NIB file for `DetailViewController`'s interface has been loaded.

Recall that in Chapter 4, you overrode `loadView` to create views programmatically. If you are creating and constraining an entire view hierarchy, then you override `loadView`. If you are creating and constraining an additional view to add to a view hierarchy that was created by loading a NIB file, then you override `viewDidLoad` instead.

In `DetailViewController.swift`, implement `viewDidLoad` to create an instance of `UIImageView`.

```
override func viewDidLoad() {
    super.viewDidLoad()

    let iv = UIImageView()

    // The contentMode of the image view in the XIB was Aspect Fit:
    iv.contentMode = .ScaleAspectFit

    // Do not produce a translated constraint for this view
    iv.setTranslatesAutoresizingMaskIntoConstraints(false)

    // The image view was a subview of the view
    view.addSubview(iv)

    // The image view was pointed to by the imageView property
    imageView = iv
}
```

The line of code regarding translating constraints has to do with an older system for scaling interfaces – autoresizing masks. Before Auto Layout was introduced, iOS applications used autoresizing masks to allow views to scale for different-sized screens at runtime.

Every view has an autoresizing mask. By default, iOS creates constraints that match the autoresizing mask and adds them to the view. These translated constraints will often conflict with explicit constraints in the layout and cause an unsatisfiable constraints problem. The fix is to turn off this default translation by setting the property `translatesAutoresizingMaskIntoConstraints` to `NO`. (There is more about Auto Layout and autoresizing masks at the end of this chapter.)

Now let's consider how you want the image view to appear. The `UIImageView` should span the entire width of the screen and should maintain the standard 8 point spacing between itself and the `dateLabel` above and the toolbar below. Here are the constraints for the image view spelled out:

- left edge is 0 points from the image view's container
- right edge is 0 points from the image view's container

- top edge is 8 points from the date label
- bottom edge is 8 points from the toolbar

Apple recommends using a special syntax called *Visual Format Language* (VFL) to create constraints programmatically. This is how you will constrain the image view. However, there are times when a constraint cannot be described using VFL. In those cases, you must take another approach. You will see how to do that at the end of the chapter.

Visual Format Language

Visual Format Language is a way of describing constraints in a literal string. You can describe multiple constraints in one visual format string. A single visual format string, however, cannot describe both vertical and horizontal constraints. Thus, for the image view, you are going to come up with two visual format strings: one that constrains the horizontal spacing of the image view and one that constrains its vertical spacing.

Here is how you would describe the horizontal spacing constraints for the image view as a visual format string:

```
"H: |-0-[imageView]-0-|"
```

The H: specifies that these constraints refer to horizontal spacing. The view is identified inside square brackets. The pipe character (|) stands for the view's container. This image view, then, will be 0 points away from its container on its left and right edges.

When the number of points between the view and its container (or some other view) is 0, the dashes and the 0 can be left out of the string:

```
"H:|[imageView]|"
```

The string for the vertical constraints looks like this:

```
"V:[dateLabel]-8-[imageView]-8-[toolbar]"
```

Notice that “top” and “bottom” are mapped to “left” and “right”, respectively, in this necessarily horizontal display of vertical spacing. The image view is 8 points from the date label at its top edge and 8 points from the toolbar at its bottom edge.

You could write this same string like this:

```
"V:[dateLabel]-[imageView]-[toolbar]"
```

The dash by itself sets the spacing to the standard number of points between views, which is 8.

To see a little more of VFL grammar, consider a hypothetical situation. Imagine you had two image views with the following horizontal constraints:

- the horizontal spacing between the image views should be 10 points
- the lefthand image view's left edge should be 20 points from its superview
- the righthand image view's right edge should be 20 points from its superview

You could describe the three constraints in one visual format string:

```
"H:|-20-[imageViewLeft]-10-[imageViewRight]-20-|"
```

The syntax for a fixed size constraint is simply adding an equality operator and a value in parentheses inside a view's visual format:

```
"V:[someView(==50)]"
```

This view's height would be constrained to 50 points.

Creating Constraints

A constraint is an instance of the class **NSLayoutConstraint**. When creating constraints programmatically, you explicitly create one or more instances of **NSLayoutConstraint** and then add them to the appropriate view

object. Creating and adding constraints is one step when working with a XIB, but it is always two distinct steps in code.

You create constraints from a visual format string using the **NSLayoutConstraint** method:

```
class func constraintsWithVisualFormat(format: String,
    options opts: NSLayoutConstraintOptions,
    metrics: [NSObject : AnyObject]?,
    views: [NSObject : AnyObject]) -> [AnyObject]
```

This method returns an array of **NSLayoutConstraint** objects because a visual format string typically creates more than one constraint.

The first argument is the visual format string. For now, you can ignore the next two arguments, but the fourth is critical.

The fourth argument is a **Dictionary** that maps the names in the visual format string to view objects in the view hierarchy. The two visual format strings that you will use to constrain the image view refer to view objects by the names of the variables that point to them.

```
"H:|-[imageView]-|"
"V:[dateLabel]-[imageView]-[toolbar]"
```

However, a visual format string is just a string, so putting the name of a variable inside it means nothing unless you explicitly make the association.

In `DetailViewController.swift`, create a dictionary of names for the views at the end of `viewDidLoad`.

```
view.addSubview(iv)
imageView = iv

let nameMap = ["imageView":imageView,
               "dateLabel":dateLabel,
               "toolbar":toolbar]
}
```

You are using the names of your variables as keys, but you can use any key to name a view. The only exception is the `|` character, which is a reserved name for the superview (container) of the views being referenced in the string.

Next, in `DetailViewController.swift`, create the horizontal and vertical constraints for the image view:

```
let nameMap = ["imageView":imageView,
               "dateLabel":dateLabel,
               "toolbar":toolbar]

// imageView is 0 pts from superview at left and right edges
let horizontalConstraints =
    NSLayoutConstraint.constraintsWithVisualFormat("H:|-0-[imageView]-0-|",
        options: nil,
        metrics: nil,
        views: nameMap)

// imageView is 8 pts from dateLabel at its top edge...
// ... and 8 pts from toolbar at its bottom edge
let verticalConstraints =
    NSLayoutConstraint.constraintsWithVisualFormat("V:[dateLabel]-[imageView]-[toolbar]",
        options: nil,
        metrics: nil,
        views: nameMap)
}
```

Activating Constraints

You now have two arrays of **NSLayoutConstraint** objects. However, these constraints will have no effect on the layout until you explicitly activate them by setting their `active` property to `true`.

A convenient way of doing this is by using the **NSLayoutConstraint** class method `activateConstraints(_:)` which loops through an array of constraints and sets their `active` property to `true`.

In `DetailViewController.swift`, activate the constraints at the end of `viewDidLoad`.

```

let verticalConstraints =
    NSLayoutConstraint.constraintsWithVisualFormat("V:[dateLabel]-[imageView]-[toolbar]",
        options: nil,
        metrics: nil,
        views: nameMap)

NSLayoutConstraint.activateConstraints(horizontalConstraints)
NSLayoutConstraint.activateConstraints(verticalConstraints)
}

```

This is identical to looping over the constraints and setting their active property to true:

```

NSLayoutConstraint.activateConstraints(constraints)

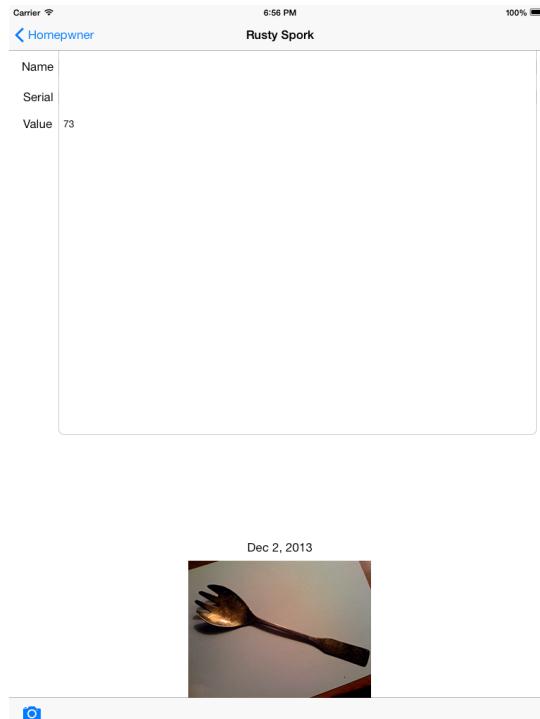
// Accomplishes the same thing as

for c in constraints as [NSLayoutConstraint] {
    c.active = true
}

```

Build and run the application. Create an item and select an image. Your detail interface may look all right or it may not. It depends on the size of the image that you selected. If the image you selected is small, you may be looking at something like this:

Figure 11.1 Small-size image gives unexpected results



What's going on? The text field for the value has become monstrously tall. To understand why this is happening, let's look at a view's intrinsic content size and how it interacts with Auto Layout.

(Note: If your `valueField` is missing, this is due to a current bug in Auto Layout. Baseline constraints – like the one between `valueField` and the `Value` label – are not being respected. Remove that baseline constraint, and replace it with a `CenterY` constraint. If you use the Control-click and drag approach to create this constraint, it will set the constant to the current difference between centers, in effect maintaining the baseline alignment.)

Intrinsic Content Size

Intrinsic content size is information that a view has about how big it should be based on what it displays. For example, a label's intrinsic content size is based on how much text it is displaying. In your case, the image view's intrinsic content size is the size of the image that you selected.

Auto Layout takes this information into consideration by creating intrinsic content size constraints for each view. Unlike other constraints, these constraints have two priorities: a content hugging priority and a content compression resistance priority.

Content hugging priority

tells Auto Layout how important it is that the view's size stay close to, or "hug", its intrinsic content. A value of **1000** means that the view should never be allowed to grow larger than its intrinsic content size. If the value is less than **1000**, then Auto Layout may increase the view's size when necessary.

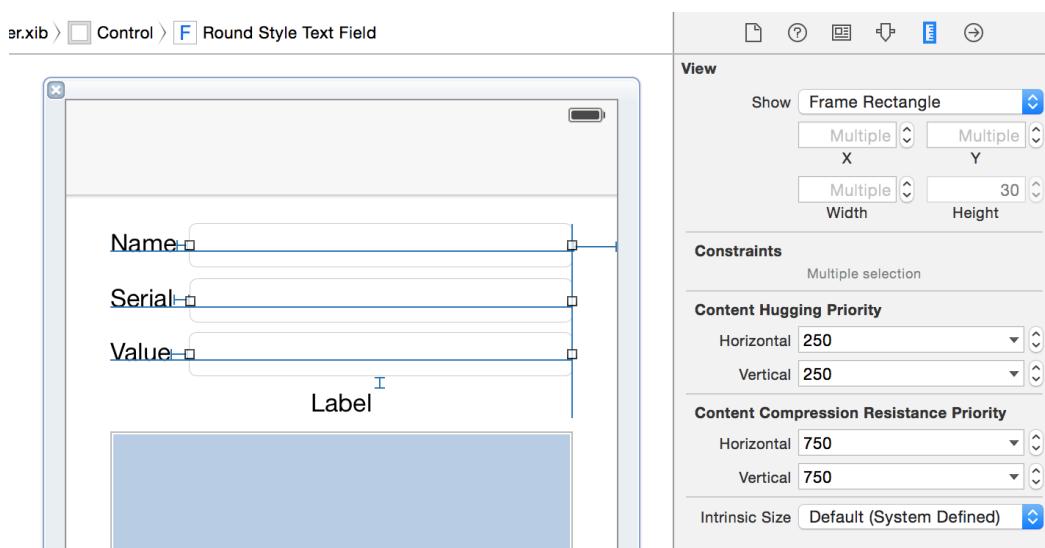
Content compression resistance priority

tells Auto Layout how important it is that the view avoid shrinking, or "resist compressing", its intrinsic content. A value of **1000** means that the view should never be allowed to be smaller than its intrinsic content size. If the value is less than **1000**, then Auto Layout may shrink the view when necessary.

In addition, both priorities have separate horizontal and vertical values so that you can set different priorities for a view's height and width. This makes a total of four intrinsic content size priority values per view.

You can see and edit these values in both code and Interface Builder. Reopen `DetailViewController.xib`. Shift-click to select all three text fields in the canvas. Head to the inspector and select the `W` tab to reveal the *size inspector*. Find the Content Hugging Priority and Content Compression Resistance Priority sections.

Figure 11.2 Content priorities



First, notice that these values are not **1000** and thus will never conflict with the constraints that you have added so far. This is why the layout will appear incorrectly with smaller-sized images. The value text field's content hugging vertical property is **250**, which is lower than that of the image view (which is **251**), so when faced with a small image, Auto Layout chooses to make the text field taller than its intrinsic content size. The image view is, in effect, pulling down on the text field's bottom edge making it taller.

It would be better if the image view had a smaller vertical content hugging priority than the other subviews. Open `DetailViewController.swift` and update `viewDidLoad` to lower this priority.

```
override func viewDidLoad() {
    super.viewDidLoad()

    let iv = UIImageView()
    iv.contentMode = .ScaleAspectFit
    iv.setTranslatesAutoresizingMaskIntoConstraints(false)

    view.addSubview(iv)
    imageView = iv

    imageView.setContentHuggingPriority(200, forAxis: .Vertical)
```

...

Build and run again. Now, when dealing with smaller-sized images, Auto Layout will change the image size and leave the height of the text fields alone.

The Other Way

There are times when a constraint cannot be created with a visual format string. For instance, you cannot use VFL to create a constraint based on a ratio, like if you wanted the date label to be twice as tall as the name label or if you wanted the image view to always be 1.5 times as wide as it is tall.

In these cases, you can create an instance of **NSLayoutConstraint** using the method

```
convenience init(item view1: AnyObject,
    attribute attr1: NSLayoutAttribute,
    relatedBy relation: NSLayoutRelation,
    toItem view2: AnyObject?,
    attribute attr2: NSLayoutAttribute,
    multiplier: CGFloat,
    constant c: CGFloat)
```

This method creates a single constraint using two layout attributes of two view objects. The multiplier is the key to creating a constraint based on a ratio. The constant is a fixed number of points, like you have used in your spacing constraints.

The layout attributes are defined as constants in the **NSLayoutConstraint** class:

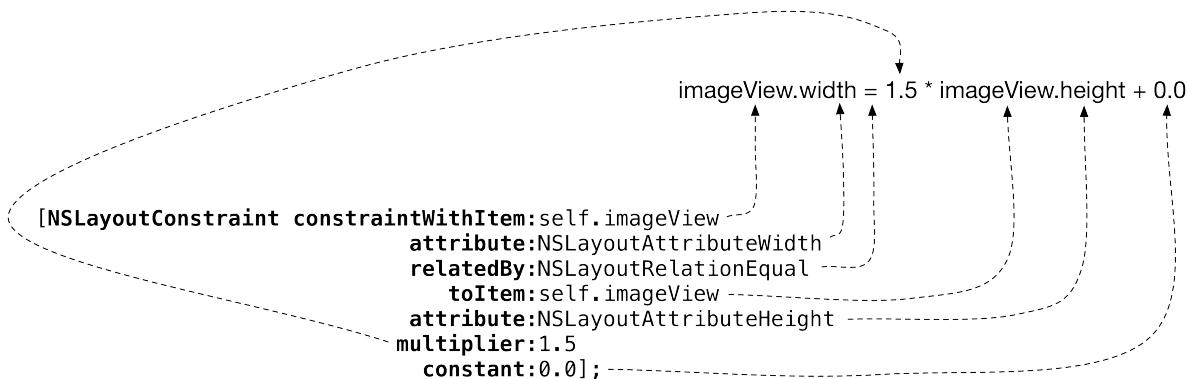
- **NSLayoutAttribute.Left**
- **NSLayoutAttribute.Right**
- **NSLayoutAttribute.Top**
- **NSLayoutAttribute.Bottom**
- **NSLayoutAttribute.Width**
- **NSLayoutAttribute.Height**
- **NSLayoutAttribute.Baseline**
- **NSLayoutAttribute.CenterX**
- **NSLayoutAttribute.CenterY**
- **NSLayoutAttribute.Leading**
- **NSLayoutAttribute.Trailing**

Let's consider a hypothetical constraint. Say you wanted the image view to be 1.5 times as wide as it is tall. You cannot do this with a visual format string, so you would create it individually instead with the following code. (Do not type this hypothetical constraint in your code! It will conflict with others you already have.)

```
let aspectConstraint = NSLayoutConstraint(item: imageView,
    attribute: .Width,
    relatedBy: .Equal,
    toItem: imageView,
    attribute: .Height,
    multiplier: 1.5,
    constant: 0.0)
```

To understand how this method works, think of this constraint as the equation shown in Figure 11.3.

Figure 11.3 **NSLayoutConstraint** equation



You relate a layout attribute of one view to the layout attribute of another view using a multiplier and a constant to define a single constraint.

For the More Curious: NSAutoresizingMaskLayoutConstraint

Before Auto Layout, iOS applications used another system for managing layout: *autoresizing masks*. Each view had an autoresizing mask that constrained the relationship between a view and its superview, but this mask could not affect relationships between sibling views.

By default, views create and add constraints based on their autoresizing mask. However, these translated constraints often conflict with your explicit constraints in your layout, which results an unsatisfiable constraints problem.

To see this happen, comment out the line in `viewDidLoad` that turns off the translation of autoresizing masks.

```
// The contentMode of the image view in the XIB was Aspect Fit:
    iv.contentMode = UIViewContentModeScaleAspectFit;

// Turn off old school layout handling
iv.translatesAutoresizingMaskIntoConstraints = NO;

// The image view was a subview of the view
[self.view addSubview:iv];
```

Now the image view has a resizing mask that will be translated into a constraint. Build and run the application and navigate to the detail interface. You will not like what you see. The console will report the problem and its solution.

Unable to simultaneously satisfy constraints.

Probably at least one of the constraints in the following list is one you don't want. Try this: (1) look at each constraint and try to figure out which you don't expect; (2) find the code that added the unwanted constraint or constraints and fix it. (Note: If you're seeing NSAutoresizingMaskLayoutConstraint that you don't understand, refer to the documentation for the `UIView` property `translatesAutoresizingMaskIntoConstraints`)

```
(
    "<NSLayoutConstraint:0x914a2e0 H:[UILabel:0x914a1e0(42)]>",
    "<NSLayoutConstraint:0x9153ee0
        H:|-|(20)-[UILabel:0x9149f00]   (Names: '|':UIControl:0x91496e0 )>",
    "<NSLayoutConstraint:0x9153fa0
        UILabel:0x9149970.leading == UILabel:0x9149f00.leading>",
    "<NSLayoutConstraint:0x91540c0
        UILabel:0x914a1e0.leading == UILabel:0x9149970.leading>",
    "<NSLayoutConstraint:0x9154420
        H:[UITextField:0x914fe20]-(20)-|   (Names: '|':UIControl:0x91496e0 )>",
    "<NSLayoutConstraint:0x9154450
        H:[UILabel:0x914a1e0]-(12)-[UITextField:0x914fe20]>",
    "<NSLayoutConstraint:0x912f5a0
        H:|-(NSSpace(20))-UIImageView:0x91524d0]   (Names: '|':UIControl:0x91496e0 )>",
    "<NSLayoutConstraint:0x91452a0
        H:[UIImageView:0x91524d0]-(NSSpace(20))-|   (Names: '|':UIControl:0x91496e0 )>",
    "<NSAutoresizingMaskLayoutConstraint:0x905f130
        h==& v==& UIImageView:0x91524d0.midX ==>"
```

Will attempt to recover by breaking constraint
`<NSLayoutConstraint:0x914a2e0 H:[UILabel:0x914a1e0(42)]>`

Let's go over this output. Auto Layout is reporting that it is "Unable to simultaneously satisfy constraints." This happens when a view hierarchy has constraints that conflict.

Then, the console spits out some handy tips and a list of all constraints that are involved. Each constraint's `description` is shown in the console. Let's look at the format of one of these constraints more closely.

```
<NSLayoutConstraint:0x9153fa0 UILabel:0x9149970.leading == UILabel:0x9149f00.leading>
```

This description indicates that the constraint located at memory address `0x9153fa0` is setting the leading edge of the **UILabel** (at `0x9149970`) equal to the leading edge of the **UILabel** (at `0x9149f00`).

Four of these constraints are instances of **NSLayoutConstraint**. The fifth, however, is an instance of **NSAutoresizingMaskLayoutConstraint**. This constraint is the product of the translation of the image view's autoresizing mask.

Finally, it tells you how it is going to solve the problem by listing the conflicting constraint that it will ignore. Unfortunately, it chooses poorly and ignores one of your explicit instances of **NSLayoutConstraint** instead of the **NSAutoresizingMaskLayoutConstraint**. This is why your interface looks like it does.

The note before the constraints are listed is very helpful: the **NSAutoresizingMaskLayoutConstraint** needs to be removed. Better yet, you can prevent this constraint from being added in the first place by explicitly disabling translation in **viewDidLoad**:

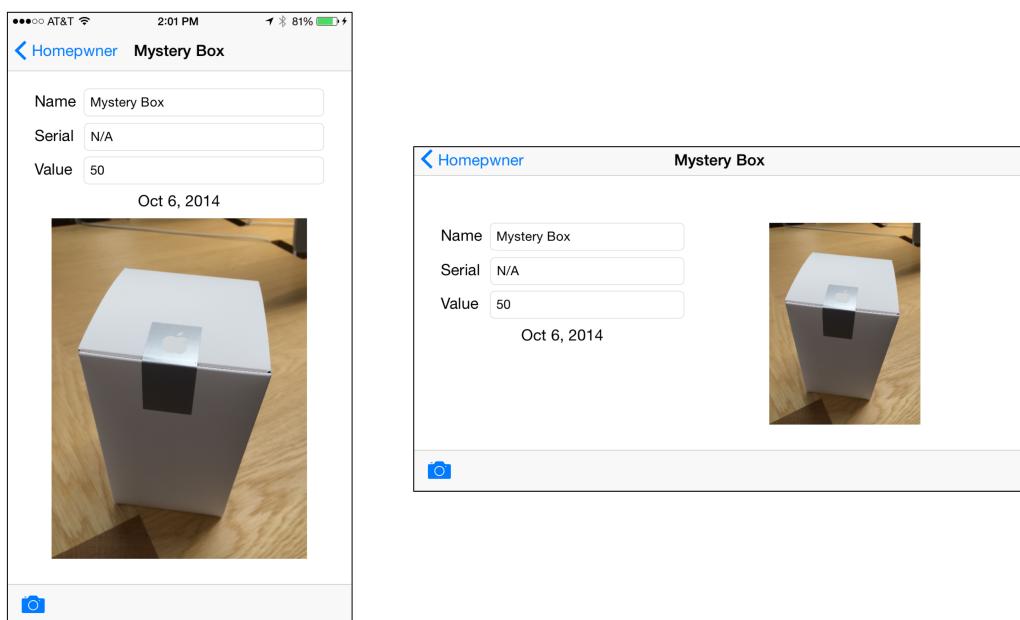
```
// The contentMode of the image view in the XIB was Aspect Fit:  
iv.contentMode = .ScaleAspectFit  
  
// Do not produce a translated constraint for this view  
iv.setTranslatesAutoresizingMaskIntoConstraints(false)  
  
// The image view was a subview of the view  
view.addSubview(iv)
```

12

Size Classes

In this chapter, you will have Homepwner have a different layout when in landscape on an iPhone. You will accomplish this using *size classes*.

Figure 12.1 Finished Application



Since Homepwner uses Auto Layout, its interface already scales appropriately as the screen size changes, either due to rotation or by running on different devices. But what if you want a different layout as the size of the screen changes? You can use size classes to get the job done.

In this section, you'll update Homepwner to have a different interface when it's in landscape.

Updating the Image View

The most convenient way to use size classes is with Interface Builder. Let's set up the image view from the last chapter in the XIB instead of from code.

Open `DetailViewController.swift` and remove `viewDidLoad`.

```

override func viewDidLoad() {
    super.viewDidLoad()

    let iv = UIImageView()
    iv.contentMode = .ScaleAspectFit
    iv.setTranslatesAutoresizingMaskIntoConstraints(false)
    view.addSubview(iv)
    imageView = iv

    imageView.setContentHuggingPriority(200, forAxis: .Vertical)

    let nameMap = ["imageView": imageView,
                  "dateLabel": dateLabel,
                  "toolbar": toolbar]

    let horizontalConstraints =
        NSLayoutConstraint.constraintsWithVisualFormat("H:|-0-[imageView]-0|",
            options: nil,
            metrics: nil,
            views: nameMap)

    let verticalConstraints =
        NSLayoutConstraint.constraintsWithVisualFormat("V:[dateLabel]-[imageView]-[toolbar]",
            options: nil,
            metrics: nil,
            views: nameMap)

    NSLayoutConstraint.activateConstraints(horizontalConstraints)
    NSLayoutConstraint.activateConstraints(verticalConstraints)
}

```

In `DetailViewController.xib`, drag a `UIImageView` onto the View and position it between the date label and the toolbar (Figure 12.2).

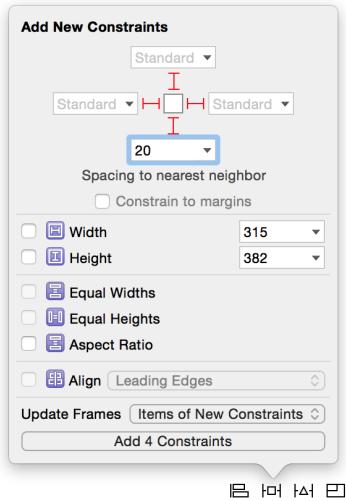
Figure 12.2 Image View



Connect the `DetailViewController`'s `imageView` property to the image view by Control-dragging from the File's Owner to the image view on the canvas.

Select the image view on the canvas. The constraints that were added to the image view programmatically in the previous chapter need to be added here in the XIB. Open the Pin menu and add constraints to the four sides of the image view (Figure 12.3). Then select Add 4 Constraints.

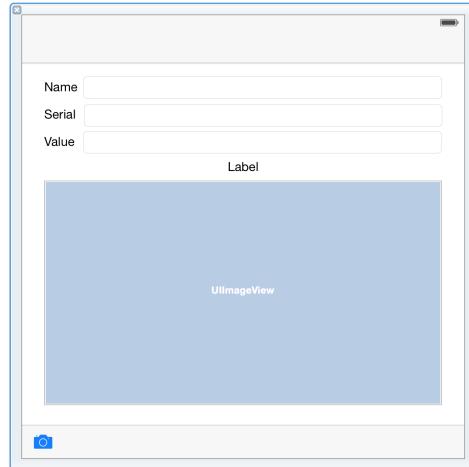
Figure 12.3 Constraints between image view and label



Open the Size inspector for the image view and lower its Vertical Content Hugging Priority to 200.

Switch to the File inspector (Command-Option-1) and confirm that the checkbox for Use Size Classes is checked. Then, select the root level View on the canvas. Open its Attribute inspector and change the Size back to Inferred. The View will go back to its original square shape, however the interface still looks great due to the auto layout constraints you added (Figure 12.4).

Figure 12.4 Size Classes Enabled



What is a Size Class

So what is a size class? A size class represents a relative amount of screen space in a given dimension. Each dimension - horizontal and vertical - can either have a *compact* or a *regular* size class. This gives us four possible combination of size classes. The table below shows the various size class combinations for the current iOS devices in both portrait and landscape.

Figure 12.5 Size Classes

	Portrait	Landscape
iPhone 5c iPhone 5s iPhone 6	Compact Width Regular Height	Compact Width Compact Height
iPhone 6 Plus	Compact Width Regular Height	Regular Width Compact Height
iPad Air iPad mini	Regular Width Regular Height	Regular Width Regular Height

Instead of thinking about our interfaces in terms of orientation or device, it's best to instead think about our interfaces in terms of size classes.

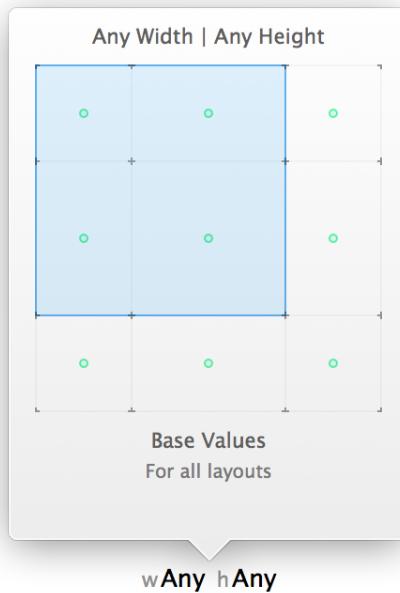
Updating the Interface

The primary purpose of size classes is to allow our interfaces to look and behave differently based on the general width and height of the interface. You saw this with **UISplitViewController** where the interface varies based on the horizontal size class. We can do the same within our applications, and storyboards make this very simple. In this section, you'll modify the interface for **DetailViewController** to display the text fields and image view side by side instead of on top of one another when running in a vertically compact environment.

Open **DetailViewController.xib**. At the bottom of Interface Builder you will see a button that says **w Any h Any**. The corresponding popup allows you to configure your interface differently for different size classes. **Any Any** is the default and is the most generic; when editing in the **Any Any** configuration, you are providing the interface that is common across all size classes.

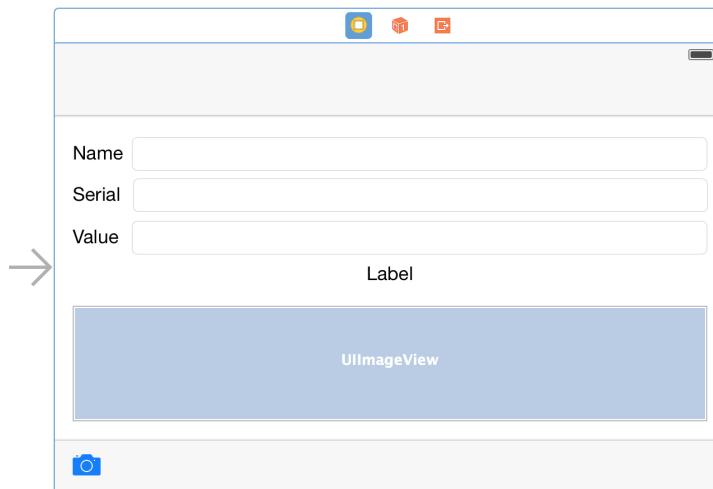
Click on the **Any Any** popup and notice the three by three grid that appears (Figure 12.6). This grid represents all of the size class combinations, as well as options that apply to a specific dimension of a size class. For example, you can edit the interface for Compact width and Compact height, Regular width and Compact height, or, if you only care about the height, you can edit the interface for Any width and Compact height. You'll use the latter for the **DetailViewController**.

Figure 12.6 Choosing a Size Class to Edit



With the size class popup open, choose the top middle box that corresponds to Any Width | Compact Height. Notice that the size of our view controllers are shorter (Figure 12.7). Additionally, the bottom of Interface Builder now has a blue bar to help remind you that you are no longer editing the layout for all size classes.

Figure 12.7 DetailViewController in the Any | Compact Environment



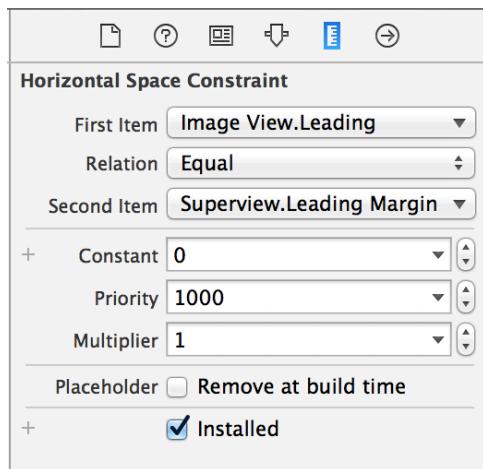
When editing the interface for a specific size class combination, you are able to change:

- whether a specific subview is installed
- whether a specific constraint is installed
- the constant of a constraint
- the font for subviews that display text

In **Homeowner**, we'll focus on the second item in that list - installing new and uninstalling some existing constraints. Let's start by making the image view take up the right side of the **DetailViewController**.

Select the **UIImageView** in the canvas, and then open the **Size inspector**. Double click on the **Leading Space to: Superview**. The **Size inspector** is now showing the details for the selected constraint (Figure 12.8).

Figure 12.8 Leading Constraint Size Inspector



At the bottom of the Size inspector, notice that the Installed box is checked. In a vertical compact environment, we do not want this constraint. Next to the checkbox, click on the + and then select Any Width | Compact Height (current). Finally, uncheck the box for Any C (Figure 12.9). The canvas now reflects the fact that this constraint doesn't exist in the current size class combination.

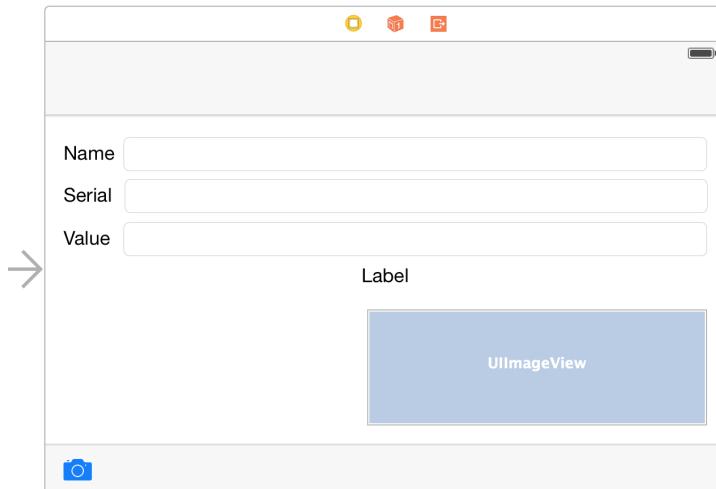
Figure 12.9 Uninstalling Constraints



The image view will take up half the screen width in a vertically compact environment. Control-drag from the image view to the toolbar and select the Equal Widths constraint. Back in the Size inspector, double click on the Equal Width to: Toolbar constraint. Notice how this is only installed in the current configuration, Any C. Change the Multiplier to be 0.5. (If the First Item is Toolbar.Width, click on the First Item drop-down and select Reverse First and Second Item and then make sure the Multiplier is still 0.5.)

Back on the canvas, make sure the image view is selected and then click on the Resolve Auto Layout Issues button. From the popup, choose Update Frames under Selected Views. The image view should now take up half of the width of its superview (Figure 12.10).

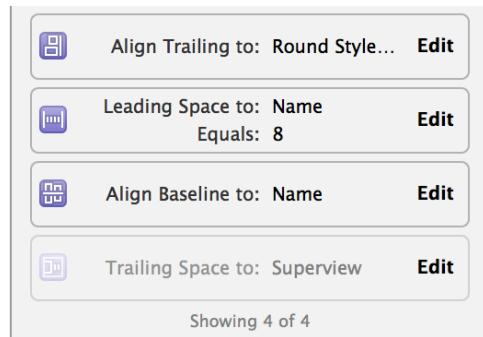
Figure 12.10 Updated Image View



Next, you'll update the other subviews to span the first half of the superview width. Since Auto Layout was set up intelligently, the text fields and labels are all anchored to `nameField`, the top text field.

Select the top text field and open the Size inspector. You'll use a shortcut to uninstall this constraint in this size class combination. Single click on the Trailing Space to: Superview and press Delete. You'll see that constraint grayed out indicating that it is uninstalled in this size class combination (Figure 12.11).

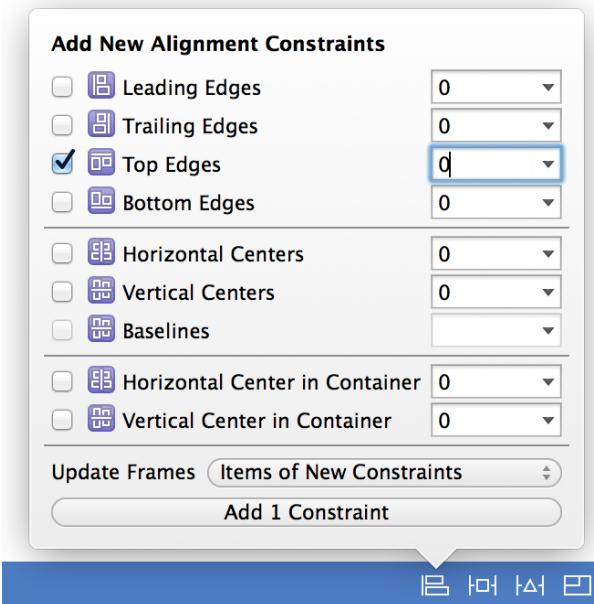
Figure 12.11 Grayed Out Constraint



The top text field will be the standard spacing from the image view. Control-drag from the top text field to the image view and select Horizontal Spacing. With the image view still selected, edit the constant for the newly added constraint in the Size inspector to be the Standard value.

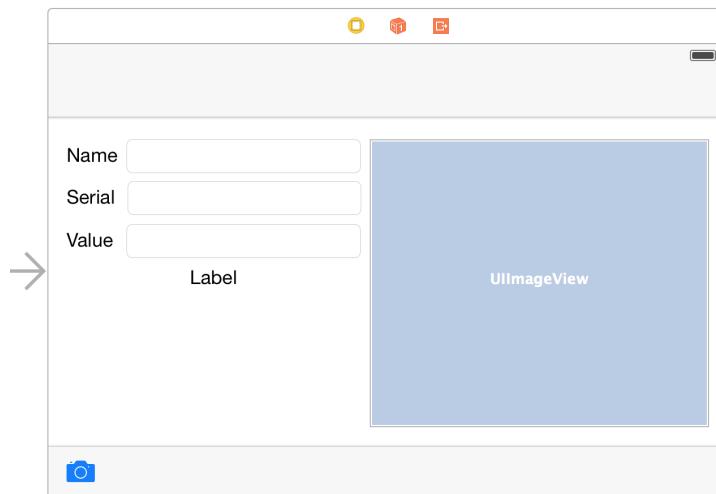
Finally, the image view's top should align with the top of the `nameField`. With the Size inspector still open for the image view, click on the Top Space to: Label constraint and press Delete. Then select both the image view and the top text field in the canvas, open the Align constraint popover, select Top Edges and then Update Frames for Items of New Constraints, and then Add 1 Constraint (Figure 12.12).

Figure 12.12 Aligning the Tops



The interface for the detail view controller is now complete. It should look like Figure 12.13.

Figure 12.13 Completed Detail View Controller Interface



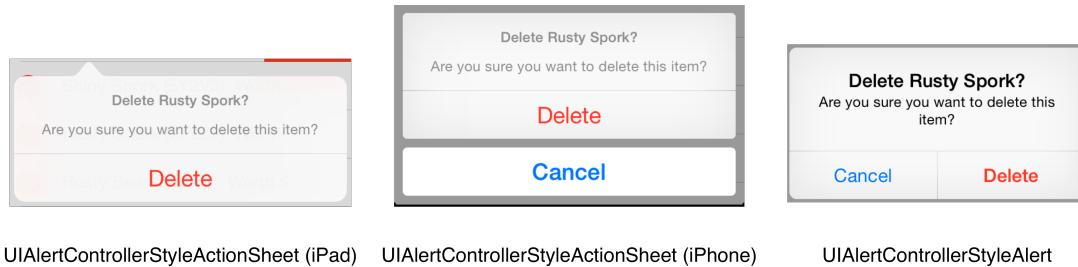
Build and run the application on iPhone. Create an item and drill down to its details to add a photo. Rotate between portrait and landscape and notice how the interface is laid out as you've specified for both regular and compact height.

UIAlertController

UIAlertController is instantiated with a preferred style. The two possible styles are **UIAlertControllerStyleAlert** and **UIAlertControllerStyleActionSheet**, which correspond to the old **UIalertView** and **UIActionSheet**.

Open `ItemsViewController.swift`, and modify **tableView:commitEditingStyle:forRowAtIndexPath:** to use a **UIAlertController** so that the user is forced to confirm the delete action.

Figure 12.14 UIAlertController Styles



```
override func tableView(tableView: UITableView,
    commitEditingStyle editingStyle: UITableViewCellEditingStyle,
    forRowAtIndexPath indexPath: NSIndexPath) {

    // If the table view is asking to commit a delete command...
    if editingStyle == .Delete {
        let item = itemStore.allItems[indexPath.row]

        let title = "Delete \(item.name)?"
        let message = "Are you sure you want to delete this item?"

        let ac = UIAlertController(title: title,
            message: message,
            preferredStyle: .ActionSheet)

        let cancelAction = UIAlertAction(title: "Cancel", style: .Cancel, handler: nil)
        ac.addAction(cancelAction)

        let deleteAction = UIAlertAction(title: "Delete", style: .Destructive,
            handler: { (action) -> Void in
                // Remove the item from the store
                self.itemStore.removeItem(item)

                self.imageStore.deleteImageForKey(item.itemKey)

                // Also remove that row from the table view with an animation
                self.tableView.deleteRowsAtIndexPaths([indexPath], withRowAnimation: .Fade)
            })
        ac.addAction(deleteAction)

        // Use popover style when in regular width environment
        ac.modalPresentationStyle = .Popover

        // Configure popover
        if let cell = tableView.cellForRowAtIndex(indexPath) {
            ac.popoverPresentationController?.sourceView = cell
            ac.popoverPresentationController?.sourceRect = cell.bounds
        }

        // Present the alert controller
        presentViewController(ac, animated: true, completion: nil)
    }
}
```


13

Autorotation, Modal View Controllers, and Popover Controllers

A couple of chapters ago, you used Auto Layout to ensure that Homepwner maintains a standard appearance relative to the device's screen size. For instance, you made sure that the toolbar is always at the bottom and as wide as the screen.

When designing a universal application, you often need the application to behave differently depending on the type of device being used. The different devices have different idioms that users expect, so identical behavior or features on every device would feel foreign at times.

In this chapter, you are going to make four changes to Homepwner's behavior that will tailor the app's behavior to whatever device it is running on.

- On iPads only, allow the interface to rotate in any orientation
- On iPhones only, only allow the interface to be in portrait orientation
- On all devices, present the detail interface modally when the user creates a new item.
- On iPads only, show the image picker in a popover controller when the user presses the camera button.

To do this, you will learn about rotation, more about modal view controllers, and about popover controllers.

Autorotation

There are two distinct orientations in iOS: *device orientation* and *interface orientation*.

The device orientation represents the physical orientation of the device, whether it is right-side up, upside down, rotated left, rotated right, on its face, or on its back. You can access the device orientation through the `UIDevice` class's `orientation` property.

The interface orientation, by contrast, is a property of the running application. The following list shows all of the possible interface orientations:

<code>UIInterfaceOrientation.Portrait</code>	The Home button is below the screen.
<code>UIInterfaceOrientation.PortraitUpsideDown</code>	The Home button is above the screen.
<code>UIInterfaceOrientation.LandscapeLeft</code>	The device is on its side and the Home button is to the right of the screen.
<code>UIInterfaceOrientation.LandscapeRight</code>	The device is on its side and the Home button is to the left of the screen.

When the application's interface orientation changes, the size of the window for the application also changes. The window will take on its new size and will rotate its view hierarchy. The views in the hierarchy will lay themselves out again according to their constraints.

Open `Homepwner.xcodeproj` and build the application on the iPad simulator.

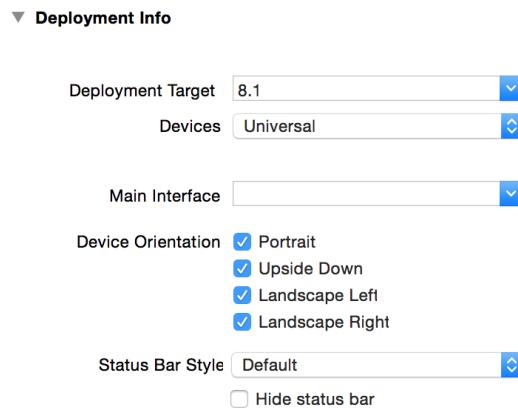
While Homepwner is running on the simulator, you can simulate a rotation. Navigate to the **DetailViewController**. From the simulator's Hardware menu, select Rotate Left option. The simulator window will rotate, causing your "device" to rotate its window and contents. Because you have configured your constraints properly, the interface looks great in all orientations.

When the device orientation changes, your application is informed about the new orientation. Your application can decide whether to allow its interface orientation to match the new orientation of the device.

Rotate to the left again to put the application in portrait upside-down orientation. This time, the interface does not rotate. Even though the device orientation changed, the interface orientation stayed the same because Homepwner does not allow its interface orientation to be set to `.PortraitUpsideDown`. You can change which interface orientations an application supports in the same editor where you universalized Homepwner.

In the Target information's General tab, look at the Device Orientations section. Notice that the portrait and the two landscape options are selected, but Upside Down is not. Click on the Upside Down button to toggle it on (Figure 13.1).

Figure 13.1 Let Homepwner be launched upside down

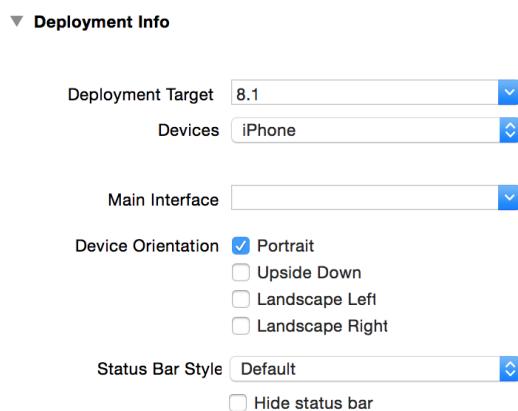


Build and run the application on the iPad simulator and rotate the device in the same direction twice using the **Hardware** menu. Now, the interface will rotate in all directions. It is typical that an iPad application can rotate to all four orientations while an iPhone application can rotate in any orientation other than upside down. Note that the section for iPhone / iPod Deployment Info maintains that this device can only rotate to portrait and the landscape orientations while on the iPhone.

We said one of our goals was to constrain the iPhone application to portrait-only. As of Xcode 6.1, the process to do this is not very intuitive.

Go back to the Target information's General tab, and look at the Devices section. Change it from Universal to iPhone. Then, change the Device Orientations to be just Portrait (Figure 13.2). Switch the Devices to iPad and confirm that all orientations are selected. Finally, switch the Devices back to Universal.

Figure 13.2 Portrait only for iPhone



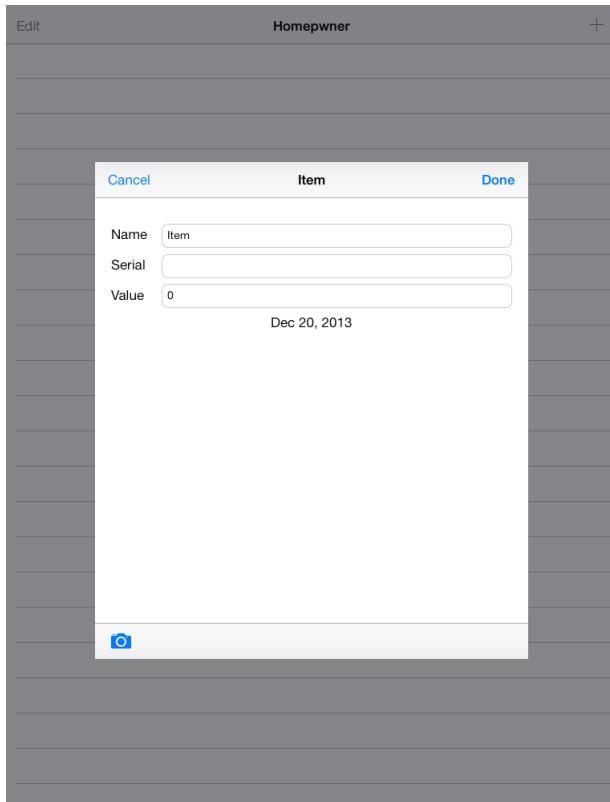
Build and run the application on iPhone. The interface orientation will stay in portrait as you rotate the device. Now build and run on iPad. The application is able to rotate through all orientations.

Some applications will want to lock the user to a specific orientation. For example, many games only allow the two landscape orientations, and many iPhone applications will only allow portrait. Toggling these buttons will allow you to choose which orientations are valid for your application.

Modal View Controllers

In this part of the chapter, you will update Homeowner to present the **DetailViewController** modally when the user creates a new **Item** (Figure 13.3). When the user selects an existing **Item**, the **DetailViewController** will be pushed onto the **UINavigationController**'s stack as before.

Figure 13.3 New item



To implement this dual usage of **DetailViewController**, you will give it a new property, **isNew**. This property will check whether the instance is being used to create a new **Item** or to show an existing one. Then it will configure the interface accordingly.

In **DetailViewController.swift**, declare this property.

```
var isNew: Bool = false
```

If the **DetailViewController** is being used to create a new **Item**, you want it to show a **Done** button and a **Cancel** button on its navigation item. Add a property observer to **isNew** that will set both bar button items appropriately.

```
var isNew: Bool = false {
    didSet {
        if isNew {
            // If this is a new item, provide Cancel and Done items

            let cancelItem = UIBarButtonItem(barButtonSystemItem: .Cancel,
                target: self,
                action: "cancel:")
            navigationItem.leftBarButtonItem = cancelItem

            let doneItem = UIBarButtonItem(barButtonSystemItem: .Done,
                target: self,
                action: "save:")
            navigationItem.rightBarButtonItem = doneItem
        } else {
            // If this is not a new item, use the default items

            navigationItem.leftBarButtonItem = navigationItem.backBarButtonItem
            navigationItem.rightBarButtonItem = nil
        }
    }
}
```

Now that you have your new property in place, let's change what happens when the user adds a new item.

In `ItemsViewController.swift`, edit the `addNewItem(_:)` method to create an instance of `DetailViewController` in a `UINavigationController` and present the navigation controller modally.

```
@IBAction func addNewItem(sender: AnyObject) {

    // Create a new Item and add it to the store
    let newItem = itemStore.createItem()

    // Figure out where that item is in the array
    if let index = find(itemStore.allItems, newItem) {
        let indexPath = NSIndexPath(forRow: index, inSection: 0)
        // Insert this new row into the table.
        tableView.insertRowsAtIndexPaths([indexPath], withRowAnimation: .Top)
    }

    let dvc = DetailViewController(item: newItem,
        itemStore: itemStore,
        imageStore: imageStore)
    dvc.isNew = true

    let nc = UINavigationController(rootViewController: dvc)
    presentViewController(nc, animated: true, completion: nil)
}
```

Build and run the application and tap the New button to create a new item. An instance of `DetailViewController` will slide up from the bottom of the screen with a Done button and a Cancel button on its navigation item. (Tapping these buttons, of course, will throw an exception, since you have not implemented the action methods yet.)

Notice that you are creating an instance of `UINavigationController` that will never be used for navigation. This gives this view the same title bar across the top that every other view has. It also gives you a place to put the Done and Cancel buttons.

Dismissing modal view controllers

To dismiss a modally-presented view controller, you must send the message `dismissViewControllerAnimated(_:completion:)` to the view controller that presented it. You have done this before with `UIImagePickerController` – the `DetailViewController` presented it, and when the image picker told the `DetailViewController` it was done, the `DetailViewController` dismissed it.

Now, you have a slightly different situation. When a new item is created, the `ItemsViewController` presents the `DetailViewController` modally. The `DetailViewController` has two buttons on its `navigationItem`

that will dismiss it when tapped: Cancel and Done. There is a problem here: the action messages for these buttons are sent to the **DetailViewController**, but it is the responsibility of the **ItemsViewController** to do the dismissing. The **DetailViewController** needs a way to tell the view controller that presented it, “Hey, I’m done, you can dismiss me now.”

To solve this problem, you will add two closure properties to **DetailViewController**: one that will be called if the user cancels and another that will be called if the user saves.

Open **DetailViewController.swift** and add these two properties.

```
var cancelClosure: (() -> ())?
var saveClosure: (() -> ())?
```

Each of these properties is for an optional closure that takes in no arguments and returns nothing.

In **DetailViewController.swift**, implement the action methods for both the Done and Cancel buttons.

```
func save(sender: AnyObject) {
    saveClosure?()
}

func cancel(sender: AnyObject) {
    cancelClosure?()
}
```

Now the **ItemsViewController** can configure some code to be executed when either of the buttons are tapped.

Open **ItemsViewController.swift** and modify **addNewItem(_:)** to set the **saveClosure** and **cancelClosure** on the **DetailViewController**.

```
@IBAction func addNewItem(sender: AnyObject) {
    // Create a new Item and add it to the store
    let newItem = itemStore.createItem()

    let dvc = DetailViewController(item: newItem,
        itemStore: itemStore,
        imageStore: imageStore)
    dvc.isNew = true

    dvc.cancelClosure = {
        // Remove the new Item from the store
        self.itemStore.removeItem(newItem)

        self.dismissViewControllerAnimated(true, completion: nil)
    }

    dvc.saveClosure = {
        // Figure out where that item is in the array
        if let index = find(self.itemStore.allItems, newItem) {
            let indexPath = NSIndexPath(forRow: index, inSection: 0)
            // Insert this new row into the table.
            self.tableView.insertRowsAtIndexPaths([indexPath], withRowAnimation: .Top)
        }

        self.dismissViewControllerAnimated(true, completion: nil)
    }

    let nc = UINavigationController(rootViewController: dvc)
    presentViewController(nc, animated: true, completion: nil)
}
```

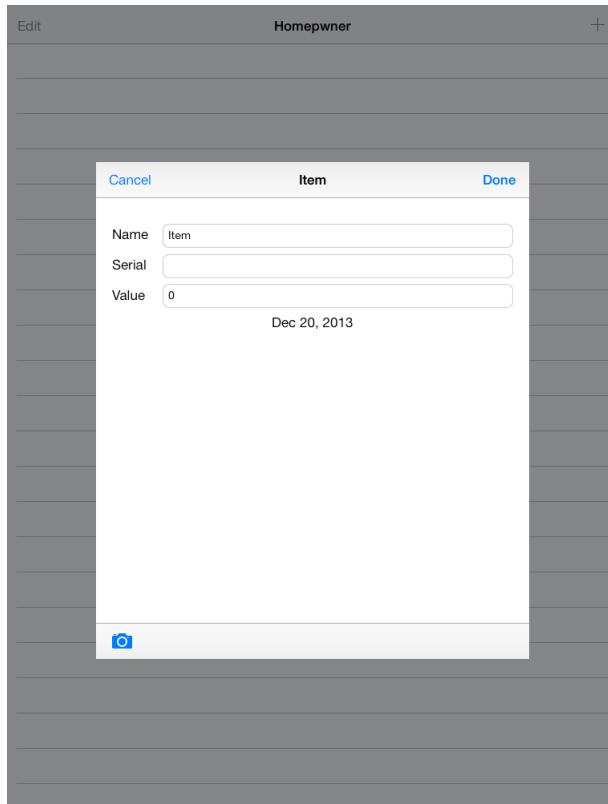
Build and run the application. Create a new item and tap the Cancel button. The instance of **DetailViewController** will slide off the screen, and nothing will be added to the table view. Then, create a new item and tap the Done button. The **DetailViewController** will slide off the screen, and your new **Item** will appear in the table view.

Modal view controller styles

On the iPhone or iPod touch, a modal view controller takes over the entire screen. This is the default behavior and the only possibility on these devices. On the iPad, you have two additional options: a form sheet style and a page sheet style. You can change the presentation of the modal view controller by setting its `modalPresentationStyle` property to a pre-defined constant – `UIModalPresentationStyle.FormSheet` or `UIModalPresentationStyle.PageSheet`.

The form sheet style shows the modal view controller's view in a rectangle in the center of the iPad's screen and dims out the presenting view controller's view (Figure 13.4).

Figure 13.4 An example of the form sheet style



The page sheet style is the same as the default full-screen style in portrait mode. In landscape mode, it keeps its width the same as in portrait mode and dims the left and right edges of the presenting view controller's view that stick out behind it.

In `ItemsViewController.swift`, modify the `addNewItem(_:)` method to change the presentation style of the `UINavigationController` that is being presented.

```
let nc = UINavigationController(rootViewController: dvc)
nc.modalPresentationStyle = UIModalPresentationStyle.FormSheet
presentViewController(nc, animated: true, completion: nil)
```

Notice that you change the presentation style of the `UINavigationController`, not the `DetailViewController`, since it is the one that is being presented modally.

Build and run the application on the iPad simulator or on an iPad. Tap the button to add a new item and watch the modal view controller slide onto the screen. Add some item details and then tap the Done button. The table view reappears, but your new `Item` is not there. What happened?

Before you changed its presentation style, the modal view controller took up the entire screen, which caused the view of the **ItemsViewController** to disappear. When the modal view controller was dismissed, the methods **viewWillAppear(_)** and **viewDidAppear(_)** were called on the **ItemsViewController** and it took this opportunity to reload its table to catch any updates to the **ItemStore**.

With the new presentation style, the **ItemsViewController**'s view does not disappear when it presents the view controller. So it is not sent the appearance messages when the modal view controller is dismissed, and it does not get the chance to reload its table view.

You have to find another opportunity to reload the data. The code for the **ItemsViewController** to reload its table view is simple. It looks like this:

```
tableView.reloadData()
```

What you need to do is to package up this code and have it executed right when the modal view controller is dismissed. Fortunately, there is a built-in mechanism in **dismissViewControllerAnimated:completion:** that you can use to accomplish this.

Modal view controller transitions

In addition to changing the presentation style of a modal view controller, you can change the animation that places it on screen. Like presentation styles, there is a view controller property (**modalTransitionStyle**) that you can set with a pre-defined constant. By default, the animation will slide the modal view controller up from the bottom of the screen. You can also have the view controller fade in, flip in, or appear underneath a page curl.

The various transition styles are:

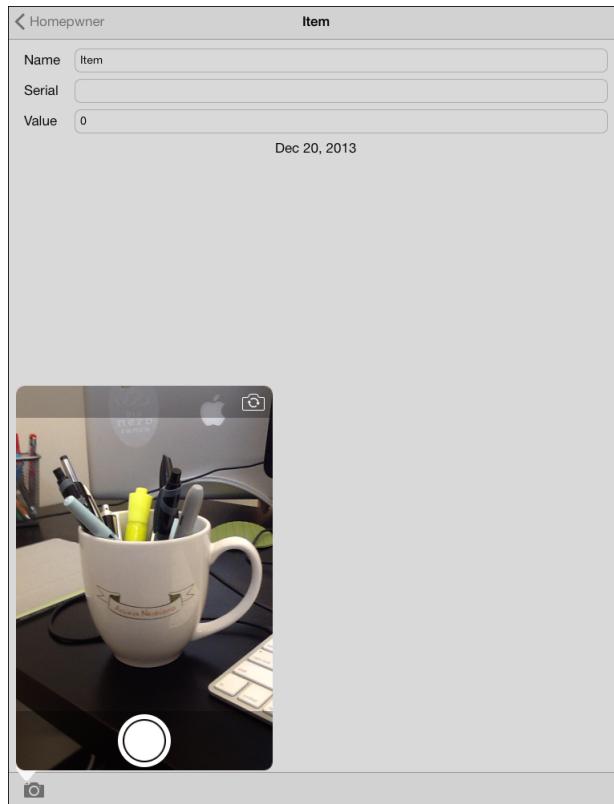
<code>UIModalTransitionStyle.CoverVertical</code>	slides up from the bottom
<code>UIModalTransitionStyle.CrossDissolve</code>	fades in
<code>UIModalTransitionStyle.FlipHorizontal</code>	flips in with a 3D effect
<code>UIModalTransitionStyle.PartialCurl</code>	presenting view controller is peeled up revealing the modal view controller

UIPopoverController

With iPad applications, you have a lot more screen space to work with. Let's take advantage of this by presenting the **UIImagePickerController** in a *popover* when the user taps the camera button in the detail interface.

A popover displays another view controller's view in a bordered window that floats above the rest of the application's interface. It is only available on iPads. When you create a **UIPopoverController**, you set this other view controller as the popover controller's **contentViewController**. Popover controllers are useful when giving the user a list of choices (like picking a photo out of the photo library) or some extra information about something that is summarized on the screen. For example, a form may have some buttons next to some of the fields. Tapping on the button would reveal a popover whose **contentViewController** explains the intended use of that field.

In this section, you will present the **UIImagePickerController** in a popover when the user taps the camera bar button item in the **DetailViewController**'s view (Figure 13.5).

Figure 13.5 **UIPopoverController**

To present a view controller in a popover, use the `.Popover modalPresentationStyle`.

In `DetailViewController.swift`, add the following code to the end of `takePicture(_:)`.

```
@IBAction func takePicture(sender: AnyObject) {
    let imagePickerController = UIImagePickerController()

    // If the device has a camera, take a picture, otherwise,
    // just pick from photo library
    if UIImagePickerController.isSourceTypeAvailable(.Camera) {
        imagePickerController.sourceType = .Camera
    }
    else {
        imagePickerController.sourceType = .PhotoLibrary
    }

    imagePickerController.delegate = self
    imagePickerController.modalPresentationStyle = UIModalPresentationStyle.Popover
    presentViewController(imagePickerController, animated: true, completion: nil)
}
```

Build and run the application on the iPad simulator or on an iPad. Navigate to the `DetailViewController` and tap the camera icon. Boom! The application crashes:

```
'UIPopoverPresentationController (<UIPopoverPresentationController: 0x7fac8a5cca20>) should
have a non-nil sourceView or barButtonItem set before the presentation occurs.'
```

A popover must be told where it should point to on the screen. There are two ways you can do this. Either tell the popover to point to a specific rectangle in a specific view (by setting a `sourceView` and `sourceRect`), or by giving it a `UIBarButtonItem` to point at. The image picker controller popover should point at the Camera bar button item, so we will set the `barButtonItem` property to be the Camera item.

So how do you configure the popover? Every `UIViewController` has a `popoverPresentationController` property. When that view controller is presented in a popover, the attributes on the `popoverPresentationController` determine how the popover is configured.

Update `takePicture(_:)` to tell the popover that it should point at the Camera bar button item.

```
@IBAction func takePicture(sender: AnyObject) {
    let imagePicker = UIImagePickerController()

    // If the device has a camera, take a picture, otherwise,
    // just pick from photo library
    if UIImagePickerController.isSourceTypeAvailable(.Camera) {
        imagePicker.sourceType = .Camera
    }
    else {
        imagePicker.sourceType = .PhotoLibrary
    }

    imagePicker.delegate = self
    imagePicker.modalPresentationStyle = UIModalPresentationStyle.Popover
    imagePicker.popoverPresentationController?.barButtonItem = sender as UIBarButtonItem

    presentViewController(imagePicker, animated: true, completion: nil)
}
```

Build and run the application on iPad. Tapping the Camera button will now display the `UIImagePickerController` in an iPad. Dismiss the popover by either tapping the Camera button again, or tapping anywhere outside of the popover.

Now build and run the application on iPhone. The popover only appears in an environment with a regular horizontal size class, so the `UIImagePickerController` will still be presented full screen on an iPhone.

Gold Challenge: Popover Appearance

You can change the appearance of a `UIPopoverController`. Do this for the popover that presents the `UIImagePickerController`. (Hint: check out the `popoverBackgroundViewClass` property in `UIPopoverPresentationController`.)

For the More Curious: View Controller Relationships

The relationships between view controllers are important for understanding where and how a view controller's view appears on the screen. Overall, there are two different types of relationships between view controllers: *parent-child* relationships and *presenting-presentee* relationships. Let's look at each one individually.

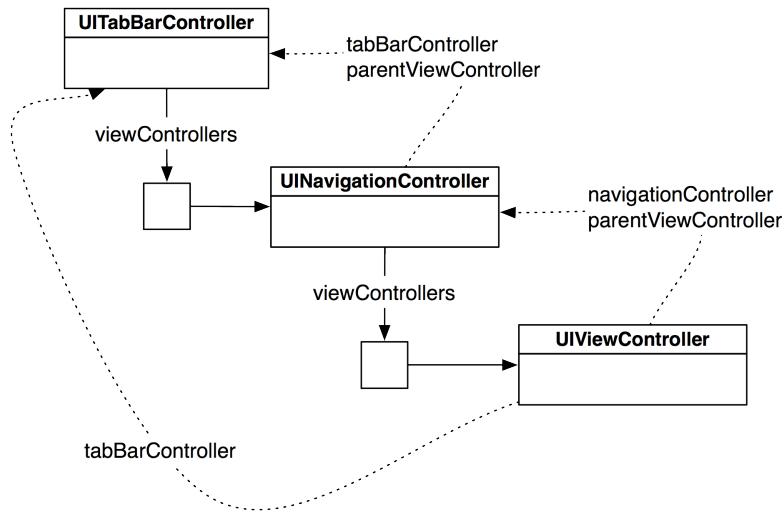
Parent-child relationships

Parent-child relationships are formed when using *view controller containers*. Examples of view controller containers are `UINavigationController`, `UITabBarController`, and `UISplitViewController` (which you will see in Chapter 19). You can identify a view controller container because it has a `viewControllers` property that is an array of the view controllers it contains.

A view controller container is always a subclass of `UIViewController` and thus has a `view`. The behavior of a view controller container is that it selectively adds the views of its `viewControllers` as subviews of its own `view`. A container has its own built-in interface, too. For example, a `UINavigationController`'s `view` shows a navigation bar and the `view` of its `topViewController`.

View controllers in a parent-child relationship form a *family*. So, a `UINavigationController` and its `viewControllers` are in the same family. A family can have multiple levels. For example, imagine a situation where a `UITabBarController` contains a `UINavigationController` that contains a `UIViewController`. These three view controllers are in the same family (Figure 13.6). The container classes have access to their children through the `viewControllers` array, and the children have access to their ancestors through four properties of `UIViewController`.

Figure 13.6 A view controller family



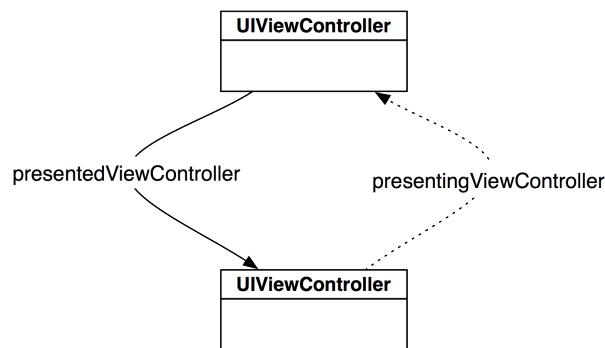
Every **UIViewController** has a **parentViewController** property. This property holds the closest view controller ancestor in the family. Thus, it could return a **UINavigationController**, **UITabBarController**, or a **UISplitViewController** depending on the makeup of the family tree.

The ancestor-access methods of **UIViewController** include **navigationController**, **tabBarController**, and **splitViewController**. When a view controller is sent one of these messages, it searches up the family tree (using the **parentViewController** property) until it finds the appropriate type of view controller container. If there is no ancestor of the appropriate type, these methods return **nil**.

Presenting-presenter relationships

The other kind of relationship is a presenting-presenter relationship, which occurs when a view controller is presented modally. When a view controller is presented modally, its view is added *on top of* the view controller's view that presented it. This is different than a view controller container, which intentionally keeps a spot open on its interface to swap in the views of the view controllers it contains. Any **UIViewController** can present another view controller modally.

Figure 13.7 Presenting-presenter relationship



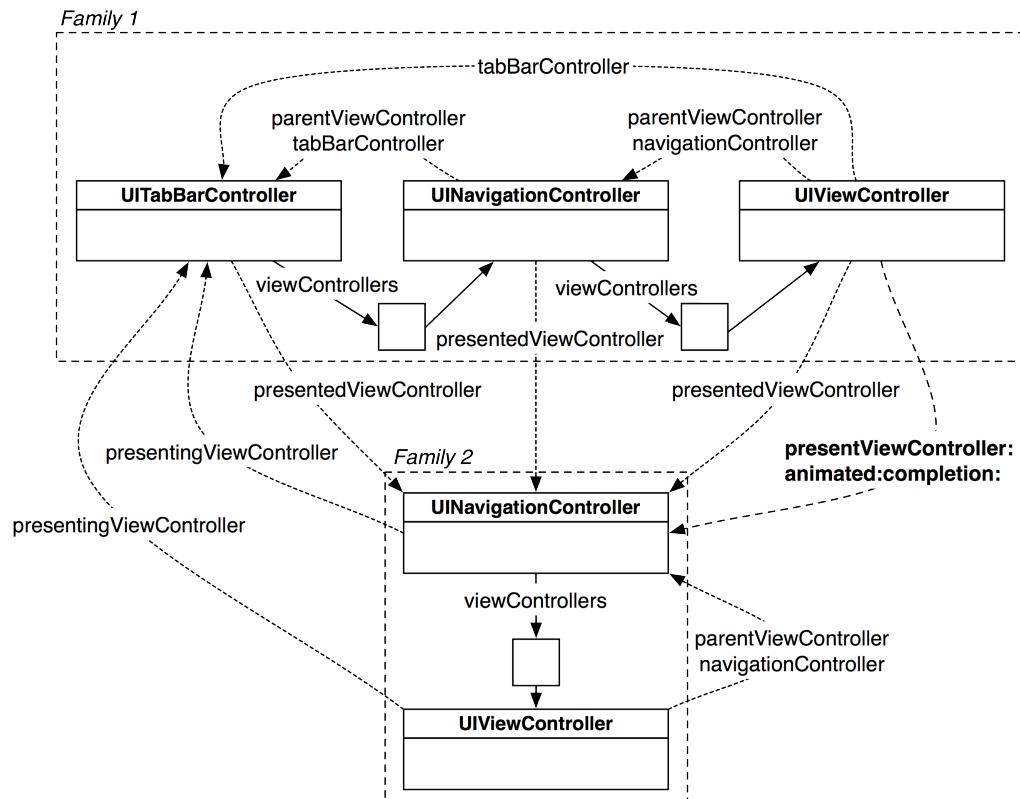
There are two built-in properties for managing the relationship between presenter and presentee. A modally-presented view controller's **presentingViewController** will point back to the view controller that presented it, while the presenter will keep a pointer to the presentee in its **presentedViewController** property (Figure 13.7).

Inter-family relationships

A presented view controller and its presenter are *not* in the same view controller family. Instead, the presented view controller has its own family. Sometimes, this family is just one **UIViewController**; other times, this family is made up of multiple view controllers.

Understanding the difference in families will help you understand the values of properties like `presentedViewController` and `navigationController`. Consider the view controllers in Figure 13.8. There are two families, each with multiple view controllers. This diagram shows the values of the view controller relationship properties.

Figure 13.8 A view controller hierarchy



First, notice that the properties for parent-child relationships can never cross over family boundaries. Thus, sending `tabBarController` to a view controller in Family 2 will not return the `UITabBarController` in Family 1; it will return `nil`. Likewise, sending `navigationController` to the view controller in Family 2 returns its `UINavigationViewController` parent in Family 2 and not the `UINavigationViewController` in Family 1.

Perhaps the oddest view controller relationships are the ones between families. When a view controller is presented modally, the actual presenter is the oldest member of the presenting family. For example, in Figure 13.8, the `UITabBarController` is the `presentingViewController` for the view controllers in Family 2. It does not matter which view controller in Family 1 was sent `presentViewController:animated:completion:`, the `UITabBarController` is always the presenter.

This behavior explains why the `DetailViewController` obscures the `UINavigationBar` when presented modally but not when presented normally in the `UINavigationController`'s stack. Even though the `ItemsViewController` is told to do the modal presenting, its oldest ancestor, the `UINavigationController`, actually carries out the task. The `DetailViewController` is put on top of the `UINavigationController`'s view and thus obscures the `UINavigationBar`.

Notice also that the `presentingViewController` and `presentedViewController` are valid for every view controller in each family and always point to the oldest ancestor in the other family.

14

Core Data

When deciding between approaches to saving and loading for iOS applications, the first question is typically “Local or remote?” If you want to save data to a remote server, this is typically done with a web service. Let’s assume that you want to store data locally. The next question is typically “Archiving or Core Data?”

At the moment, Homepwner uses keyed archiving to save item data to the filesystem. The biggest drawback to archiving is its all-or-nothing nature: to access anything in the archive, you must unarchive the entire file; to save any changes, you must rewrite the entire file. Core Data, on the other hand, can fetch a small subset of the stored objects. And if you change any of those objects, you can update just that part of the file. This incremental fetching, updating, deleting, and inserting can radically improve the performance of your application when you have a lot of model objects being shuttled between the filesystem and RAM.

Object-Relational Mapping

Core Data is a framework that provides *object-relational mapping*. In other words, Core Data can turn objects into data that is stored in a SQLite database file and vice-versa. SQLite is a relational database that is stored in a single file. (Technically, SQLite is the library that manages the database file, but we use the word to mean both the file and the library.) It is important to note that SQLite is not a full-fledged relational database server like Oracle, MySQL, or SQLServer, which are their own applications that clients can connect to over a network.

Core Data gives us the ability to fetch and store data in a relational database without having to know SQL. However, you do have to understand a bit about how relational databases work. This chapter will give you that understanding as you replace keyed archiving with Core Data in Homepwner’s **ItemStore**.

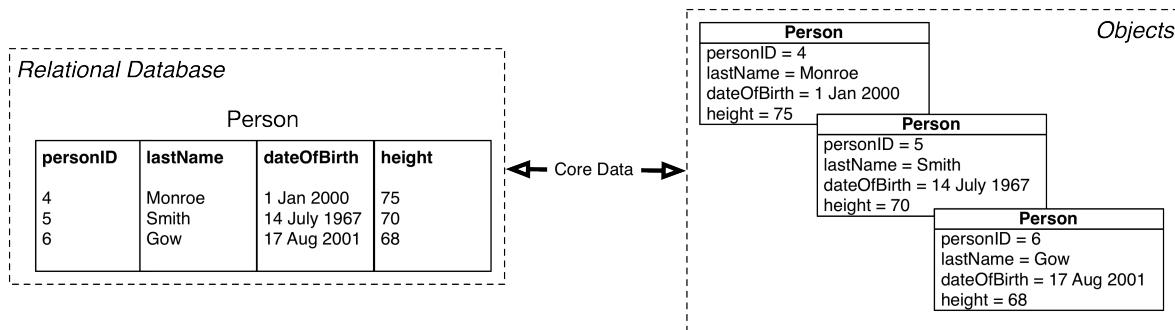
Moving Homepwner to Core Data

Your Homepwner application currently uses archiving to save and reload its data. For a moderately sized object model (say, fewer than 1000 objects), this is fine. As your object model gets larger, however, you will want to be able to do incremental fetches and updates, and Core Data can do this.

The model file

In a relational database, we have something called a *table*. A table represents some type; you can have a table of people, a table of a credit card purchases, or a table of real-estate listings. Each table has a number of columns to hold pieces of information about that thing. A table that represents people might have columns for the person’s last name, date of birth, and height. Every row in the table represents a single person.

Figure 14.1 Role of Core Data

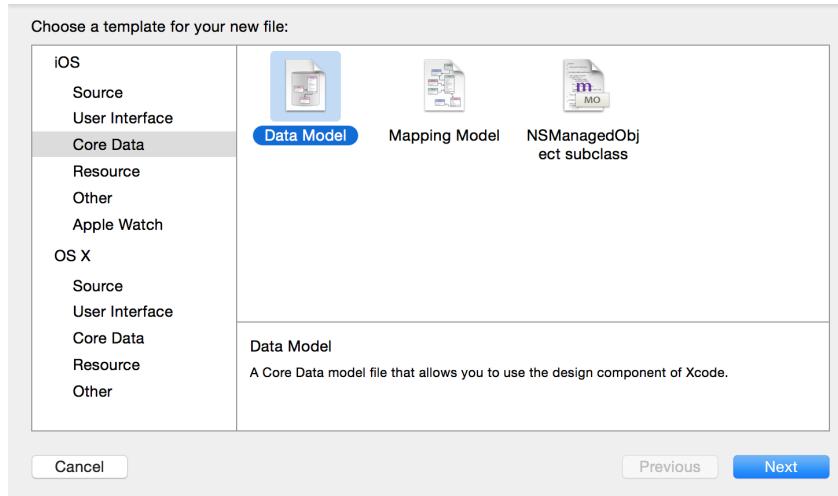


This organization translates well to Swift. Every table is like a Swift class. Every column is one of the class's properties. Every row is an instance of that class. Thus, Core Data's job is to move data to and from these two representations (Figure 14.1).

Core Data uses different terminology to describe these ideas: a table/class is called a *entity*, and the columns/properties are called *attributes*. A Core Data model file is the description of every entity along with its attributes in your application. In Homepwner, you are going to describe a **Item** entity in a model file and give it attributes like `itemName`, `serialNumber`, and `valueInDollars`.

Open `Homepwner.xcodeproj`. From the File menu, create a new file. Select Core Data in the iOS section and create a new Data Model. Name it `Homepwner`.

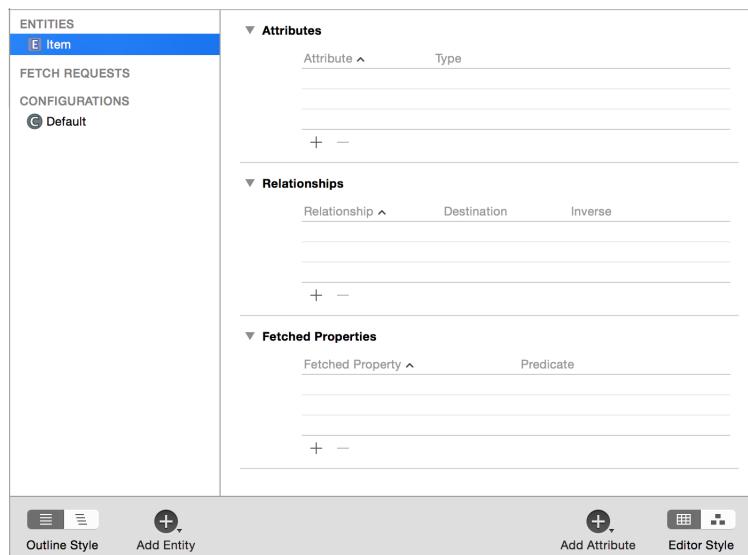
Figure 14.2 Create the model file



This will create a `Homepwner.xcdatamodeld` file and add it to your project. Select this file from the project navigator, and the editor area will reveal the user interface for manipulating a Core Data model file.

Find the Add Entity button at the bottom left of the window and click it. A new Entity will appear in the list of entities in the lefthand table. Double-click this entity and change its name to **Item** (Figure 14.3).

Figure 14.3 Create the **Item** entity



Now your **Item** entity needs attributes. Remember that these will be the properties of the **Item** class. The necessary attributes are listed below. For each attribute, click the + button in the Attributes section and edit the Attribute and Type values:

- `name` is a String
- `serialNumber` is a String
- `valueInDollars` is an Integer 32
- `dateCreated` is a Date
- `itemKey` is a String

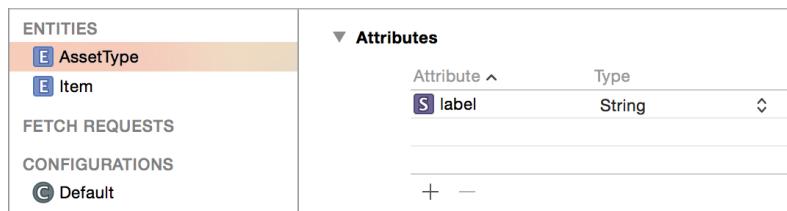
There is one more attribute to add. In `Homepwner`, users can order items by changing their positions in the table view. Archiving items in an array naturally respects this order. However, relational tables do not order their rows. Instead, when you fetch a set of rows, you specify their order using one of the attributes (Fetch me all the `Employee` objects ordered by `lastName`.).

To maintain the order of items, you need to create an attribute to record each item's position in the table view. Then when you fetch items, you can ask for them to be ordered by this attribute. (You will also need to update that attribute when the items are reordered.) Create this final attribute: name it `orderingValue` and make it a Double.

At this point, your model file is sufficient to save and load items. However, one of the benefits to using Core Data is that entities can be related to one another, so you are going to add a new entity called `AssetType` that describes a category of items. For example, a painting might be of the `Art` asset type. `AssetType` will be an entity in the model file, and each row of that table will be mapped to a Swift object at runtime.

In `Homepwner.xcdatamodeld`, add another entity called `AssetType`. Give it an attribute called `label` of type String. This will be the name of the category the `AssetType` represents.

Figure 14.4 Create the `AssetType` entity



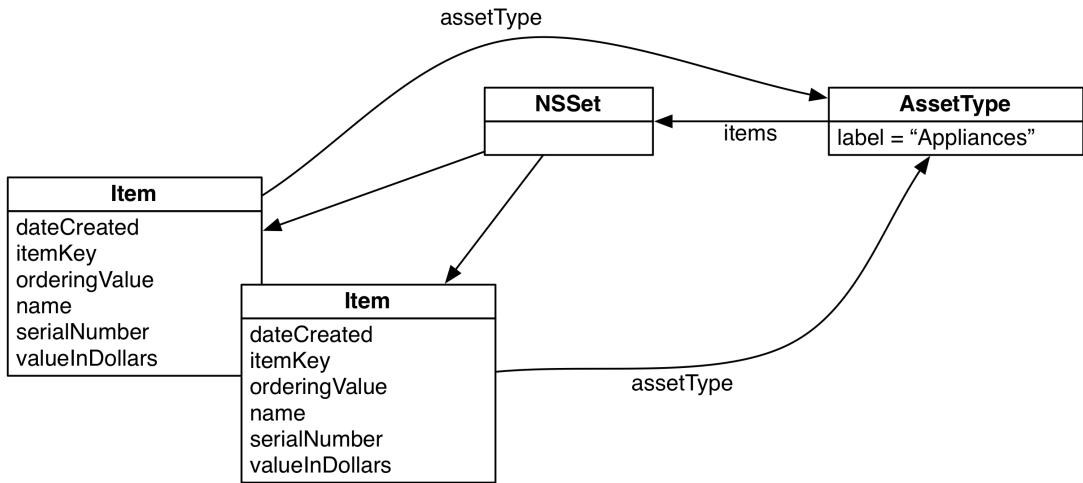
Now you need to establish relationships between `AssetType` and `Item`. Relationships between entities are represented by pointers between objects. There are two kinds of relationships: *to-one* and *to-many*.

When an entity has a to-one relationship, each instance of that entity will have a pointer to an instance in the entity it has a relationship to. The `Item` entity will have a to-one relationship to the `AssetType` entity. Thus, a `Item` instance will have a pointer to a `AssetType` instance.

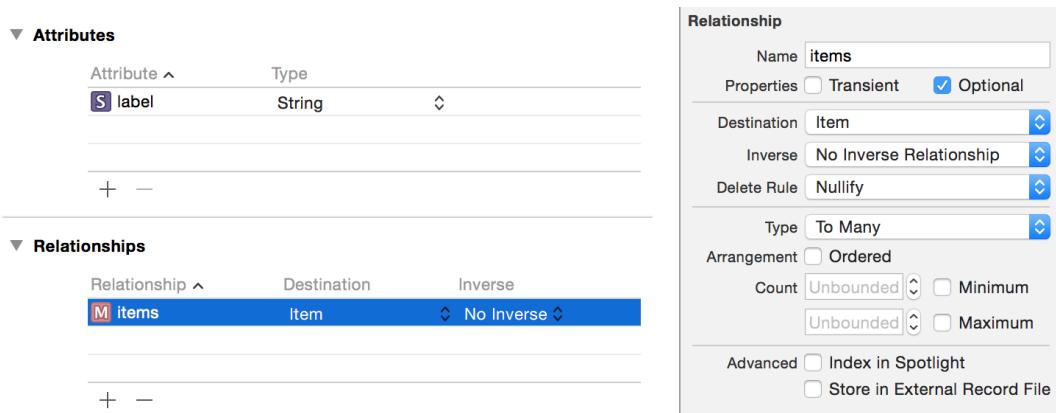
When an entity has a to-many relationship, each instance of that entity has a pointer to an `NSSet`. This set contains the instances of the entity that it has a relationship with. The `AssetType` entity will have a to-many relationship to the `Item` entity because many instances of `Item` can have the same `AssetType`. Thus, a `AssetType` object will have a pointer to a set of all of the `Item` objects that are its type of asset.

With these relationships set up, you can ask a `AssetType` object for the set of `Item` objects that fall into its category, and you can ask a `Item` which `AssetType` it falls under. Figure 14.5 diagrams the relationships between `AssetType` and `Item`.

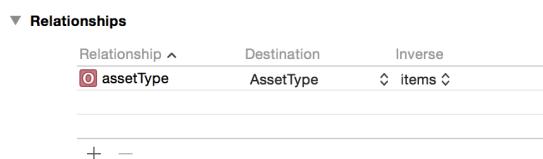
Figure 14.5 Entities in Homeowner



Let's add these relationships to the model file. Select the **AssetType** entity and then click the **+** button in the Relationships section. Name this relationship **items** in the Relationship column. Then, select **Item** from the Destination column. In the data model inspector, change the Type drop-down from **To One** to **To Many** (Figure 14.6).

Figure 14.6 Create the **items** relationship

Now go back to the **Item** entity. Add a relationship named **assetType** and pick **AssetType** as its destination. In the **Inverse** column, select **items** (Figure 14.7).

Figure 14.7 Create the **assetType** relationship

NSManagedObject and subclasses

When an object is fetched with Core Data, its class, by default, is **NSManagedObject**. **NSManagedObject** is a subclass of **NSObject** that knows how to cooperate with the rest of Core Data. An **NSManagedObject** works a bit like a dictionary: it holds a key-value pair for every property (attribute or relationship) in the entity.

An **NSManagedObject** is little more than a data container. If you need your model objects to *do* something in addition to holding data, you must subclass **NSManagedObject**. Then, in your model file, you specify that this entity is represented by instances of your subclass, not the standard **NSManagedObject**.

There is one problem: the **Item** class already exists, and it does not inherit from **NSManagedObject**. Changing the superclass of the existing **Item** to **NSManagedObject** will require considerable modifications. Thus, the easiest solution is to remove your current **Item** class files, have Core Data generate a new **Item** class, and then add your behavior methods back to the new class files.

In Finder, drag **Item.swift** to your desktop for safekeeping. Then, in Xcode, delete this file from the project navigator. (It will appear in red after you have moved the file).

Now, open **Homepwner.xcdatamodeld** again and select the **Item** entity. Then, select File New File....

From the iOS section, select Core Data, choose the **NSManagedObject subclass** option, and click Next. The checkbox for the Homepwner data model should already be checked. If it is not checked, go ahead and check the box, and then click Next. On the following screen, make sure that the **Item** is checked, and then click Next one last time. Finally, make sure the Language is Swift and click Create to generate the **NSManagedObject** subclass files.

Xcode will generate a new **Item.swift** file. Open **Item.swift** and see what Core Data has wrought. By default, Xcode generates the properties as objects, so your ints are now instances of **NSNumber**. Change **orderingValue** to be a **Double** and **valueInDollars** to be an **Int**.

```
import Foundation
import CoreData

class Item: NSManagedObject {

    @NSManaged var name: String
    @NSManaged var serialNumber: String?
    @NSManaged var valueInDollars: Int
    @NSManaged var dateCreated: NSDate
    @NSManaged var itemKey: String
    @NSManaged var orderingValue: Double
    @NSManaged var assetType: NSManagedObject

}
```

Of course, when you first launch an application, there are no saved items or asset types. When the user creates a new **Item** instance, it will be added to the database. When objects are added to the database, they are sent the message **awakeFromInsert**. Here is where you will set the **dateCreated** and **itemKey** properties of a **Item**. Implement **awakeFromInsert** in **Item.swift**.

```
override func awakeFromInsert() {
    super.awakeFromInsert()

    dateCreated = NSDate()

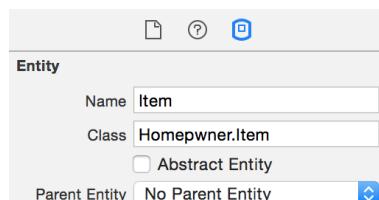
    // Create an NSUUID object and get its string representation
    let uuid = NSUUID()
    let key = uuid.UUIDString
    itemKey = key
}
```

This adds the extra behavior of **Item**'s old designated initializer.

When a new **Item** entity is created, Core Data needs to know that the type of this object is **Item** instead of the default **NSManagedObject**.

Open **Homepwner.xcdatamodeld** and select the **Item** entity. Show the data model inspector and change the Class field to **Homepwner.Item**, as shown in Figure 14.8.

Figure 14.8 Changing the class of an entity



Why `Homeowner.Item` instead of `Item`? Core Data need to be able to support both Swift classes and Objective-C classes. In Swift, every type name consists of two elements: the module name and the type name, separated by a period. The module name for us is the project name. So the full name of the `Item` class is actually `Homeowner.Item`. In fact, everywhere in code that you refer to the `Item` class, you could replace it with `Homeowner.Item`.

```
// For example
let item = Item()
```

```
// Could be rewritten as
let item = Homeowner.Item()
```

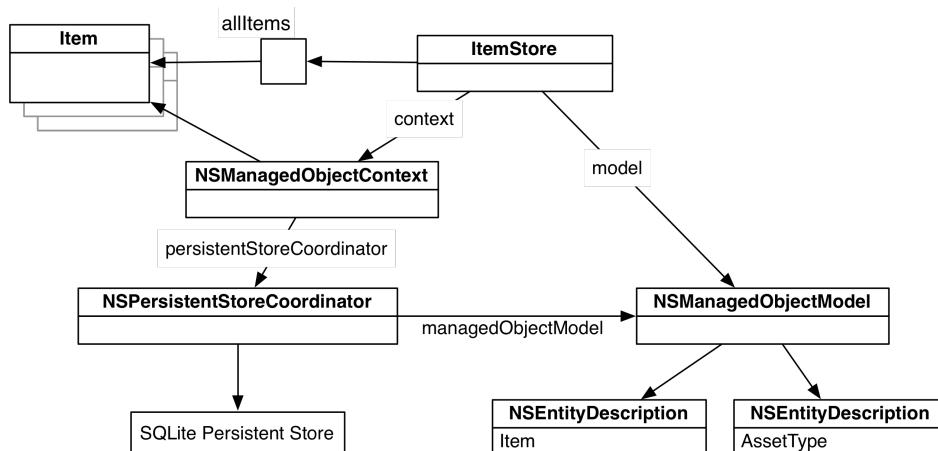
In practice, the only time you would need to do this is if you have multiple modules in your project that have types with identical names. You'd need to use the fully qualified name to disambiguate which type you are referring to.

Now, when a `Item` entity is fetched with Core Data, the type of this object will be `Item`. (`AssetType` instances will still be of type `NSManagedObject`.)

Updating ItemStore

The portal through which you talk to the database is the `NSManagedObjectContext`. The `NSManagedObjectContext` uses an `NSPersistentStoreCoordinator`. You ask the persistent store coordinator to open a SQLite database at a particular filename. The persistent store coordinator uses the model file in the form of an instance of `NSManagedObjectModel`. In `Homeowner`, these objects will work with the `ItemStore`. These relationships are shown in Figure 14.9.

Figure 14.9 `ItemStore` and `NSManagedObjectContext`



In `ItemStore.swift`, import Core Data and add three properties.

```
import UIKit
import CoreData

class ItemStore: NSManagedObject {
    var allItems: [Item] = []
    var allAssetTypes: [NSManagedObject] = []
    var context: NSManagedObjectContext!
    var model: NSManagedObjectModel!
```

Then change the implementation of `itemArchivePath` to return a different path that Core Data will use to save data.

```
let itemArchivePath: String = {
    let documentsDirectories =
        NSSearchPathForDirectoriesInDomains(.DocumentDirectory, .UserDomainMask, true)
    let documentDirectory = documentsDirectories.first as String
    return documentDirectory.stringByAppendingPathComponent("items.archive")
    return documentDirectory.stringByAppendingPathComponent("store.data")
}()
```

When the **ItemStore** is initialized, it needs to set up the **NSManagedObjectContext** and an **NSPersistentStoreCoordinator**. The persistent store coordinator needs to know two things: “What are all of my entities and their attributes and relationships?” and “Where am I saving and loading data from?” To answer these questions, you need to create an instance of **NSManagedObjectModel** to hold the entity information of `Homepwner.xcdatamodeld` and initialize the persistent store coordinator with this object. Then, you will create the instance of **NSManagedObjectContext** and specify that it use this persistent store coordinator to save and load objects.

In `ItemStore.swift`, update `init()` to set up the Core Data properties.

```
override init() {
    super.init()

    if let archivedItems = NSKeyedUnarchiver.unarchiveObjectWithFile(itemArchivePath) as? [Item] {
        allItems += archivedItems
    }

    if let model = NSManagedObjectModel.mergedModelFromBundles(nil) {
        self.model = model

        let psc = NSPersistentStoreCoordinator(managedObjectContext: model)

        if let storeURL = NSURL(fileURLWithPath: itemArchivePath) {
            var error: NSError?

            psc.addPersistentStoreWithType(NSSQLiteStoreType,
                configuration: nil,
                URL: storeURL,
                options: nil,
                error: &error)

            context = NSManagedObjectContext()
            context.persistentStoreCoordinator = psc
        }
    }

    let nc = NSNotificationCenter.defaultCenter()
    nc.addObserver(self,
        selector: "appDidEnterBackground:",
        name: UIApplicationDidEnterBackgroundNotification,
        object: nil)
}
```

Before, **ItemStore** would write out the entire **Array** of **Item** objects when you asked it to save using keyed archiving. Now, you will have it send the message `save(_:)` to the **NSManagedObjectContext**. The context will update all of the records in `store.data` with any changes since the last time it was saved. In `ItemStore.swift`, change `saveChanges`.

```
func saveChanges() -> Bool {
    return NSKeyedArchiver.archiveRootObject(allItems, toFile: itemArchivePath)

    var error: NSError?
    let success = context.save(&error)
    if !success {
        println("Error saving: \(error!.localizedDescription)")
    }
    return success
}
```

Recall that this method is already called when the application is moved to the background.

NSFetchRequest and NSPredicate

In this application, you will fetch all of the items in `store.data` the first time you need them. To get objects back from the **NSManagedObjectContext**, you must prepare and execute an **NSFetchRequest**. After a fetch request is executed, you will get an array of all the objects that match the parameters of that request.

A fetch request needs an entity description that defines which entity you want to get objects from. To fetch **Item** instances, you specify the **Item** entity. You can also set the request’s *sort descriptors* to specify the order of the

objects in the array. A sort descriptor has a key that maps to an attribute of the entity and a BOOL that indicates if the order should be ascending or descending. You want to sort the returned instances of **Item** by `orderingValue` in ascending order.

In `ItemStore.swift`, define a new method, `loadAllItems`, to prepare and execute the fetch request and save the results into the `allItems` array.

```
func loadAllItems() {
    let request = NSFetchedResultsController()

    let e = NSEntityDescription.entityForName("Item",
                                              inManagedObjectContext: context)
    request.entity = e

    let sd = NSSortDescriptor(key: "orderingValue", ascending: true)
    request.sortDescriptors = [sd]

    var error: NSError?
    if let result = context.executeFetchRequest(request,
                                                error: &error) as? [Item] {
        allItems = allItems + result
    } else {
        println("Fetch failed: \(error!.localizedDescription)")
    }
}
```

Also in `ItemStore.swift`, call this method at the end of `init()`.

```
    context = NSManagedObjectContext()
    context.persistentStoreCoordinator = psc

    self.loadAllItems()
}
}
```

You can build to check for syntax errors.

In this application, you immediately fetched all the instances of the **Item** entity. This is a simple request. In an application with a much larger data set, you would carefully fetch just the instances you needed. To selectively fetch instances, you add a *predicate* (an **NSPredicate**) to your fetch request, and only the objects that satisfy the predicate are returned.

A predicate contains a condition that can be true or false. For example, if you only wanted the items worth more than \$50, you would create a predicate and add it to the fetch request like this:

```
NSPredicate *p = [NSPredicate predicateWithFormat:@"valueInDollars > 50"];
[request setPredicate:p];
```

The format string for a predicate can be very long and complex. Apple's *Predicate Programming Guide* is a complete discussion of what is possible.

Adding and deleting items

Thus far, you have taken care of saving and loading, but what about adding and deleting? When the user wants to create a new **Item**, you will not allocate and initialize this new **Item**. Instead, you will ask the **NSManagedObjectContext** to insert a new object from the **Item** entity. It will then return an instance of **Item**.

In `ItemStore.swift`, edit the `createItem` method.

```

func createItem() -> Item {
    let newItem = Item()

    var order = 0.0
    if allItems.count == 0 {
        order = 1.0
    }
    else {
        order = allItems.last!.orderingValue + 1.0
    }
    println("Adding after \(allItems.count) items, order = \(order)")

    let newItem = NSEntityDescription.insertNewObjectForEntityForName("Item",
        inManagedObjectContext: context) as Item
    newItem.orderingValue = order

    allItems.append(newItem)
    return newItem
}

```

When a user deletes a **Item**, you must inform the context so that it is removed from the database. In **ItemStore.swift**, add the following code to **removeItem(_:)**.

```

func removeItem(item: Item) {
    if let index = find(allItems, item) {
        allItems.removeAtIndex(index)
        context.deleteObject(item)
    }
}

```

Reordering items

The last bit of functionality you need to replace for **Item** is the ability to re-order items in the **ItemStore**. Because Core Data will not handle ordering automatically, you must update a **Item**'s **orderingValue** every time it is moved in the table view.

This would get rather complicated if the **orderingValue** was an integer: every time a **Item** was placed in a new index, you would have to change the **orderingValue**'s of other items. This is why you created **orderingValue** as a double. You can take the **orderingValues** of the items that will be before and after the moving item, add them together, and divide by two. The new **orderingValue** will fall directly in between the values of the items that surround it.

In **ItemStore.swift**, modify **moveItemAtIndexPath(_:_:toIndex:)** to handle reordering items.

```
func moveItemAtIndex(fromIndex: Int, toIndex: Int) {
    if fromIndex == toIndex {
        return
    }

    let movedItem = allItems[fromIndex]
    allItems.removeAtIndex(fromIndex)
    allItems.insert(movedItem, atIndex: toIndex)

    // Computing a new orderValue for the object that was moved
    var lowerBound = 0.0
    if toIndex > 0 {
        lowerBound = allItems[toIndex - 1].orderingValue
    }
    else {
        lowerBound = allItems[1].orderingValue - 2.0
    }

    var upperBound = 0.0
    // Is there an object after it in the array?
    if toIndex > allItems.count - 1 {
        upperBound = allItems[toIndex + 1].orderingValue
    }
    else {
        upperBound = allItems[toIndex - 1].orderingValue + 2.0
    }

    let newOrderValue = (lowerBound + upperBound) / 2.0
    println("Moving to order \(newOrderValue)")
    movedItem.orderingValue = newOrderValue
}
```

Finally, you can build and run your application. Of course, the behavior is the same as it always was, but it is now using Core Data.

Adding AssetTypes to Homepwner

In the model file, you described a new entity, **AssetType**, that every item will have a to-one relationship to. You need a way for the user to set the **AssetType** of a **Item**. Also, the **ItemStore** will need a way to fetch the asset types. (Creating new instances of **AssetType** is left as a challenge at the end of this chapter.)

In **ItemStore.swift**, define a new method. If this is the first time the application is being run – and therefore there are no **AssetType** objects in the store – create three default types.

```

func fetchAssetTypes() {
    let request = NSFetchedResultsController()

    let e = NSEntityDescription.entityForName("AssetType", inManagedObjectContext: context)
    request.entity = e

    var error: NSError?
    if let results =
        context.executeFetchRequest(request, error: &error) as? [NSManagedObject] {
        allAssetTypes = allAssetTypes + results
    }
    else {
        println("Fetch failed: \(error!.localizedDescription)")
    }

    if allAssetTypes.count == 0 {
        var type: NSManagedObject
        type = NSEntityDescription.insertNewObjectForEntityForName("AssetType",
            inManagedObjectContext: context) as NSManagedObject
        type.setValue("Furniture", forKey: "label")
        allAssetTypes.append(type)

        type = NSEntityDescription.insertNewObjectForEntityForName("AssetType",
            inManagedObjectContext: context) as NSManagedObject
        type.setValue("Jewelry", forKey: "label")
        allAssetTypes.append(type)

        type = NSEntityDescription.insertNewObjectForEntityForName("AssetType",
            inManagedObjectContext: context) as NSManagedObject
        type.setValue("Electronics", forKey: "label")
        allAssetTypes.append(type)
    }
}

```

In `ItemStore.swift`, fetch the asset types when initializing the Core Data properties.

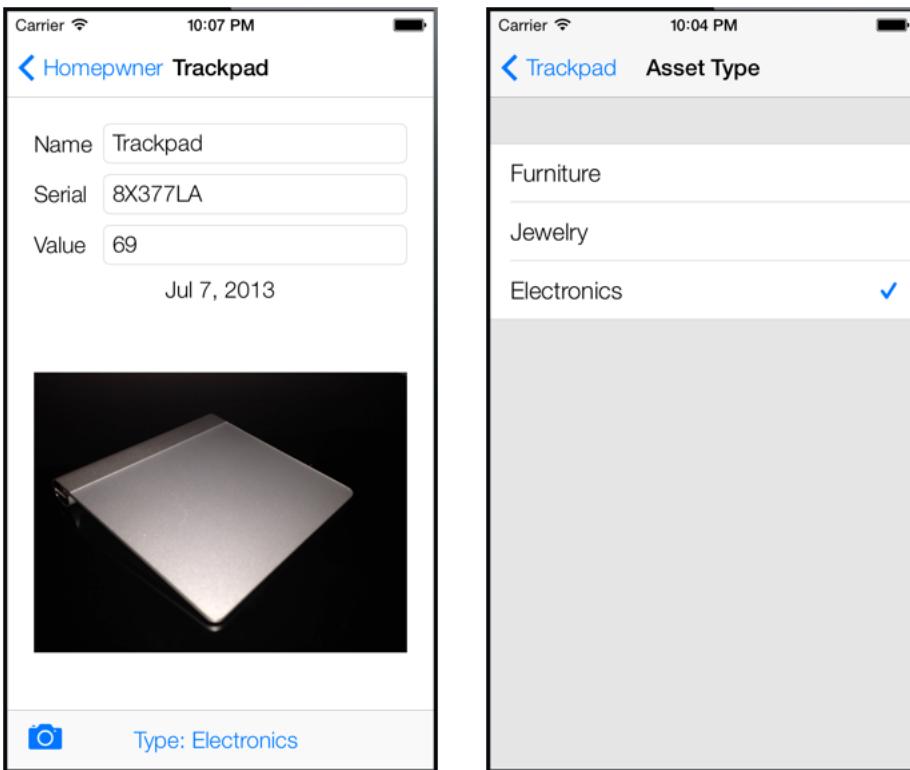
```

context = NSManagedObjectContext()
context.persistentStoreCoordinator = psc

self.loadAllItems()
self.fetchAssetTypes()

```

Now you need to change the user interface so that the user can see and change the `AssetType` of the `Item` in the `DetailViewController`.

Figure 14.10 Interface for **AssetType**

Create a new Cocoa Touch Class template file and choose **NSObject** as the superclass. Name this class **AssetTypeViewController**.

In **AssetTypeViewController.swift**, change the superclass to **UITableViewController** and give it an **Item** and **ItemStore** property.

```
class AssetTypeViewController: NSObject {
class AssetTypeViewController: UITableViewController {
    let item: Item
    let itemStore: ItemStore

    init(item: Item, itemStore: ItemStore) {
        self.item = item
        self.itemStore = itemStore

        super.init(nibName: nil, bundle: nil)
    }

    required init(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }
}
```

This table view controller will show a list of the available asset types. Tapping a button on the **DetailViewController**'s view will display it. Implement the data source methods **AssetTypeViewController.swift**. (You have seen all this before.)

```
override func viewDidLoad() {
    super.viewDidLoad()
    tableView.registerClass(UITableViewCell.self, forCellReuseIdentifier: "UITableViewCell")
}

override func tableView(tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    return itemStore.allAssetTypes.count
}
```

```

override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("UITableViewCell",
        forIndexPath: indexPath) as UITableViewCell

    let assetType = itemStore.allAssetTypes[indexPath.row]
    if let assetString = assetType.valueForKey("label") as? String {
        cell.textLabel?.text = assetString
    }

    if assetType == item.assetType {
        cell.accessoryType = .Checkmark
    } else {
        cell.accessoryType = .None
    }

    return cell
}

override func tableView(tableView: UITableView,
    didSelectRowAtIndexPath indexPath: NSIndexPath) {
    if let cell = tableView.cellForRowAtIndex(indexPath) {
        cell.accessoryType = .Checkmark

        let assetType = itemStore.allAssetTypes[indexPath.row]
        item.assetType = assetType
    }
}

```

In `DetailViewController.xib`, drag a `UIBarButtonItem` onto the toolbar. Create an outlet to this button by selecting the toolbar then the new bar button and Control-dragging to `DetailViewController.swift`. Name this outlet `assetTypeButton`. Then, create an action from this button in the same way it `showAssetTypePicker`.

The following method and instance variable should now be declared in `DetailViewController.swift`:

```

class DetailViewController: UIViewController {

    // Other outlets

    @IBOutlet weak var assetTypeButton: UIBarButtonItem!

    // Other methods

    @IBAction func takePicture(sender: AnyObject) {

    }
}

```

Finish implementing `showAssetTypePicker(_)` in `DetailViewController.swift`.

```

@IBAction func showAssetTypePicker(sender: AnyObject) {
    view.endEditing(true)

    let avc = AssetTypeViewController(item: item, itemStore: itemStore)
    showViewController(avc, sender: self)
}

```

And finally, update the title of the button to show the asset type of a `Item`. In `DetailViewController.swift`, add the following code to `viewWillAppear(_)`:

```

if let typeLabel = item.assetType.valueForKey("label") as? String {
    assetTypeButton.title = typeLabel
} else {
    assetTypeButton.title = "None"
}

```

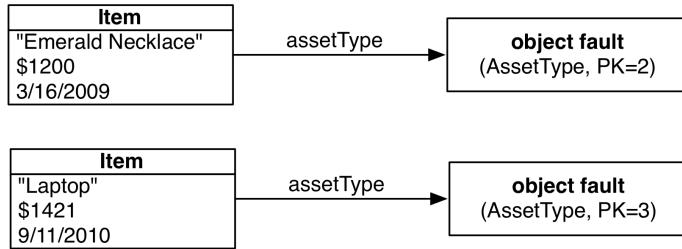
Build and run the application. Select a `Item` and set its asset type.

Faults

Relationships are fetched in a lazy manner. When you fetch a managed object with relationships, the objects at the other end of those relationships are *not* fetched. Instead, Core Data uses *faults*. Faults are lightweight placeholder objects that provide an endpoint for a relationship until the potentially larger objects are actually needed. This provides both performance and memory usage boons to object management.

There are to-many faults (which stand in for sets) and to-one faults (which stand in for managed objects). So, for example, when the instances of **Item** are fetched into your application, the instances of **AssetType** are not. Instead, fault objects are created that stand in for the **AssetType** objects until they are really needed.

Figure 14.11 Object faults

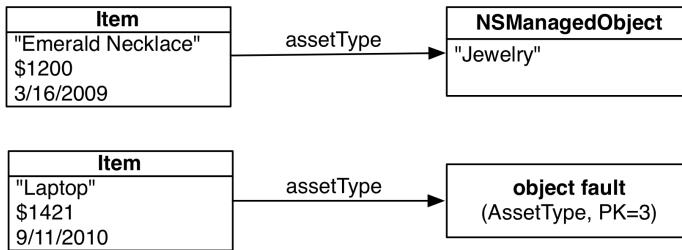


An object fault knows what entity it is from and what its primary key is. So, for example, when you ask a fault that represents an asset type what its label is, you will see SQL executed that looks something like this:

```
SELECT t0.Z_PK, t0.Z_OPT, t0.ZLABEL FROM ZAssetType t0 WHERE t0.Z_PK = 2
```

(Why is everything prefixed with `Z`? We do not know. What is `OPT`? We guess it is short for “optimistic locking.” These details are not important.) The fault is replaced, in the exact same location in memory, with a managed object containing the real data.

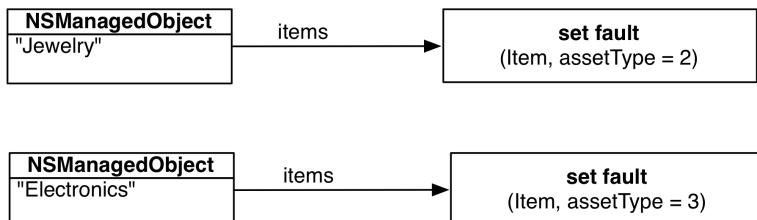
Figure 14.12 After one fault is replaced



This lazy fetching makes Core Data not only easy to use, but also quite efficient.

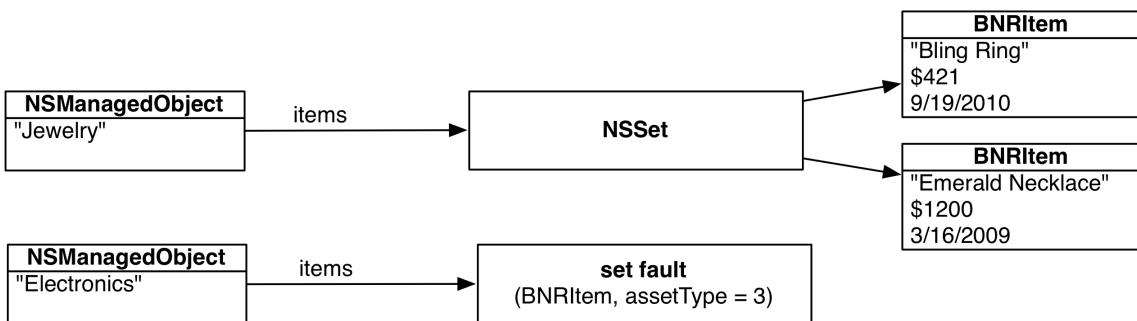
What about to-many faults? Imagine that your application worked the other way: the user is presented with a list of asset types to select from. Then, the items for that asset type are fetched and displayed. How would this work? When the assets are first fetched, each one has a set fault that is standing in for the **NSSet** of item objects:

Figure 14.13 Set faults



When the set fault is sent a message that requires the **Item** objects, it fetches them and replaces itself with an **NSSet**:

Figure 14.14 Set fault replaced



Core Data is a very powerful and flexible persistence framework, and this chapter has been just a quick introduction to its capabilities. For more details, we strongly suggest that you read Apple's *Core Data Programming Guide*. Here are some of the things we have not delved into:

- **NSFetchRequest** is a powerful mechanism for specifying data you want from the persistent store. We used it a little, but you will want to go deeper. You should also explore the following related classes: **NSPredicate**, **NSSortOrdering**, **NSEXpressionDescription**, and **NSEXpression**. Also, fetch request templates can be created as part of the model file.
- A *fetched property* is a little like a to-many relationship and a little like an **NSFetchRequest**. You typically specify them in the model file.
- As your app evolves from version to version, you will need to change the data model over time. This can be tricky – in fact, Apple has an entire guide about it: *Data Model Versioning and Data Migration Programming Guide*.
- There is good support for validating data as it goes into your instances of **NSManagedObject** and again as it moves from your managed object into the persistent store.
- You can have a single **NSManagedObjectContext** working with more than one persistent store. You partition your model into *configurations* and then assign each configuration to a particular persistent store. You are not allowed to have relationships between entities in different stores, but you can use fetched properties to achieve a similar result.

Trade-offs of Persistence Mechanisms

At this point, you can start thinking about the trade-offs between the common ways that iOS applications can store their data. Which is best for your application? Use Table 14.1 to help you decide.

Table 14.1 Data storage pros and cons

Technique	Pros	Cons
Archiving	Allows ordered relationships (arrays, not sets). Easy to deal with versioning.	Reads all the objects in (no faulting). No incremental updates.
Web Service	Makes it easy to share data with other devices and applications.	Requires a server and a connection to the Internet.
Core Data	Lazy fetches by default. Incremental updates.	Versioning is awkward (but can certainly be done using an NSModelMapping). No real ordering within an entity, although to-many relationships can be ordered.

Bronze Challenge: Assets on the iPad

On the iPad, present the **AssetTypeViewController** in a **UIPopoverController**.

Silver Challenge: New Asset Types

Make it possible for the user to add new asset types by adding a button to the `AssetTypeViewController`'s `navigationItem`.

Gold Challenge: Showing Assets of a Type

In the `AssetTypeViewController` view controller, create a second section in the table view. This section should show all of the assets that belong to the selected asset type.

15

Localization

The appeal of iOS is global – iOS users live in many different countries and speak many different languages. You can ensure that your application is ready for this global audience through the processes of internationalization and localization. *Internationalization* is making sure your native cultural information is not hard-coded into your application. (By cultural information, we mean language, currency, date formats, number formats, and more.)

Localization, on the other hand, is the process of providing the appropriate data in your application based on the user's Language and Region Format settings. You can find these settings in the Settings application. Select the General row and then the International row.

Figure 15.1 International settings



Apple makes these processes relatively simple. An application that takes advantage of the localization APIs does not even need to be recompiled to be distributed in other languages or regions. In this chapter, you are going to localize the item detail view of Homepwner. (By the way, “internationalization” and “localization” are long words. You will sometimes see people abbreviate them to i18n and L10n, respectively.)

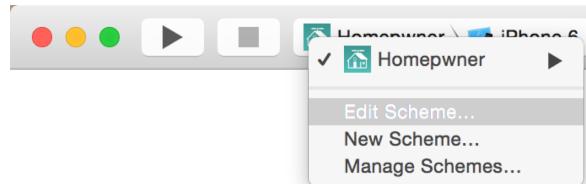
In this chapter, you will internationalize the Homepwner application, and then localize it into Spanish.

Internationalization

In this first section, you will use the class **NSNumberFormatter** to internationalize the number format and currency symbol for the value of an item.

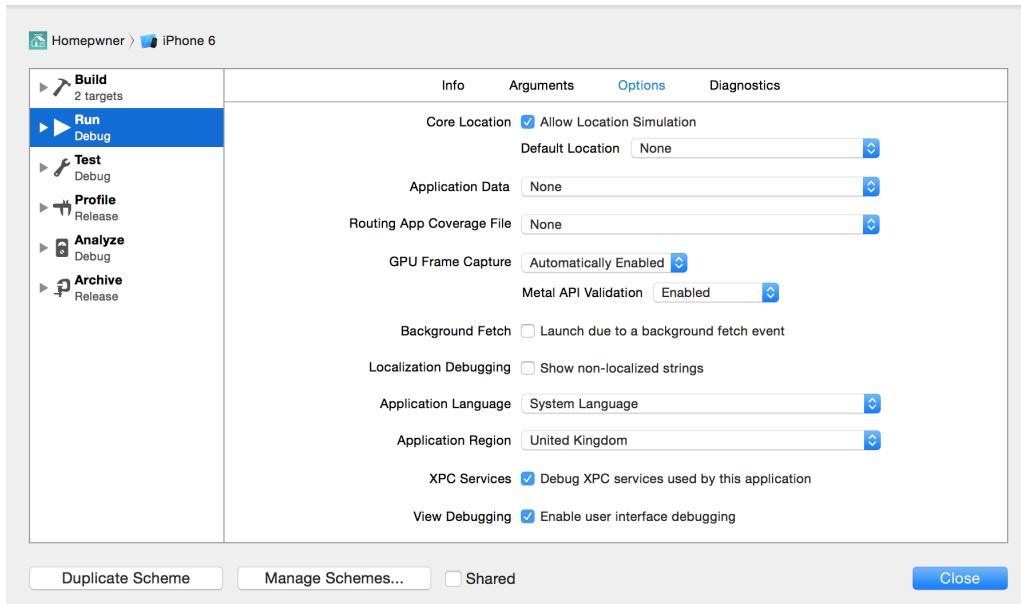
Did you know that Homepwner is already partially internationalized? Open `Homepwner.xcodeproj`. Launch the application and add a new item. The date label in the **DetailViewController** is formatted according to the current regional settings. In the US, the dates are displayed as *Month Day, Year*. Cancel adding a new item.

Figure 15.2 Edit Scheme



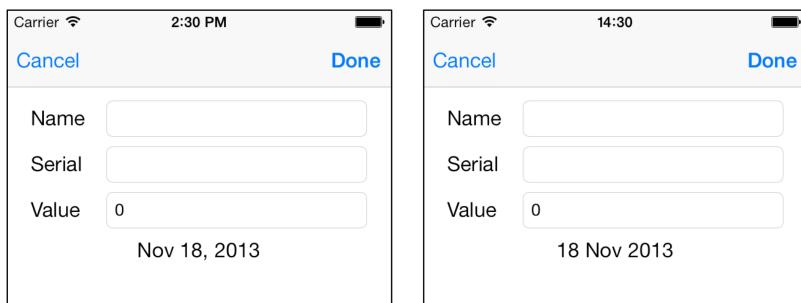
Now, run the application using the United Kingdom region. To do this, select the active scheme popup and select Edit Scheme (Figure 15.2). Make sure that Run is selected on the lefthand side and then select the Options tab at the top. For the Application Region popup, select Europe and then United Kingdom (Figure 15.3). Finally, Close the active scheme window.

Figure 15.3 Selecting a Different Region



Build and run Homepwner again and add a new item. This time, the date is displayed as *Day Month Year*. The text for the date label has already been internationalized. When did this happen?

Figure 15.4 Date format: US vs UK



Formatters

In Chapter 7, you used an instance of `NSDateFormatter` to set the text of the date label of `DetailViewController`. `NSDateFormatter` has a `locale` property, which is set to the device's current locale.

Whenever you use an **NSDateFormatter** to create a date, it checks its **locale** property and sets the format accordingly. So the text of the date label has been internationalized from the start.

NSLocale knows how different regions display symbols, dates, and decimals and whether they use the metric system. An instance of **NSLocale** represents one region's settings for these variables. In the Settings application, the user can choose a region, like United States or United Kingdom. (Why does Apple use "region" instead of "country?" Some countries have more than one region with different settings. Scroll through the options in Application Region to see for yourself.)

When you send the message **currentLocale** to **NSLocale**, the instance of **NSLocale** that represents the user's region setting is returned. Once you have that instance of **NSLocale**, you can ask it questions like, "What is the currency symbol for this region?" or "Does this region use the metric system?"

To ask one of these questions, you send the **NSLocale** instance the message **objectForKey(_:)** with one of the **NSLocale** constants as an argument. (You can find all of these constants in the **NSLocale** documentation page.)

```
let currentLocale = NSLocale.currentLocale()
let isMetric = currentLocale.objectForKey(NSLocaleUsesMetricSystem)?.boolValue
let currencySymbol = currentLocale.objectForKey(NSLocaleCurrencyCode)
```

Let's internationalize the value amount displayed in each **ItemCell**.

While **NSLocale** is extremely powerful and useful, always using it directly would make the process of localizing apps very tedious. That's why you used **NSDateFormatter** earlier. There is another class, **NSNumberFormatter** that does for numbers what **NSDateFormatter** does for dates. For example, to format a number appropriately for the current locale, use the **stringFromNumber(_:)** method. Depending on the locale, the **numberAsString** may be `123,456.789` or `123 456,789` or some other value.

```
let numberFormatter = NSNumberFormatter()
let numberAsString = numberFormatter.stringFromNumber(123456.789)
```

What makes **NSNumberFormatter** even more useful is its capability to format currency amounts. If the number formatter's **numberStyle** property is set to **NSNumberFormatterStyle.CurrencyStyle**, it will start producing the numbers formatted not only with the appropriate group and decimal separators, but also with the currency symbol. (In some countries, numbers may be formatted differently for currency and non-currency purposes.)

In `ItemsViewController.swift`, locate the method **tableView(_:cellForRowAtIndexPath:)**. Create an **NSNumberFormatter** and set its **numberStyle** to **.CurrencyStyle**.

```
cell.serialNumberLabel.text = item.serialNumber

// Create a number formatter for currency
let currencyFormatter = NSNumberFormatter()
currencyFormatter.numberStyle = .CurrencyStyle

cell.valueLabel.text = "$\\" + (item.valueInDollars)"
```

When the text of the cell's **valueLabel** is set in this method, the string "\$" is used, which makes the currency symbol always a dollar sign. Use the **currencyFormatter** to format the amount correctly.

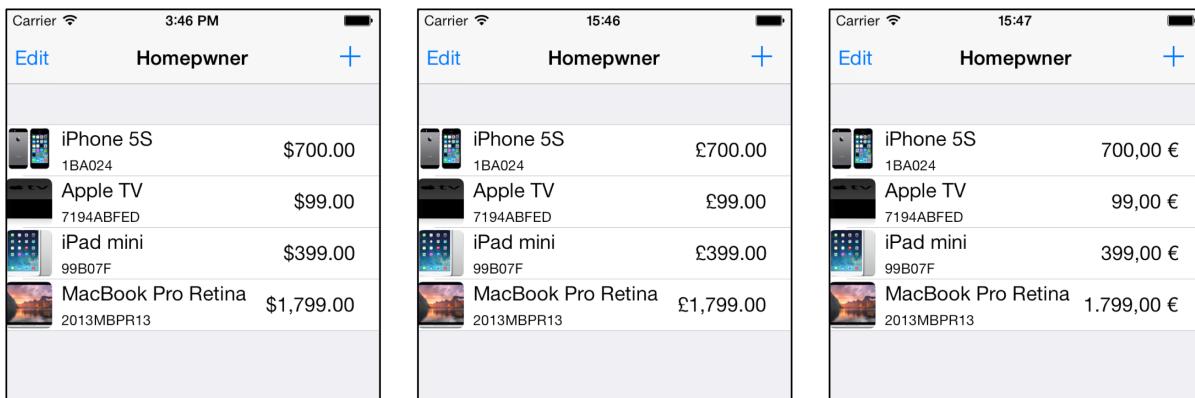
```
// Create a number formatter for currency
let currencyFormatter = NSNumberFormatter()
currencyFormatter.numberStyle = .CurrencyStyle

cell.valueLabel.text = "$\\" + (item.valueInDollars)""
cell.valueLabel.text = currencyFormatter.stringFromNumber(item.valueInDollars)
```

These changes will display the value formatted appropriately for the user's region, with both the number format and currency symbol.

Build and run the application. You will see the value amount formatted according to the currently selected region, which is United Kingdom.

Figure 15.5 Number format: US vs UK vs Germany



Base Internationalization

When internationalizing, you ask the instance of `NSLocale` questions. But the `NSLocale` only has a few region-specific variables. This is where localization comes into play: Localization is the process by which application-specific substitutions are created for different region and language settings.

Localization usually means one of two things:

- generating multiple copies of resources like images, sounds, and NIB files for different regions and languages
- creating and accessing *strings tables* to translate text into different languages

Any resource, whether it is an image or a XIB file, can be localized. Localizing a resource puts another copy of the resource in the application bundle. These resources are organized into language-specific directories, known as `lproj` directories. Each one of these directories is the name of the localization suffixed with `lproj`. For example, the American English localization is `en_US`: where `en` is the English language code and `US` is the United States of America region code. (The region can be omitted if you do not need to make regional distinctions in your resource files.) These language and region codes are standard on all platforms, not just iOS.

When a bundle is asked for the path of a resource file, it first looks at the root level of the bundle for a file of that name. If it does not find one, it looks at the locale and language settings of the device, finds the appropriate `lproj` directory, and looks for the file there. Thus, just by localizing resource files, your application will automatically load the correct file.

One option is to create separate XIB files and to manually edit each string in this XIB file in Xcode. However, this approach does not scale well if you are planning multiple localizations. What happens when you add a new label or button to your localized XIB? You have to add this view to the XIB for every language. This is not fun.

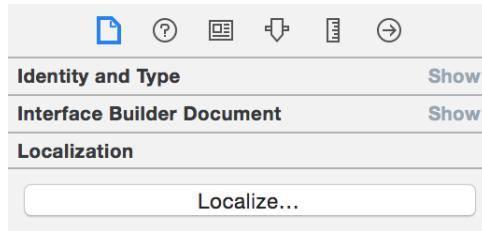
To simplify the process of localizing XIB files, Xcode has a feature called *base internationalization*. When it is enabled for the project, base internationalization creates the `Base.lproj` directory which contains the main XIB files. Localizing individual XIB files can then be done by creating just the `Localizable.strings` files. It is still possible to create the full XIB files, in case localization cannot be done by changing strings alone. However, with the help of Auto Layout, strings replacement may be sufficient for most localization needs.

In this section, you are going to localize one of Homepwner's interfaces: the `DetailViewController.xib` file. You will create English and Spanish localizations, which will create two `lproj` directories, in addition to the base one. Normally, you would first enable Base Internationalization in the project Info settings. However, as of this writing, there is a bug in Xcode that will not let you enable that option until at least one XIB file is localized.

So, start by localizing a XIB file. Select `DetailViewController.xib` in the project navigator. Then, show the utility area.

Click the tab in the inspector selector to open the *file inspector*. Find the section in this inspector named Localization and click the `Localize...` button (Figure 15.6).

Figure 15.6 Localizing DetailViewController.xib, beginning



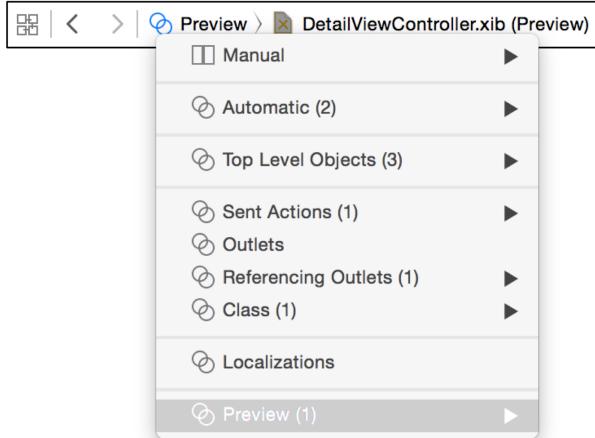
Select **Base**. This signifies to Xcode that this file can be localized, automatically creates `Base.lproj`, and moves the `DetailViewController.xib` file to it.

Check the English box and make sure that the drop-down says **Localizable Strings**. This will create a strings table that you will use later to localize the application.

Pseudolanguages

Open `DetailViewController.xib`, and then show the Assistant editor. From the jump bar drop-down, select **Preview** (Figure 15.7). The *preview assistant* allows you to easily see how your interface will look across various screen sizes and orientations as well as between different localized languages.

Figure 15.7 Opening the Preview Assistant



In the bottom lefthand corner, there is an add button which allows you to add additional screen sizes to the preview canvas. An existing device on the canvas can be deleted by selecting it and pressing the delete key. This allows you to easily see how changes to your interface propagate across various screen sizes and orientations simultaneously. We will stick with the iPhone 4.7-inch (Figure 15.8).

Figure 15.8 Preview Assistant

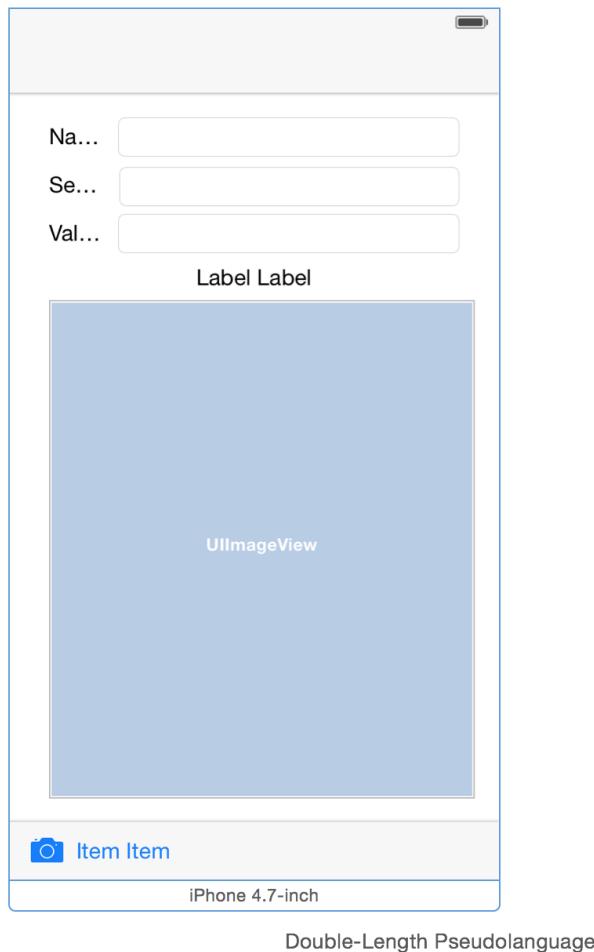


In the bottom righthand corner, there is a button to select a language to preview this XIB with. You have not localized the application into another language yet, but Xcode supplies a *pseudolanguage* for you to use.

Pseudolanguages help you internationalize your applications before receiving translations for all of your strings and assets. The built-in pseudolanguage, Double-Length Pseudolanguage, mimicks languages that are more verbose by repeating whatever text string is in the text element. So, for example, “Name” becomes “Name Name”.

Select the Language popup that says English and select Double-Length Pseudolanguage. The labels and bar button items all have their text doubled (Figure 15.9).

Figure 15.9 Double-Length Pseudolanguage



By using the double-length pseudolanguage, a problem is immediately identified: all three labels at the top each are truncating their text. The name, serial number, and value labels all have fixed width and fixed height constraints. By fixing the width and height, the label is not able to shrink and grow as its text and font change. Instead, views should rely on their intrinsic content size whenever possible to determine their width and height.

In `DetailViewController.xib`, delete the explicit width and height constraints on the Name, Serial, and Value labels. The preview assistant confirms that the text is no longer being truncated (Figure 15.10). Since the text fields are always positioned between the label and their superview, the text fields shrink to accommodate the longer text. Finally, Update Frames for any misplaced views on the canvas.

Figure 15.10 Updated Auto Layout

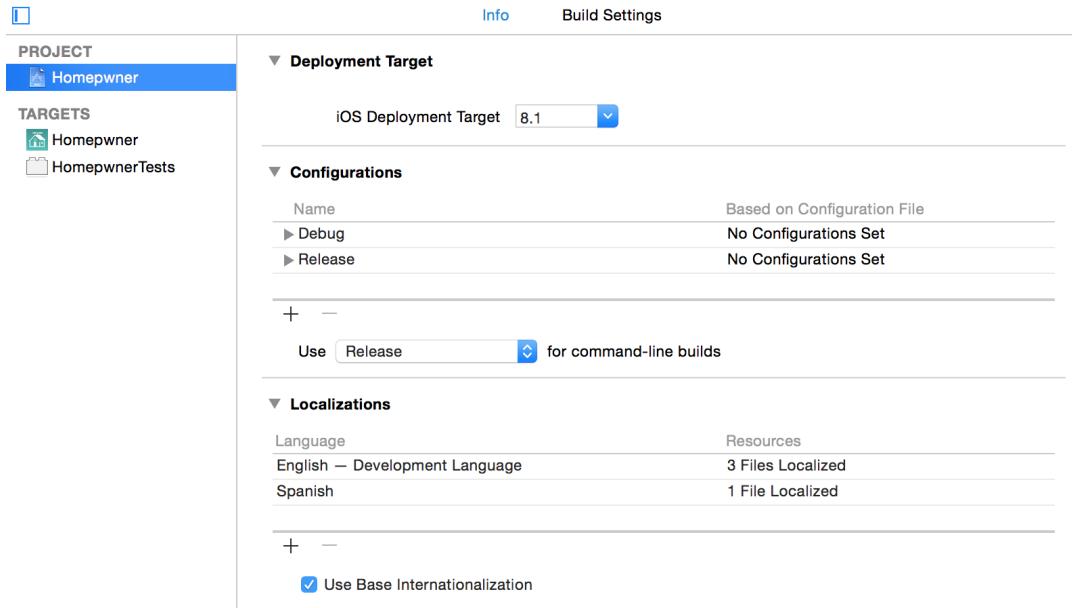


Localization

Homepwner is now internationalized - its interface is able to adapt to various languages and regions. Now it's time to localize the app; that is, update the strings and resources in the application for a new language.

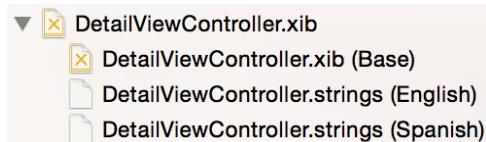
In the Project navigator, select the Homepwner project at the top. Then select the Homepwner project in the side list, and make sure the Info tab is open. Click the + button under the list of languages and select Spanish. In the dialog, you can uncheck the LaunchScreen.xib file and only keep the DetailViewController.xib file checked. Make sure that the reference language is Base and the file type is Localizable Strings. Click Finish. This creates an es.lproj folder and generates the DetailViewController.strings in it that contains all the strings from the base XIB file. The Localizations configuration should look like Figure 15.11.

Figure 15.11 Localizations



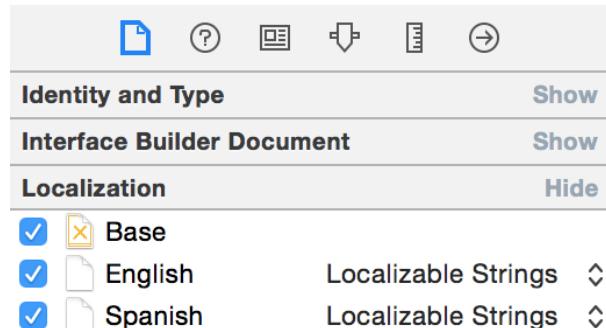
Look in the project navigator. Click the disclosure button next to DetailViewController.xib (Figure 15.12). Xcode moved the DetailViewController.xib file to the Base.lproj directory and created the DetailViewController.strings file in the es.lproj directory.

Figure 15.12 Localized XIB in the project navigator



Select the DetailViewController.xib. It does not matter if you select the top level one or the one marked Base. The file inspector should look like Figure 15.13.

Figure 15.13 Localizing DetailViewController.xib, result



In the project navigator, click the Spanish version of `DetailViewController.strings`. When this file opens, the text is not in Spanish. You have to translate localized files yourself; Xcode is not *that* smart.

Edit this file according to the following text. The numbers and order may be different in your file, but you can use the `text` field in the comment to match up the translations.

```
/* Class = "IBUILabel"; text = "Name"; ObjectID = "DcM-Ta-Pno"; */
"DcM-Ta-Pno.text" = "Nombre";

/* Class = "IBUIBarButtonItem"; title = "Item"; ObjectID = "HFl-jt-j04"; */
"HFl-jt-j04.title" = "Item";

/* Class = "IBUILabel"; text = "Serial"; ObjectID = "IyW-WC-s8a"; */
"IyW-WC-s8a.text" = "Número de serie";

/* Class = "IBUILabel"; text = "Value"; ObjectID = "JgR-jx-IXE"; */
"JgR-jx-IXE.text" = "Valor";

/* Class = "IBUILabel"; text = "Label"; ObjectID = "Qyh-9V-hKs"; */
"Qyh-9V-hKs.text" = "Label";
```

Notice that you did not change the `Label` and `Item` text because those strings will be replaced programmatically at runtime. Save this file.

Now that you have finished localizing this XIB file, let's test it out. First, there is a little Xcode glitch to be aware of: sometimes Xcode just ignores a resource file's changes when you build an application. To ensure your application is being built from scratch, first delete it from your device or simulator. (Press and hold its icon in the launcher. When it starts to wiggle, tap the delete badge.) Relaunch Xcode (yes, exit and start it again). Then, choose `Clean` from the `Product` menu. Finally, to be absolutely sure, press and hold the Option key while opening the `Product` menu and choose `Clean Build Folder...` This will force the application to be entirely re-compiled, re-bundled, and re-installed.

Open the active scheme popup and select `Edit Scheme`. Make sure `Run` is selected in the lefthand side pane, and open the Options tab. Open the Application Language popup and select Spanish. Finally open the Application Region popup and select Europe then Spain. Close the window.

Build and run the application. Select an item row, and you will see the interface in Spanish. Because you have already set the constraints on the labels, they resize themselves fairly well (Figure 15.14).

Figure 15.14 Spanish `DetailViewController`



The interface would look a lot nicer if the text fields' leading edges were aligned. Now that the `Name`, `Serial`, and `Value` labels no longer have their width constraints, this is possible.

In `DetailViewController.xib`, select the `nameField`, `serialNumberField`, and `valueField`. Open the Align Auto Layout menu and check the box for Leading Edges. Make sure the constant is `0`. Update Frames for Items of New Constraints and then Add 2 Constraints. Either build and run the application, or view the interface in the preview assistant to see the aligned text fields (Figure 15.15).

Figure 15.15 Text Fields Aligned



NSLocalizedString and Strings Tables

In many places in your applications, you create **String** instances dynamically or display string literals to the user. To display translated versions of these strings, you must create a *strings table*. A strings table is a file containing a list of key-value pairs for all of the strings that your application uses and their associated translations. It is a resource file that you add to your application, but you do not need to do a lot of work to get data from it.

You might use a string in your code like this:

```
let greeting = "Hello!"
```

To internationalize the string in your code, you replace literal strings with the function **NSLocalizedString**.

```
let greeting = NSLocalizedString("Hello!", comment: "The greeting for the user")
```

This function takes two arguments: a key and a comment that describes the string's use. The key is the lookup value in a strings table. At runtime, **NSLocalizedString()** will look through the strings tables bundled with your application for a table that matches the user's language settings. Then, in that table, the function gets the translated string that matches the key.

Now you are going to internationalize the string "Homepwner" that is displayed in the navigation bar. In **ItemsViewController.swift**, locate the **init(itemStore:imageStore:)** method and change the line of code that sets the title of the **navigationItem**.

```
init(itemStore: ItemStore, imageStore: ImageStore) {
    self.itemStore = itemStore
    self.imageStore = imageStore
    super.init(nibName: nil, bundle: nil)

    navigationItem.title = "Homepwner"
    navigationItem.title = NSLocalizedString("Homepwner", comment: "Name of application")
```

Two more view controllers contain hard-coded strings that can be internationalized. The toolbar in the **DetailViewController** shows the asset type. The title of the **AssetTypeViewController** needs to be updated just like the title of the **ItemsViewController**.

In **DetailViewController.swift**, update the **viewWillAppear(_:)** method:

```
if let typeLabel = item?.assetType?.valueForKey("label") as? String {
    assetTypeButton.title = "Type: \(typeLabel)"

    let formatString = NSLocalizedString("Type: %@", comment: "Asset type button")
    let assetButtonTitle = NSString(format: formatString, typeLabel)
    assetTypeButton.title = assetButtonTitle
}

else {
    assetTypeButton.title = "None"
    assetTypeButton.title = NSLocalizedString("None", comment: "Type label None")
}
```

Since string interpolation is being used to build up the title, you must use **NSString**'s interpolation technique in order for it to work well with internationalization. To understand why, you must understand **NSString(format:_:)**.

NSString(format:_:) takes a format string as its first argument. This format string will contain a number of placeholder tokens, such as "%@". Each token corresponds to a variable that should be inserted into the string, and each token will have a corresponding argument with that variable after the format string.

For example, the follow creates a string with information about a person:

```
let person = ...

let name: String = person.name
let age: Int = person.age
let height: Float = person.height

let infoString =
    NSString(format: "%@ is %i years old and is %f inches tall.", name, age, height)
```

How does this help us? We are able to localize the format string independently of each of the arguments:

```
let formatString =
    NSLocalizedString("%@ is %i years old and is %f inches tall.", comment: "Information describing a person")
let infoString = NSString(format: formatString, name, age, height)
```

By contrast, if you used the **String** approach to interpolation, you might implement the asset title button like so:

```
assetTypeButton.title =
    NSLocalizedString("Type: \(typeLabel)", comment: "Asset type button")
```

The problem with this approach is that at run time, if the `typeLabel` corresponds to “Electronics”, the lookup string will be “Type: Electronics”, which won't be a key in the strings table.

By using **NSString(format:_:)**, the lookup key will be “Type: %@” independent of the specific `typeLabel` getting passed as an argument.

Be sure to take a look at the documentation for **NSString** to see the various placeholder tokens that can be used in the format string.

With that out of the way, **AssetTypeViewController** now needs a title, so add it in `AssetTypeViewController.swift`'s `init` method:

```
override init() {
    super.init(nibName: nil, bundle: nil)
    navigationItem.title = NSLocalizedString("Asset Type", comment: "Asset type title")
}

required init(coder aDecoder: NSCoder) {
    fatalError("init(coder:) has not been implemented")
}
```

Once you have files that have been internationalized with the **NSLocalizedString** function, you can generate strings tables with a command-line application.

Open the Terminal app. If you have never used it before, this is a Unix terminal; it is used to run command-line tools. You want to navigate to the location of `ItemsViewController.swift`. If you have never used the Terminal app before, here is a handy trick. In Terminal, type the following:

```
cd
```

followed by a space. (Do not press Enter yet.)

Next, open Finder and locate `ItemsViewController.swift` and the folder that contains it. Drag the icon of that folder onto the Terminal window. Terminal will fill out the path for you. Press Enter.

The current working directory of Terminal is now this directory. For example, my terminal command looks like this:

```
cd /Users/cbkeur/Desktop/Homepwner/Homepwner/
```

Use the terminal command `ls` to print out the contents of the working directory and confirm that `ItemsViewController.swift` is in that list.

To generate the strings table, enter the following into Terminal and press Enter:

```
genstrings ItemsViewController.swift
```

This creates a file named `Localizable.strings` in the same directory as `ItemsViewController.swift`. Now you need to generate strings from the other two view controllers. Since the file `Localizable.strings` already exists, you will want to append to it, rather than create it from scratch. To do so, enter the following commands in the Terminal (do not forget the `-a` command line option) and press Enter after each line:

```
genstrings -a DetailViewController.swift
genstrings -a AssetTypeViewController.swift
```

The resulting file `Localizable.strings` now contains the strings from all three view controllers. Drag from the Finder into the project navigator (or use the Add Files to "Homepwner"... menu item). When the application is compiled, this resource will be copied into the main bundle.

Open `Localizable.strings`. The file should look something like this:

```
/* Name of application */
"Homepwner" = "Homepwner";

/* Type label None */
"None" = "None";

/* Asset type button */
"Type: %@", "Type: %@",

/* Asset type title */
"Asset Type" = "Asset Type";
```

Notice that the comment above your string is the second argument you supplied to the `NSLocalizedString` function. Even though the function does not require the comment argument, including it will make your localizing life easier.

Now that you have created `Localizable.strings`, localize it in Xcode the same way you did the XIB file. Select the file in the project navigator and click the Localize... button in the utility area. Select Base, and then add the Spanish and English localization by checking the box next to each language. Open the Spanish version of `Localizable.strings`. The string on the lefthand side is the *key* that is passed to the `NSLocalizedString` function, and the string on the righthand side is what is returned. Change the text on the righthand side to the Spanish translation shown below. (To type ñ, press Option-n and then "n".)

```
/* Name of application */
"Homepwner" = "Dueño de casa";

/* Type label None */
"None" = "Nada";

/* Asset type button */
"Type: %@", "Tipo: %@",

/* AssetTypePicker title */
"Asset Type" = "Tipo de activo";
```

Build and run the application again. Now all these strings, including the title of the navigation bar, will appear in Spanish. If they do not, you might need to delete the application, clean your project, and rebuild. (Or check your user language setting.)

Manually Updating the Strings Table

Currently, three strings in the project have yet to be localized: “Electronics”, “Furniture”, and “Jewelry”. These strings are stored in Core Data. Because of this, they should not be stored using `NSLocalizedString`. That way, the model is consistent across languages and regions. Instead of localizing the string to be stored in Core Data, you will localize the string when it is presented to the user.

Open `DetailViewController.swift` and update `viewWillAppear(_:)` to use a localized version of the asset type string:

```
if let typeLabel = item?.assetType?.valueForKey("label") as? String {
    let localizedTypeLabel = NSLocalizedString(typeLabel, comment: "Type label")

    let formatString = NSLocalizedString("Type: %@", comment: "Asset type button")
    let assetButtonTitle = NSString(format: formatString, localizedTypeLabel)
    assetTypeButton.title = assetButtonTitle
}
else {
    assetTypeButton.title = NSLocalizedString("None", comment: "Type label None")
}
```

Then, open `AssetViewController.swift` and do the same in `tableView(_:cellForRowAtIndexPath:)`.

```
override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("UITableViewCell",
        forIndexPath: indexPath) as UITableViewCell

    let assetType = itemStore.allAssetTypes[indexPath.row]
    if let assetString = assetType.valueForKey("label") as? String {
        let localizedAssetString =
            NSLocalizedString(assetString, comment: "Asset type string")
        cell.textLabel.text = localizedAssetString
    }

    if assetType == item.assetType {
        cell.accessoryType = .Checkmark
    }
    else {
        cell.accessoryType = .None
    }

    return cell
}
```

Now, manually add the following strings to the bottom of the Base and English `Localizable.strings` files.

```
/* Electronics */
"Electronics" = "Electronics";

/* Furniture */
"Furniture" = "Furniture";

/* Jewelry */
"Jewelry" = "Jewelry";
```

Finally, add the following string translations to the bottom of the Spanish `Localizable.strings` file.

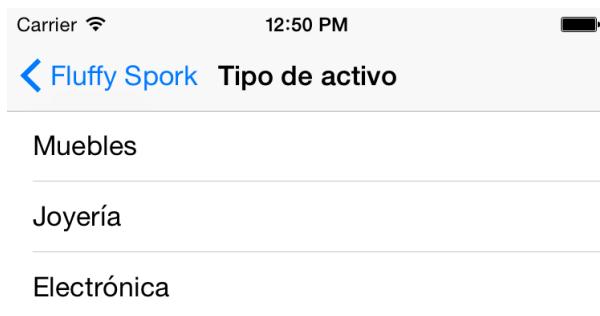
```
/* Electronics */
"Electronics" = "Electrónica";

/* Furniture */
"Furniture" = "Muebles";

/* Jewelry */
"Jewelry" = "Joyería";
```

Build and run the application. The asset types are now translated into Spanish (Figure 15.16).

Figure 15.16



Bronze Challenge: Another Localization

Practice makes perfect. Localize Homepwner for another language. Use Google Translate if you need help with the language.

For the More Curious: NSBundle's Role in Internationalization

The real work of adding a localization is done for you by the class **NSBundle**. For example, when a **UIViewController** is initialized, it is given two arguments: the name of a XIB file and an **NSBundle** object. The bundle argument is typically `nil`, which is interpreted as the application's *main bundle*. (The main bundle is another name for the application bundle – all of the resources and the executable for the application. When an application is built, all of the `lproj` directories are copied into this bundle.)

When the view controller loads its view, it asks the bundle for the XIB file. The bundle, being very smart, checks the current language settings of the device and looks in the appropriate `lproj` directory. The path for the XIB file in the `lproj` directory is returned to the view controller and loaded.

NSBundle knows how to search through localization directories for every type of resource using the instance method **pathForResource(_:ofType:)**. When you want a path to a resource bundled with your application, you send this message to the main bundle. Here is an example using the resource file `myImage.png`:

```
let path = NSBundle.mainBundle().pathForResource("myImage" ofType: "png")
```

The bundle first checks to see if there is a `myImage.png` file in the top level of the application bundle. If so, it returns the full path to that file. If not, the bundle gets the device's language settings and looks in the appropriate `lproj` directory to construct the path. If no file is found, it returns `nil`.

This is why you must delete and clean an application when you localize a file. The previous un-localized file will still be in the root level of the application bundle because Xcode will not delete a file from the bundle when you re-install. Even though there are `lproj` folders in the application bundle, the bundle finds the top-level file first and returns its path.

For the More Curious: Importing and Exporting as XLIFF

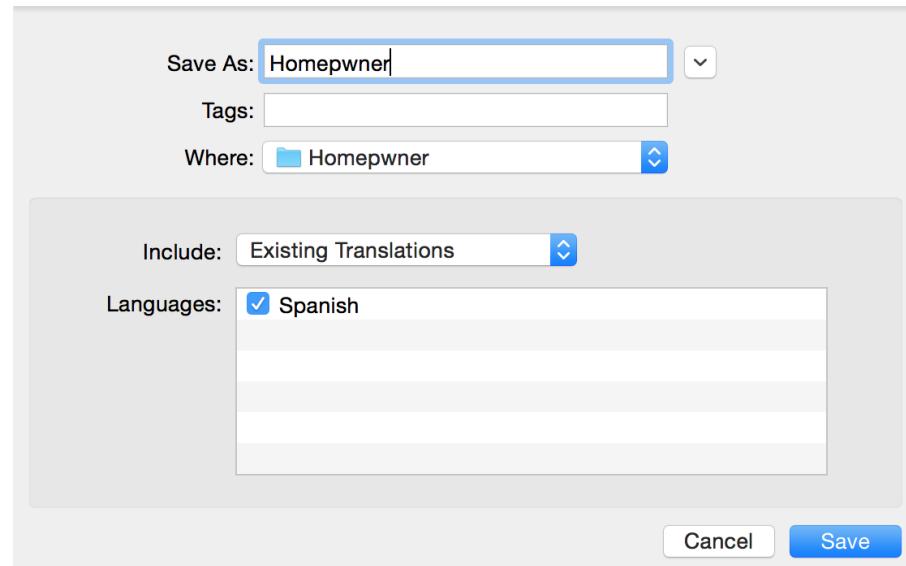
The industry standard format for localization data is the **XLIFF** data type which stands for XML Localisation Interchange File Format. When working with a translator, you will often send them an XLIFF file containing the data in the application to localize, and they will give you back a localized XLIFF file in return for you to import.

Xcode natively supports importing and exporting localization data in the XLIFF format. The exporting process will take care of finding and exporting the localized strings within the project, which you previously did manually using the `genstrings` tool.

To export the localizable strings in XLIFF format, select the project (`Homepwner`) in the Project navigator. Then select the Editor menu, and then Export For Localization.... On the next screen, you can choose whether to export

existing translations (which is probably a good idea so the translator doesn't do redundant work), and which languages you would like exported (Figure 15.17).

Figure 15.17 Exporting Localization Data as XLIFF



To import localizations, select the project (Homepwner) in the Project navigator. Then select the Editor menu, and then Import Localizations.... After choosing a file, you will be able to confirm the updates before you Import.

16

Controlling Animations

The word “animation” is derived from a Latin word that means “the act of bringing to life.” Animations are what bring your applications to life, and when used appropriately, they can guide your users through a course of actions, orient them, and overall create a delightful experience.

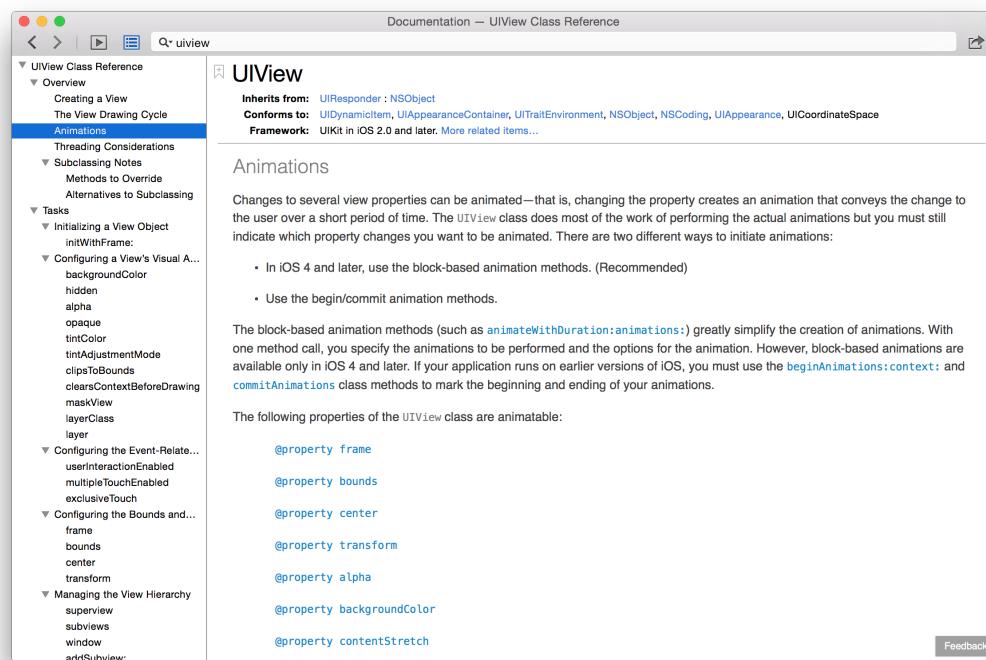
In this chapter, you will use a variety of animation techniques to animate various views in the HypnoNerd application.

Basic Animations

Animations are a great way to add an extra layer of polish to any application; games are not the only type of application to benefit from animations. Animations can smoothly bring interface elements on screen or into focus, they can draw the user’s attention to an actionable item, and they give clear indications of how your app is responding to the user’s actions.

Before updating the HypnoNerd application, let’s take a look at what can be animated. Open the documentation to the *UIView Class Reference*, and scroll down to the section titled *Animations*. The documentation will give some animation recommendations (which we will follow in this book) and also list the properties on **UIView** that can be animated (Figure 16.1).

Figure 16.1 **UIView** animation documentation



The documentation is always a good starting point in learning about any iOS technology. With that little bit of information under your belt, let’s go ahead and add some animations to HypnoNerd. The first type of animation

you are going to use is the *basic animation*. A basic animation animates between a start value and an end value (Figure 16.2).

Figure 16.2 Basic animation



Open HypnoNerd.xcodeproj.

The first animation you will add will animate the alpha value of a label associated with **HypnosisViewController**.

Open HypnosisViewController.swift. Add a property for a **UILabel** and add it to the view in **viewDidLoad**.

```
class HypnosisViewController: UIViewController {

    weak var textLabel: UILabel!

    // Other methods

    override func loadView() {
        let frame = UIScreen.mainScreen().bounds
        let backgroundView = HypnosisView(frame: frame)
        view = backgroundView

        let label = UILabel(frame: CGRectMake(0, 0, width: 200, height: 200))
        label.center = view.center
        label.font = UIFont.systemFontOfSize(24)
        label.textAlignment = NSTextAlignment.Center
        label.text = "Achieve\nNerdvana"
        label.numberOfLines = 2
        view.addSubview(label)
        textLabel = label
    }
}
```

Then, override **viewWillAppear(_:)** and **viewDidAppear(_:)** to fade in the label every time the user views the Hypnosis tab.

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)

    // Set the label's initial alpha
    textLabel.alpha = 0
}

override func viewDidAppear(animated: Bool) {
    super.viewDidAppear(animated)

    // Animate the alpha to 1.0
    UIView.animateWithDuration(1.0, animations: {
        self.textLabel.alpha = 1
    })
}
```

Build and run the application. When you view the Hypnosis tap, the label will fade into view. Animations provide a less jarring user experience than having the views just pop into existence.

The method **animateWithDuration(_:animations:)** returns immediately. That is, it starts the animation, but does not wait around for the animation to complete.

The simplest closure-based animation method on **UIView** is **animateWithDuration(_:animations:)**. This method takes in the duration that the animation should run for and a closure of changes to animate. The animation will follow an ease-in/ease-out animation curve, which will cause the animation to begin slowly, accelerate through the middle, and finally slow down at the end.

Timing functions

The acceleration of the animation is controlled by its timing function. The method `animateWithDuration(_:animations:)` uses an ease-in/ease-out timing function. To use a driving analogy, this would mean the driver accelerates smoothly from rest to a constant speed, and then gradually slows down at the end, coming to rest.

Other timing functions include linear (a constant speed from beginning to end), ease-in (accelerating to a constant speed, and then ending abruptly), and ease-out (beginning at full speed, and then slowing down at the end).

In order to use one of these other timing functions, you will need to use the `UIView` animation method that allows options to be specified: `animateWithDuration(_:delay:options:animations:completion:)`. This method gives you the most control over the animation. In addition to the duration and the animation closure, you can also specify how long to delay before the animations should begin, some options (which we will look at shortly), and a completion closure that will get called when the animation sequence completes.

In `HypnosisViewController.swift`, change the animation in `viewDidAppear(_:)` to use this new animation method:

```
override func viewDidAppear(animated: Bool) {
    super.viewDidAppear(animated)

    // Animate the alpha to 1.0
    UIView.animateWithDuration(1.0, animations: [
        self.textLabel.alpha = 1
    ])

    UIView.animateWithDuration(1.0,
        delay: 0.0,
        options: .CurveEaseIn,
        animations: {
            self.textLabel.alpha = 1.0
        },
        completion: nil)
}
```

Now, as opposed to using the default ease-in/ease-out animation curve, the animation will just ease-in. The options argument is a bitmask, so you can bitwise-or multiple values together. Here are some of the useful options that you can supply:

Animation curve options

These control the acceleration of the animation. Possible values are

- `UIViewControllerAnimatedOptions.CurveEaseInOut`
- `UIViewControllerAnimatedOptions.CurveEaseIn`
- `UIViewControllerAnimatedOptions.CurveEaseOut`
- `UIViewControllerAnimatedOptions.CurveLinear`

`UIViewControllerAnimatedOptions.AllowUserInteraction`

By default, views cannot be interacted with when animating. Specifying this option will override the default. This can be useful for repeating animations, such as a pulsing view.

`UIViewControllerAnimatedOptions.Repeat`

This will repeat the animation indefinitely. This is often paired with the `UIViewControllerAnimatedOptionAutoreverse` option.

`UIViewControllerAnimatedOptions.Autoreverse`

This will run the animation forward and then backward, returning the view to its initial state.

Be sure to check out the *Constants* section of the *UIView Class Reference* to see all of the possible options. We will look at a few more later in this chapter.

Keyframe Animations

The animations you have added so far have been basic animations; they animate from one value to another value. If you want to animate a view's properties through more than two values, you use a *keyframe animation*. A keyframe animation can be made up of any number of individual keyframes (Figure 16.3). You can think of keyframe animations as multiple basic animations going back to back.

Figure 16.3 Keyframe animation



Keyframe animations are set up similarly to basic animations, but each keyframe is added separately. To create a keyframe animation, use the `animateKeyframesWithDuration(_:delay:options:animations:completion:)` class method on `UIView`, and add keyframes in the animation closure using the `addKeyframeWithRelativeStartTime(_:relativeDuration:animations:)` class method.

An animation that benefits from keyframe animations is spinning, or rotating, a view. If you try to use a basic animation by asking it to rotate 360°, the view will essentially respond by saying “I'm already rotated to the final location, so I don't need to do anything”. So to accomplish this, you will rotate the label in steps, each step being a portion of the complete rotation.

In `HypnosisViewController.swift`, update `viewDidAppear(_:)` to set up the keyframe animation.

```
override func viewDidAppear(animated: Bool) {
    super.viewDidAppear(animated)

    // Animate the alpha to 1.0
    UIView.animateWithDuration(1.0,
        delay: 0.0,
        options: .CurveEaseIn,
        animations: {
            self.textLabel.alpha = 1.0
        },
        completion: nil)

    // Set up the keyframe animation
    UIView.animateKeyframesWithDuration(2.0,
        delay: 0.0,
        options: nil,
        animations: {

    },
    completion: nil)
}
```

Keyframe animations are created using

`animateKeyframesWithDuration(_:delay:options:animations:completion:)`. The parameters are all the same as with the basic animation except that the options are of type `UIViewKeyframeAnimationOptions` instead of `UIViewAnimationOptions`. The duration passed into this method is the duration of the entire animation.

Then add three keyframe animations to rotate the label. Why three steps instead of two? Each keyframe animation interpolates from its initial value to its final value. With two keyframes, each animation rotates half-way, so each does not know whether to rotate clockwise or counter-clockwise. The result is that the label might rotate clockwise for half of the animation, and then reverse and animate counter-clockwise for the other half. To remove this ambiguity, you will split the animation into thirds.

Add the three keyframe animations in `viewDidAppear(_:)`.

```

UIView.animateKeyframesWithDuration(2.0,
    delay: 0.0,
    options: nil,
    animations: {

        // Add a keyframe
        UIView.addKeyframeWithRelativeStartTime(0.0,
            relativeDuration: 0.4,
            animations: {
                self.textLabel.transform = CGAffineTransformMakeScale(0.5, 2.0)
            })

        // Each keyframe begins when the previous ends
        UIView.addKeyframeWithRelativeStartTime(0.4,
            relativeDuration: 0.3,
            animations: {
                self.textLabel.transform = CGAffineTransformMakeScale(2.0, 0.5)
            })

        UIView.addKeyframeWithRelativeStartTime(0.7,
            relativeDuration: 0.4,
            animations: {
                self.textLabel.transform = CGAffineTransformMakeScale(1.0, 1.0)
            })
    },
    completion: nil)

```

Individual keyframes are added using

`addKeyframeWithRelativeStartTime(_:relativeDuration:animations:)`. The first argument is the relative start time, which will be a value between 0 and 1. The second argument is the relative duration, which is a percent of the total duration and will also be a value between 0 and 1. The first keyframe starts 0% into the animation (a relative start time of `0.0`) and will last 40% of the total duration (a relative duration of `0.4`). The second keyframe starts 40% into the total duration (a relative start time of `0.40`) and lasts 30% of the total duration (a relative duration of `0.30`). Finally, the last keyframe starts 70% into the total duration and last the remaining 30% of the total duration.

Within the animation block, set the text label's alpha to be 1. Any properties updated within the animation block, but outside of any specific keyframes, will be animated over the entire duration of the keyframe animation.

```

UIView.animateKeyframesWithDuration(2.0,
    delay: 0.0,
    options: nil,
    animations: {

        // Animate the alpha over the entire animation duration
        self.textLabel.alpha = 1

        // Add a keyframe
        UIView.addKeyframeWithRelativeStartTime(0.0,
            relativeDuration: 0.4,
            animations: {
                self.textLabel.transform = CGAffineTransformMakeScale(0.5, 2.0)
            })

        // Each keyframe begins when the previous ends
        UIView.addKeyframeWithRelativeStartTime(0.4,
            relativeDuration: 0.3,
            animations: {
                self.textLabel.transform = CGAffineTransformMakeScale(2.0, 0.5)
            })

        UIView.addKeyframeWithRelativeStartTime(0.7,
            relativeDuration: 0.4,
            animations: {
                self.textLabel.transform = CGAffineTransformMakeScale(1.0, 1.0)
            })
    },
    completion: nil)

```

Build and run the application and enter some text. The labels will stretch and scale while fading into view.

Animation Completion

It can often be useful to know when an animation completes. For instance, you might want to chain different kinds of animations together or update another object when the animation completes. To know when the animation finishes, pass a closure for the completion argument.

Update `viewDidAppear(_:`) so that it logs a message to the console when the animations complete.

```
UIView.addKeyframeWithRelativeStartTime(0.7,  
    relativeDuration: 0.4,  
    animations: {  
        self.textLabel.transform = CGAffineTransformMakeScale(1.0, 1.0)  
    })  
,  
completion: {  
    success in  
    println("Keyframe animation completed")  
})
```

Build and run the application, and log messages will appear in the console as soon as the animation completes.

You might be wondering, “What if the animation repeats? Will the completion closure be executed after each repeat?” No, the completion closure will only be executed once, at the very end.

Bronze Challenge: Spring Animations

iOS has a powerful physics engine built in. An easy way to harness this power is by using a spring animation.

```
// UIView  
  
class func animateWithDuration(duration: NSTimeInterval,  
    delay: NSTimeInterval,  
    usingSpringWithDamping dampingRatio: CGFloat,  
    initialSpringVelocity velocity: CGFloat,  
    options: UIViewAnimationOptions,  
    animations: () -> Void,  
    completion: ((Bool) -> Void)?)
```

Use this method to have the label drop in from the top of the screen. Refer to the `UIView` documentation to understand each of the arguments.

Silver Challenge: Improved Quiz

Add some animations to the Quiz app that you worked on in Chapter 1.

When a new question or answer is shown, it should fly in from the left side of the screen, animating its opacity from 0 to 1 on the way. The old question or answer should fly off the right side of the screen, losing its opacity as it goes.

Tinker with timings and animation curves to make it look good.

17

UIStoryboard

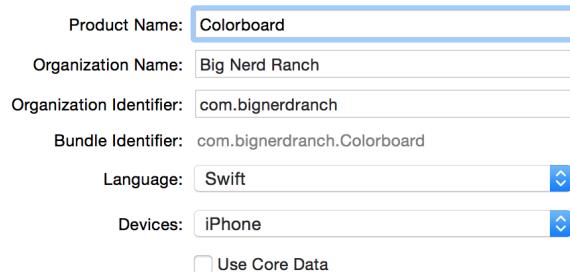
In your projects so far, you have laid out the interfaces of your view controllers in separate XIB files and then instantiated the view controllers programmatically. In this chapter, you will use a *storyboard* instead. Storyboards are a feature of iOS that allows you to instantiate and lay out all of your view controllers in one XIB-like file. Additionally, you can wire up view controllers in the storyboard to dictate how they get presented to the user.

The purpose of a storyboard is to minimize some of the simple code a programmer has to write to create and set up view controllers and the interactions between them. In this chapter, you will create an application that allows you to create a color palette of your favorite colors. You will use storyboards to see how it can simplify application control flow and remove some common repetitive code.

Creating a Storyboard

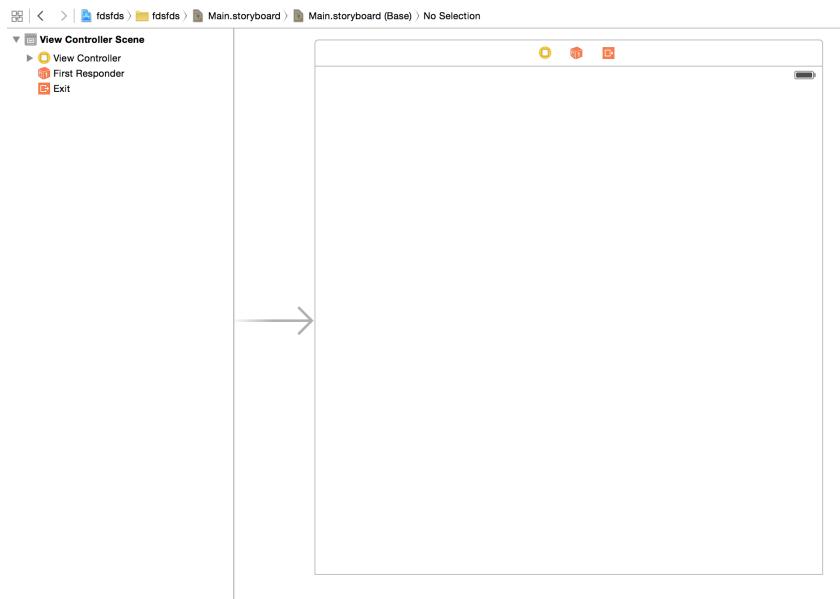
Create a new iOS Single View Application and name it Colorboard (Figure 17.1).

Figure 17.1 Creating Colorboard



The Single View Application template (and indeed most of the templates) start with a storyboard already created for you. In the Project navigator, find and open `Main.storyboard`.

Figure 17.2 Storyboard Canvas



A storyboard has a number of scenes, each representing a view controller paired with a view. You can see the View Controller Scene with its one blank white View. Select the scene either on the canvas or in the document outline and open its Attribute inspector. Under the View Controller heading, you will see a box named Is Initial View Controller. Each storyboard has an initial view controller that representing the starting point for the user interface. There is only one storyboard file in this project, and so therefore this also representing the starting point for the entire application.

How does the application know to load in this storyboard file? You might recall from previous chapters that there is a Main Interface option in the project settings.

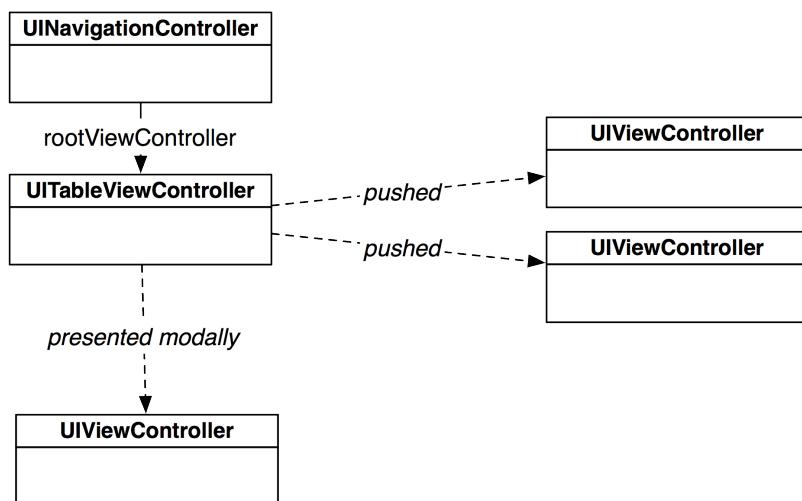
From the Project navigator, select the Colorboard project. Then under General settings, find Main Interface. You will see that it is set to Main.

When an application has a main storyboard file, it will automatically load that storyboard when the application launches. In addition to loading the storyboard and its view controllers, it will also create a window and set the initial view controller of the storyboard as the root view controller of the window. You can tell which view controller is the initial view controller by looking at the canvas in the storyboard file – the initial view controller has an arrow that fades in as it points to it.

Open `Main.storyboard` again. You will set up the scenes from scratch, so go ahead and select the View Controller Scene and delete it.

The Colorboard application will have a total of five view controllers, including a **UINavigationController** and a **UITableViewViewController**. Figure 17.3 shows an object diagram for Colorboard.

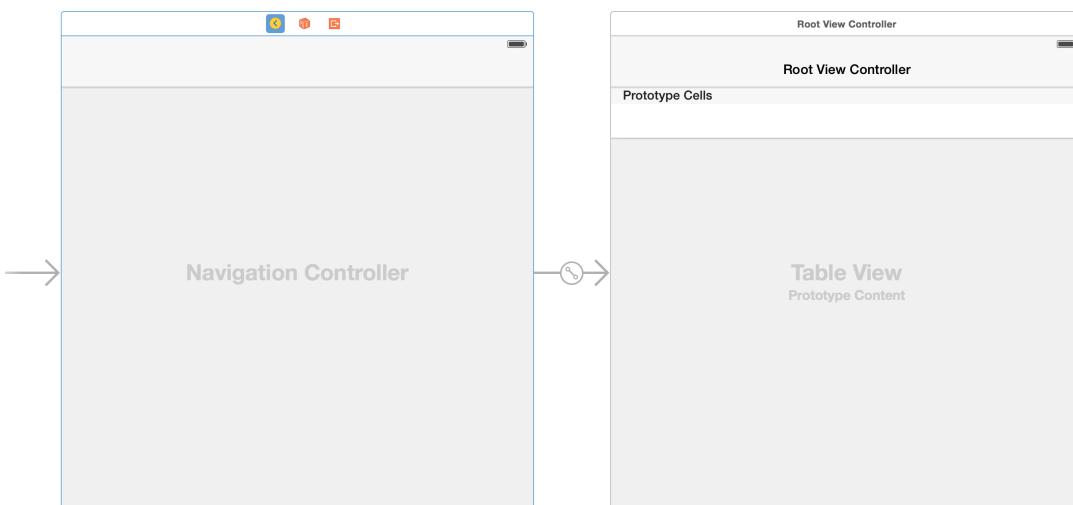
Figure 17.3 Object diagram for Colorboard



Using a storyboard, you can set up the relationships shown in Figure 17.3 without writing any code.

To get started, open the utility area and the Object Library. Drag a Navigation Controller onto the canvas. Select the Navigation Controller and open its Attributes inspector. Check the box for **Is Initial View Controller**. The canvas will now look like Figure 17.4.

Figure 17.4 Navigation controller in storyboard

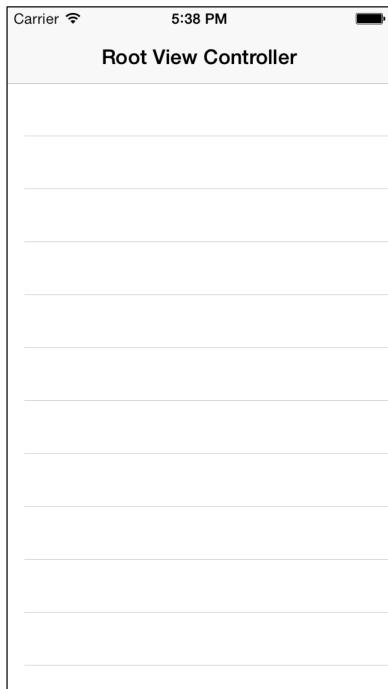


In addition to the **UINavController** object you asked for, the storyboard took the liberty of creating three other objects: the view of the navigation controller, a **UITableViewController**, and the view of the **UITableViewController**. In addition, the **UITableViewController** has been made the root view controller of the navigation controller.

The two view controller instances are represented by the white bars on the canvas, and their views are shown directly below them. You configure the view the same as you would in a normal XIB file. To configure the view controller itself, you select the first icon (a yellow circle with an arrow) on the white bar.

Build and run the application, and you will see a view of a view controller and a navigation bar that says **Root View Controller** (Figure 17.5). All of this came from the storyboard file – you did not have to write any code.

Figure 17.5 Initial Colorboard screen

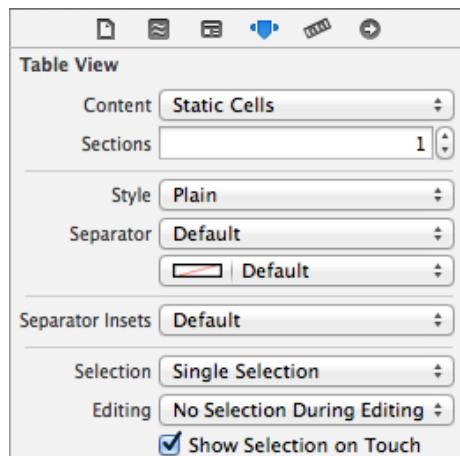


UITableViewController in Storyboards

When using a **UITableViewController**, you typically implement the appropriate data source methods to return the content of each cell. This makes sense when you have dynamic content – like a list of items that may change – but it is a lot of work when you have a table whose content never changes. Storyboards allow you to add static content to a table view without having to implement the data source methods.

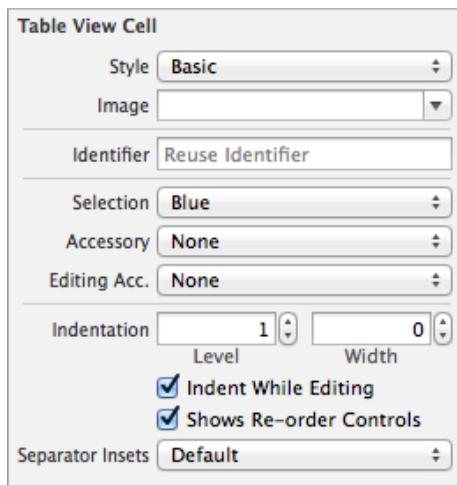
Select the Table View of the **UITableViewController** (This might be easier to do from the document outline.). In the attributes inspector, change the Content pop-up menu to Static Cells (Figure 17.6).

Figure 17.6 Static cells



Three cells will appear on the table view. You can now select and configure each one individually. Select the top-most cell and, in the attributes inspector, change its Style to Basic (Figure 17.7).

Figure 17.7 Basic **UITableViewCell**

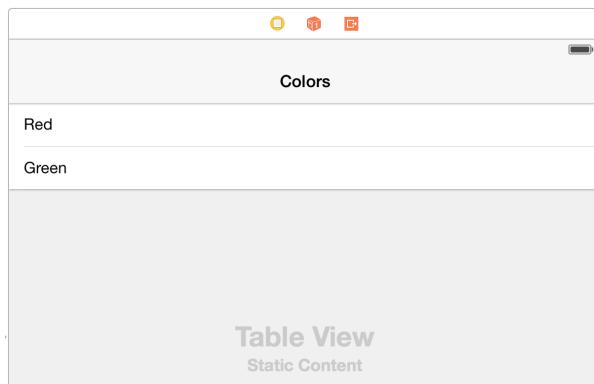


Back on the canvas, the selected cell will now say `Title`. Double-click on the text and change it to `Red`.

Repeat the same steps for the second cell, but have the title read `Green`. Let's get rid of the third cell; select it and press `Delete`.

Finally, select the navigation bar – the area above the first cell. This is present because the table view controller is embedded in a navigation controller. In the `Attributes Inspector`, change its title to `Colors`. Figure 17.8 shows the updated table view.

Figure 17.8 Configured cells



Build and run the application. You will see exactly what you have laid out in the storyboard file – a table view underneath a navigation bar. The table view is titled `Colors` and has two cells that read `Red` and `Green`. And you did not have to write any data source methods or configure a navigation item.

Segues

Most iOS applications have a number of view controllers that users navigate between. Storyboards allow you to set up these interactions as *segues* without having to write code.

A segue moves another view controller's view onto the screen when triggered and is represented by an instance of **UIStoryboardSegue**. Each segue has a style, an action item, and an identifier. The *style* of a segue determines how the view controller will be presented, such as pushed onto the stack or presented modally. The *action item* is the view object in the storyboard file that triggers the segue, like a button, a bar button item, or another **UIControl**. The *identifier* is used to programmatically access the segue. This is useful when you want to trigger a segue that does not come from an action item, like a shake or some other interface element that cannot be set up in the storyboard file.

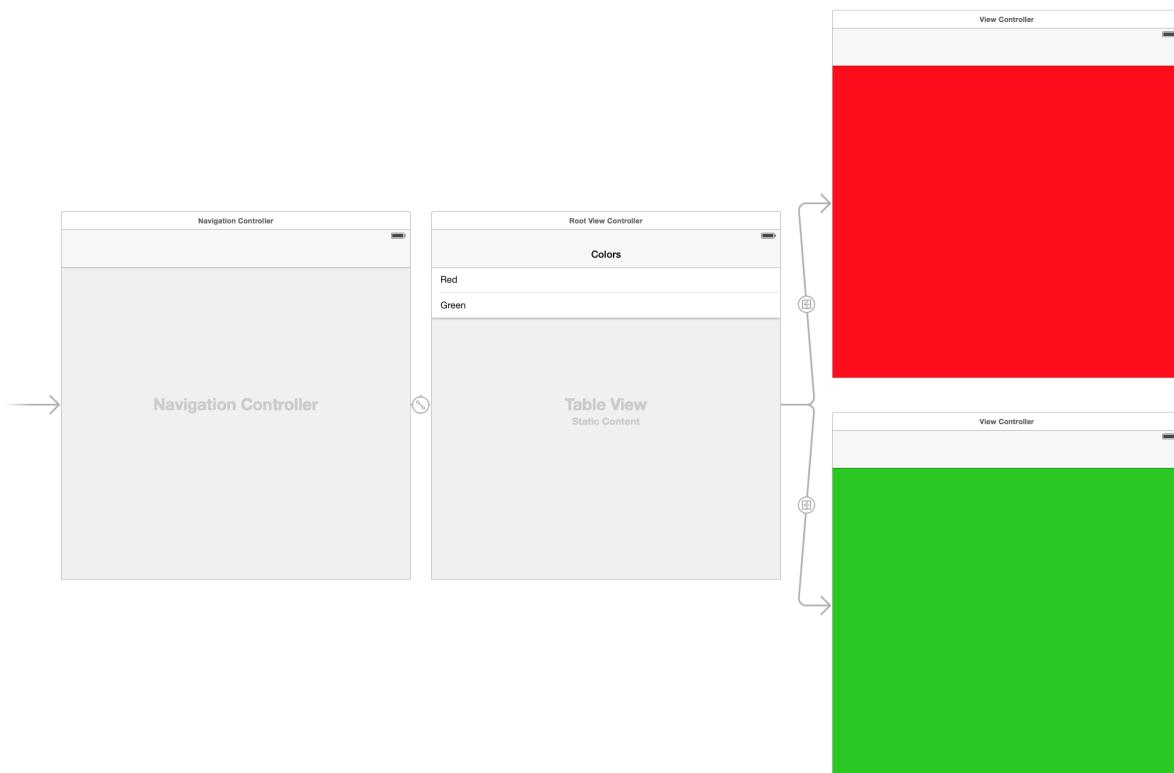
Let's start with two push segues. A push segue pushes a view controller onto the stack of a navigation controller. You will need to set up two more view controllers in your storyboard, one whose view's background is red, and the other, green. The segues will be between the table view controller and these two new view controllers. The action items will be the table view's cells; tapping a cell will push the appropriate view controller onto the navigation controller's stack.

Drag two **UIViewController** objects onto the canvas. Select the **View** of one of the view controllers and, in the attributes inspector, change its background color to red. Do the same for the other view controller's view to set its background color to green.

Next, select the cell titled Red. Control-drag to the view controller whose view has the red background. A black panel titled **Storyboard Segues** will appear. This panel lists the possible styles for this segue. Select **Show**.

Then, select the Green cell and Control-drag to the other view controller. Your canvas should look like Figure 17.9.

Figure 17.9 Setting up two segues



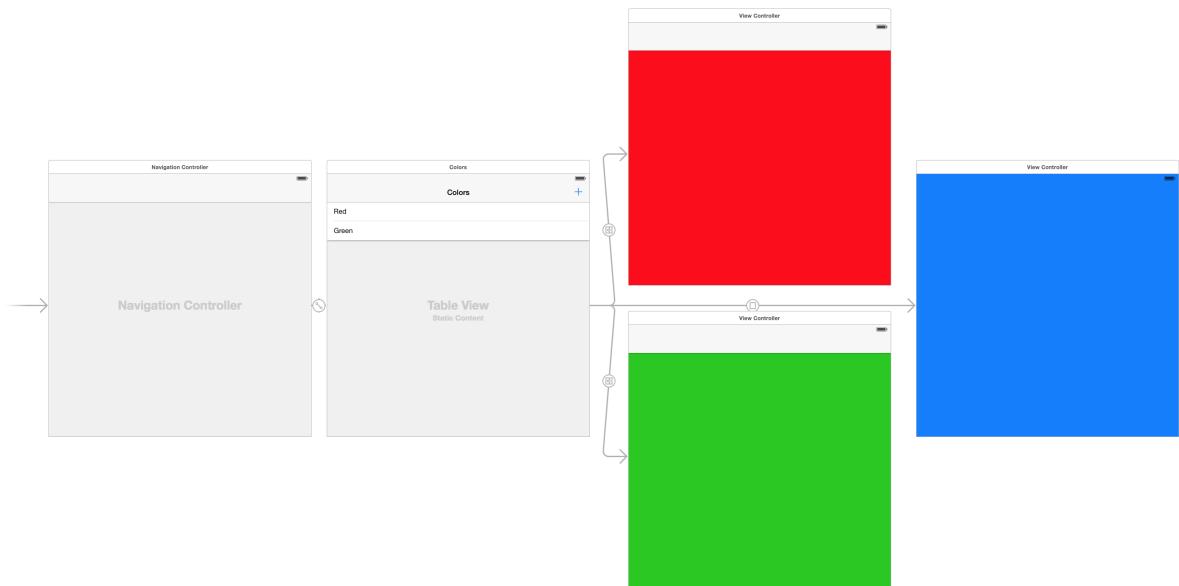
Notice the arrows that come from the table view controller to the other two view controllers. Each of these is a segue. The icon in the circle tells you that these segues are show segues.

Build and run the application. Tap on each row, and you will be taken to the appropriate view controller. You can even move back in the navigation stack to the table view controller like you would expect. The best part about this? You have not written any code yet.

Now let's look at another style of segue – a **Modal** segue. Drag a new **UIViewController** onto the canvas. Set its view's background color to blue. You want this segue's action item to be a bar button item on the table view controller's navigation item.

Drag a **Bar Button Item** from the library onto the right corner of the navigation bar at the top of the table view controller's view. In the attributes inspector, change its **Identifier** to **Add**. Then, Control-drag from this bar button item to the view controller you just dropped on the canvas. Select **present modally** from the black panel. The storyboard canvas now looks like Figure 17.10. (Notice that the icon for the modal segue is different from the icon for the push segues.)

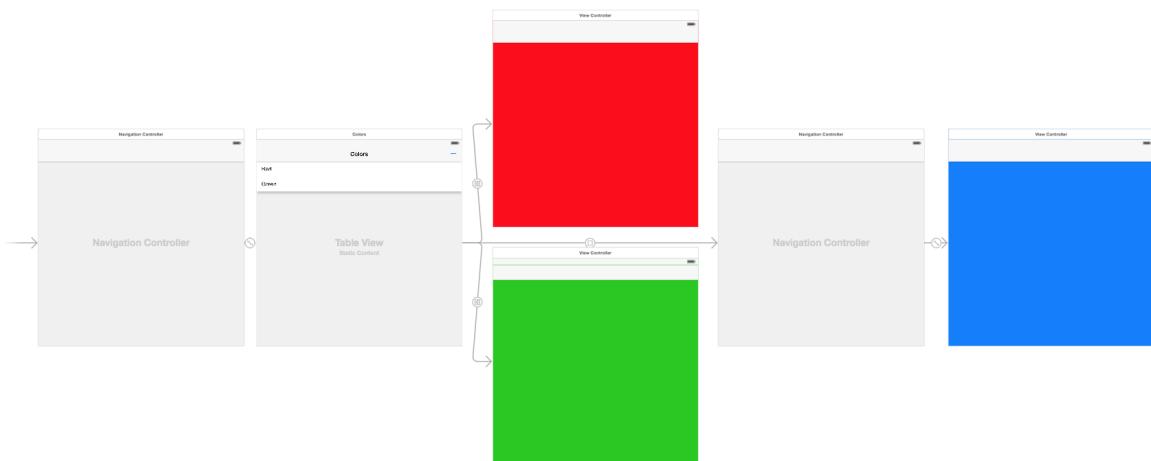
Figure 17.10 A modal segue



Build and run the application. Tap the bar button item, and a view controller with a blue view will slide onto the screen. All is well – except you cannot dismiss this view controller.

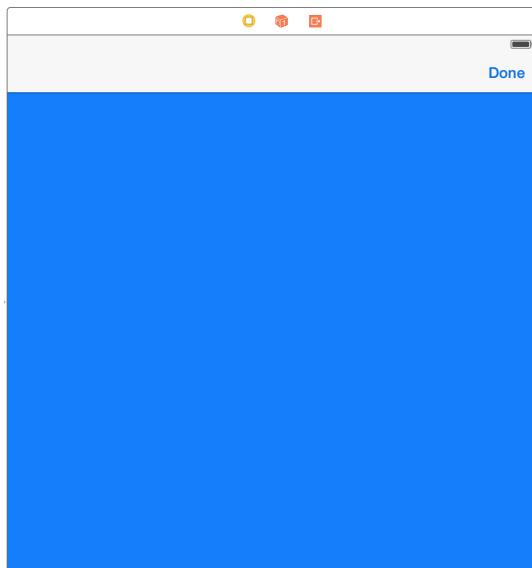
You will dismiss the view controller from a **UIBarButtonItem** on the navigation bar that says Done. Currently, the modal view controller that is being presented is not contained within a navigation controller, so it has no navigation bar for the bar button item. To fix this, select the modally presented View Controller. Then, from the Editor menu, select Embed In → Navigation Controller. Your storyboard should now look like Figure 17.11.

Figure 17.11 Adding in the navigation controller



Now that the modal view controller is within a navigation controller, it has a navigation bar at its top. Drag a bar button item to the right side of this navigation bar. Within the attributes inspector, change its Identifier to Done. The view controller should look like Figure 17.12.

Figure 17.12 Done button



This is as far as you can get without writing any code. You will need to write a method to dismiss the modal view controller and then connect this method to the Done button.

Right now, every view controller in the storyboard is a standard instance of **UIViewController** or one of its standard subclasses. You cannot write code for any of these as they are. To write code for a view controller in a storyboard, you have to create a subclass of **UIViewController** and specify in the storyboard that the view controller is an instance of your subclass.

Let's create a new **UIViewController** subclass to see how this works. Create a new **NSObject** subclass and name it **ColorViewController**.

In **ColorViewController.swift**, change the superclass to be **UIViewController**.

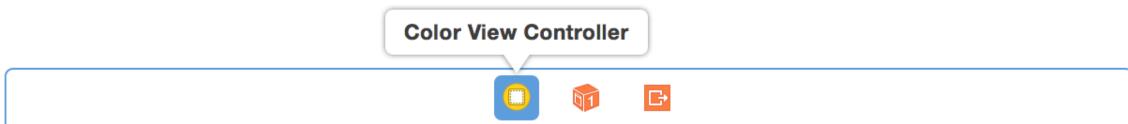
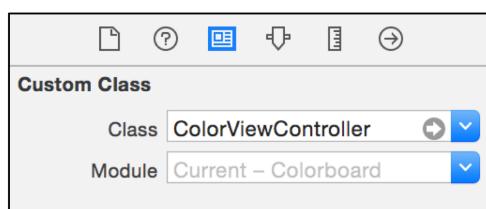
```
class ColorViewController: NSObject {
class ColorViewController: UIViewController {
```

}

Then implement a method to dismiss the view controller.

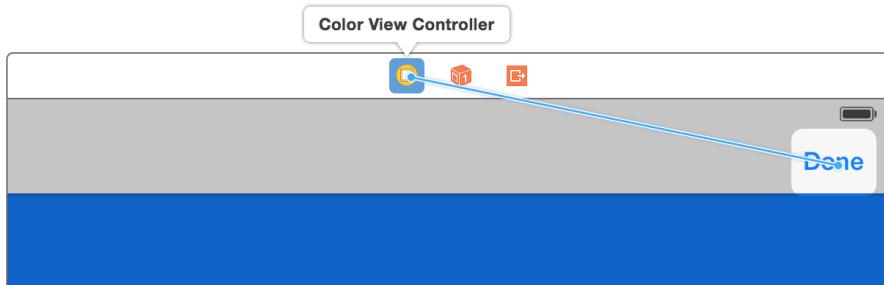
```
@IBAction func dismiss(sender: AnyObject) {
    presentingViewController?.dismissViewControllerAnimated(true, completion: nil)
}
```

Open **Main.storyboard** again. Select the modally presented (blue) view controller. (This is called the *scene dock*.) In the identity inspector, change the Class to **ColorViewController** (Figure 17.13).

Figure 17.13 Changing view controller to **ColorViewController**

Now, after making sure you are zoomed in, select the Done button. Control-drag from the button to this view controller icon and let go – when the panel appears, select the **dismiss:** method (Figure 17.14).

Figure 17.14 Setting outlets and actions in a storyboard



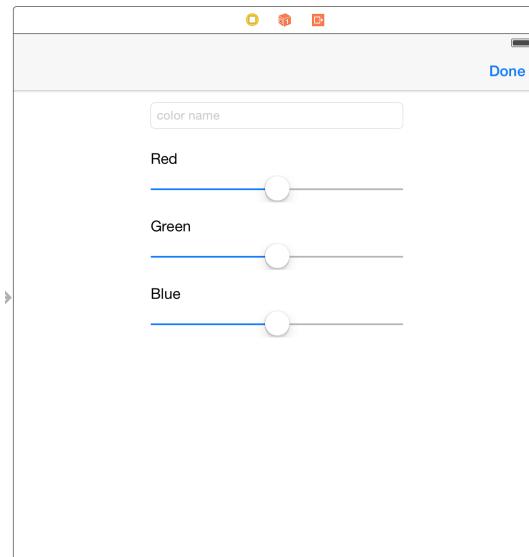
This button is now hooked up to send the message `dismiss(_:)` to its `ColorViewController` whenever tapped. Build and run the application, present the `ColorViewController`, and then tap on the Done button. Voilà!

Enabling Color Changes

You will now extend the Colorboard application to allow the user to choose a color and save it to a list of favorite colors.

Back in `Main.storyboard`, add one `UITextField`, three `UILabel` objects, and three `UISlider` objects to the view of `ColorViewController` so it looks like Figure 17.15. You'll want to add the appropriate constraints so the interface looks good on various screen sizes.

Figure 17.15 Configuring the view for `ColorViewController`



Let's have the background color of `ColorViewController`'s view match the slider values. Control-click on `ColorViewController.swift` to open it along side the storyboard file. Then, Control-drag from each of the sliders to `ColorViewController.swift` and create outlets named `redSlider`, `greenSlider`, and `blueSlider`. Do the same for the text field and name it `textField`. Your `ColorViewController` class should look like:

```
class ColorViewController: UIViewController {

    @IBOutlet weak var redSlider: UISlider!
    @IBOutlet weak var greenSlider: UISlider!
    @IBOutlet weak var blueSlider: UISlider!
    @IBOutlet weak var textField: UITextField!

    @IBAction func dismiss(sender: AnyObject) {
        presentingViewController?.dismissViewControllerAnimated(true,
            completion: nil)
    }
}
```

All three sliders will trigger the same method when their value changes. Implement this method in `ColorViewController.swift`.

```
@IBAction func changeColor(sender: AnyObject) {
    let red = CGFloat(redSlider.value)
    let green = CGFloat(greenSlider.value)
    let blue = CGFloat(blueSlider.value)

    let newColor = UIColor(red: red,
        green: green,
        blue: blue,
        alpha: 1)
    view.backgroundColor = newColor
}
```

Now open `Main.storyboard`. Control-drag from each slider to the Color View Controller and connect each to the `changeColor:` method.

Build and run the application. Moving the sliders will cause the view's background color to match.

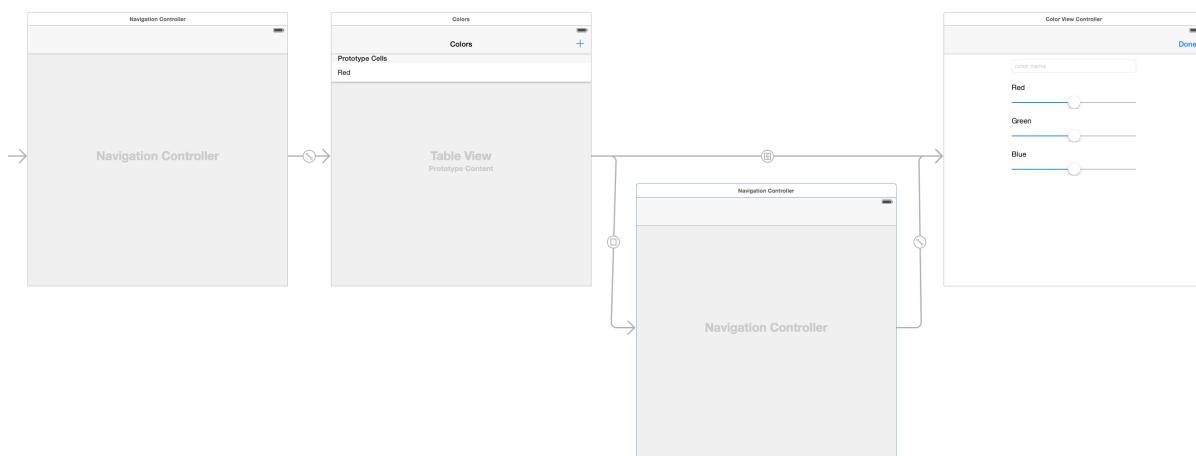
Passing Data Around

As we discussed in Chapter 7, it is often necessary for view controllers to pass data around. To show this off, you will make it so Colorboard has a list of favorite colors that can be edited by drilling down to the `ColorViewController` you just configured.

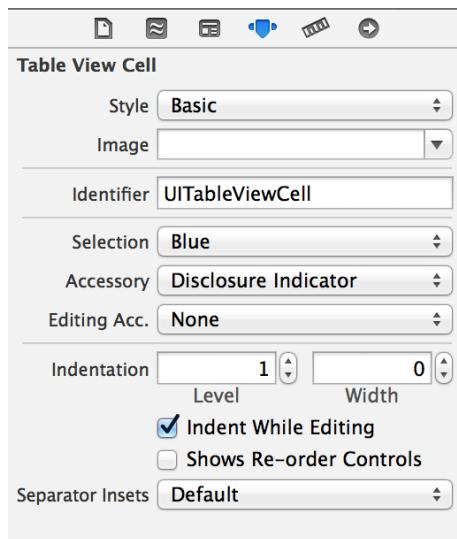
Instead of using static cells for the `UITableView`, you will go back to using *dynamic prototypes*. Because of this, you will have to implement the data source methods for the table view. Prototype cells allow you to configure the various cells you will want to return in the data source methods and assign a reuse identifier to each one.

In `Main.storyboard`, delete the Red and Green view controllers that are being pushed from the `UITableView`. Then select the Table View and open the attributes inspector. Change its Content type to Dynamic Prototypes and delete the second `UITableViewCell`. Control-drag from the remaining `UITableViewCell` to the Color View Controller and select the show segue. The storyboard should look like Figure 17.16.

Figure 17.16 Dynamic prototypes storyboard



Then select the `UITableViewCell` and set its reuse identifier to `UITableViewCell` (Figure 17.17).

Figure 17.17 **UITableViewCell** reuse identifier

In order to supply this table view controller with data for its table view, you will need to create a new **UITableViewController** subclass. Create a new **NSObject** subclass named **PaletteViewController**.

In **PaletteViewController.swift**, change the superclass to be **UITableViewController** and give the class a **colors** property.

```
class PaletteViewController: NSObject {
    class PaletteViewController: UITableViewController {

        var colors: [AnyObject] = []
    }
}
```

Next, implement **viewWillAppear(_:)** and the table view data source methods.

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)

    tableView.reloadData()
}

override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
    return colors.count
}

override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("UITableViewCell",
        forIndexPath: indexPath) as UITableViewCell

    return cell
}
```

Next, create a new **NSObject** subclass named **ColorDescription** that will represent a user-defined color.

In **ColorDescription.swift**, add properties for a **UIColor** and a name.

```
class ColorDescription: NSObject {

    var name = "Blue"
    var color = UIColor.blueColor()

}
```

To test whether the code works, let's add a new color to the **colors** array of **PaletteViewController**.

At the top of `PaletteViewController.swift`, give the `colors` array an initial `ColorDescription`. Also, change the array to hold instances of `ColorDescription` instead of `AnyObject`.

```
var colors: [ColorDescription] = [ColorDescription]()
```

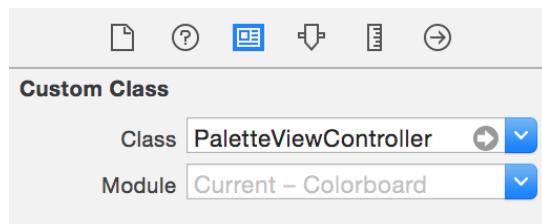
Also, update the data source method in `PaletteViewController.swift` to display the name of the color.

```
override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCellWithIdentifier("UITableViewCell",
        forIndexPath: indexPath) as UITableViewCell
    cell.textLabel?.text = colors[indexPath.row].name
    return cell
}
```

Finally, the storyboard needs to know about the new `PaletteViewController` class.

Open `Main.storyboard`. Select the Colors view controller and open its Identity inspector. Change the Class to `PaletteViewController` (Figure 17.18).

Figure 17.18 Changing the class



Build and run the application. You can drill down, but there are two problems. First, tapping on Blue will allow you to drill down, but the color information is not being passed along. Second, the view controller is currently displaying the Done button in addition to the Back button. Ideally, the Done button would only be present if you were creating a new color and presenting this view controller modally. To fix both of these issues, you will need to be able to pass data between view controllers when segues occur.

Before we move on, open `ColorViewController.swift` and add two new properties: one that determines whether you are editing a new or existing color, and another that indicates which color you are editing.

```
class ColorViewController: UIViewController {
    @IBOutlet weak var redSlider: UISlider!
    @IBOutlet weak var greenSlider: UISlider!
    @IBOutlet weak var blueSlider: UISlider!

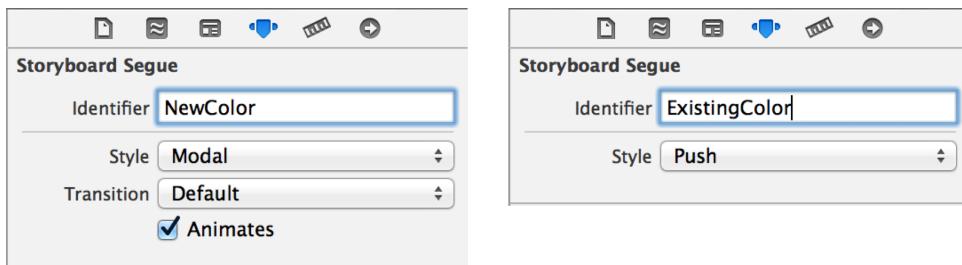
    var existingColor = false
    var currentColorDescription: ColorDescription!
```

Whenever a segue is triggered on a view controller, it gets sent the message `prepareForSegue(_:sender:)`. This method gives you both the `UIStoryboardSegue`, which gives you information about which segue is happening, and the `sender`, which is the object that triggered the segue (a `UIBarButtonItem` or a `UITableViewCell`, for example).

The segue gives you three pieces of information to use: the source view controller (where the segue is originating from), the destination view controller (where you are segueing to), and the identifier of the segue. The identifier is how you can differentiate the various segues. Let's give your two segues useful identifiers.

Open `Main.storyboard` again. Select the modal segue and open its attribute inspector. For the identifier, type in `NewColor`. Next, select the show segue and give it the identifier `ExistingColor`. The attributes inspector for both segues is shown in Figure 17.19.

Figure 17.19 Segue identifiers



With your segues identified, you can now pass your color objects around. Open `PaletteViewController.swift` and implement `prepareForSegue(_:sender:)`.

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "NewColor" {
        let newColorDescription = ColorDescription()
        colors.append(newColorDescription)

        if let nc = segue.destinationViewController as? UINavigationController {
            if let cvc = nc.topViewController as? ColorViewController {
                cvc.currentColorDescription = newColorDescription
                cvc.existingColor = false
            }
        }
    } else if segue.identifier == "ExistingColor" {
        if let row = tableView.indexPathForSelectedRow()?.row {
            let colorDescription = colors[row]

            if let cvc = segue.destinationViewController as? ColorViewController {
                cvc.currentColorDescription = colorDescription
                cvc.existingColor = true
            }
        }
    }
}
```

First the segue's identifier is checked to determine which segue is occurring. If the + button was tapped, the “NewColor” segue is triggered, so you create a new color and give it to the **ColorViewController**.

If you tap an existing color, the “ExistingColor” segue is triggered. The color that was selected is then passed to the **ColorViewController**.

(Why is the destinationViewController for “NewColor” a **UINavigationController** when it is a **ColorViewController** for “ExistingColor”? Take a look back at the storyboard file and you will notice that the modal segue presents a new **UINavigationController** whereas the push segue is pushing a view controller onto an existing navigation controller stack.)

You need to wrap up a few loose ends for the **ColorViewController**: the Done button should not be there if you are viewing an existing color, the background color and sliders need to be set up appropriately, and you need to save the new values the user has chosen when the **ColorViewController** goes away (either by dismissing the modal view controller or popping from the navigation controller stack).

Open `ColorViewController.swift` and override `viewWillAppear(_:)` to get rid of the Done button if `existingColor` is true.

```
override func viewWillAppear(animated: Bool) {
    super.viewWillAppear(animated)

    // Remove the 'Done' button if this is an existing color
    if existingColor {
        navigationItem.rightBarButtonItem = nil
    }
}
```

Then, still in `ColorViewController.swift`, override `viewDidLoad()` to set the initial background color, slider values, and color name.

```
override func viewDidLoad() {
    super.viewDidLoad()

    let color = currentColorDescription.color

    // Get the RGB values out of the UIColor object
    var red: CGFloat = 0.0
    var green: CGFloat = 0.0
    var blue: CGFloat = 0.0
    color.getRed(&red,
        green: &green,
        blue: &blue,
        alpha: nil)

    // Set the initial slider values
    redSlider.value = Float(red)
    greenSlider.value = Float(green)
    blueSlider.value = Float(blue)

    // Set the background color and text field value
    view.backgroundColor = color
    textField.text = currentColorDescription.name
}
```

Finally, save the values when the view is disappearing.

```
override func viewWillDisappear(animated: Bool) {
    super.viewWillDisappear(animated)

    // Give a default in case the view's background color is nil
    currentColorDescription.color = view.backgroundColor ?? UIColor.whiteColor()
    currentColorDescription.name = textField.text
}
```

Build and run the application and the colors should display and save correctly.

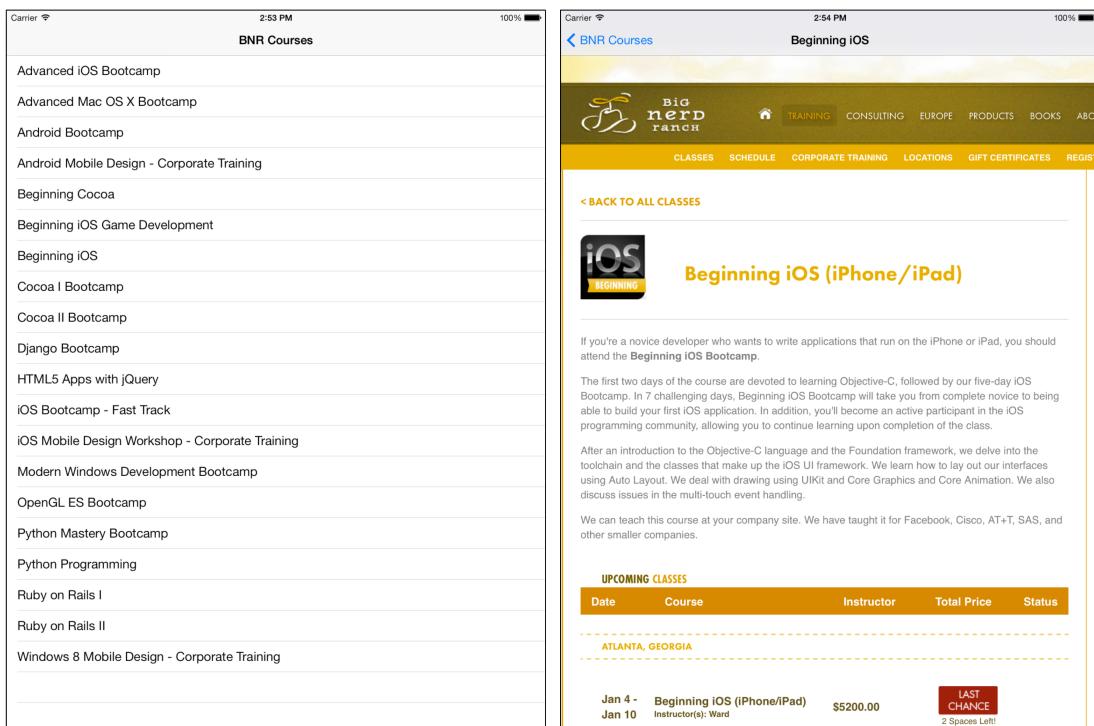
18

Web Services and WKWebView

For the next two chapters, you are going to take another break from Homeowner to work with web services and split view controllers.

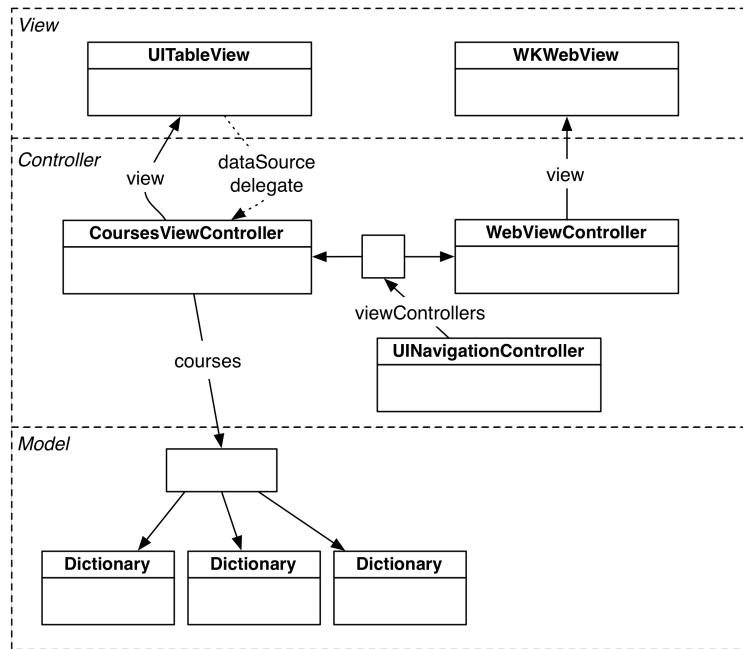
In this chapter, you will lay the foundation for an application named Nerdfeed that reads in a list of the courses that Big Nerd Ranch offers. Each course will be listed in a table view, and selecting a course will open that course's web page. Figure 18.1 shows Nerdfeed at the end of this chapter.

Figure 18.1 Nerdfeed



The work is divided into two parts. The first is connecting to and collecting data from a web service and using that data to create model objects. The second part is using the `WKWebView` class to display web content. Figure 18.2 shows an object diagram for Nerdfeed.

Figure 18.2 Nerdfeed object diagram



Web Services

Your web browser uses the HTTP protocol to communicate with a web server. In the simplest interaction, the browser sends a request to the server specifying a URL. The server responds by sending back the requested page (typically HTML and images), which the browser formats and displays.

In more complex interactions, browser requests include other parameters, like form data. The server processes these parameters and returns a customized, or dynamic, web page.

Web browsers are widely used and have been around for a long time. So the technologies surrounding HTTP are stable and well-developed: HTTP traffic passes neatly through most firewalls, web servers are very secure and have great performance, and web application development tools have become easy to use.

You can write a client application for iOS that leverages the HTTP infrastructure to talk to a web-enabled server. The server side of this application is a *web service*. Your client application and the web service can exchange requests and responses via HTTP.

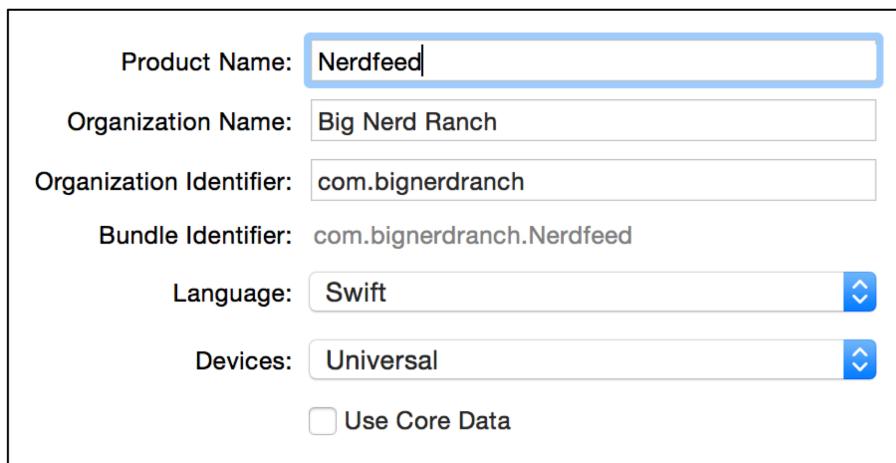
Because the HTTP protocol does not care what data it transports, these exchanges can contain complex data. This data is typically in JSON (JavaScript Object Notation) or XML format. If you control the web server as well as the client, you can use any format you like; if not, you have to build your application to use whatever the server supports.

In this chapter, you will create a client application that will make a request to the courses web service hosted at <http://bookapi.bignerdranch.com>. The data that is returned will be JSON that describes the courses.

Starting the Nerdfeed application

Create a new Single View Application for the Universal Device Family. Name this application Nerdfeed, as shown in Figure 18.3. (If you do not have an iPad to deploy to, use the iPad simulator.)

Figure 18.3 Creating a Single View Application



Let's knock out the basic UI before focusing on web services. Create a new **NSObject** subclass and name it **CoursesViewController**. In **CoursesViewController.swift**, change the superclass to **UITableViewController**.

```
class CoursesViewController: NSObject {
class CoursesViewController: UITableViewController {
```

In the Project navigator, delete the existing **ViewController.swift**.

Open **Main.storyboard** and delete the existing view controller on the canvas. From the Object library, drag a Table View Controller onto the canvas. Open its Class inspector and change the the Class to **CoursesViewController**.

Make sure the Courses View Controller is selected, and then from the Editor menu, select Embed In → Navigation Controller

Select the Navigation Controller and open its Attributes inspector. Under the View Controller heading, make sure the box for Is Initial View Controller is checked.

Finally, double-click on center of the navigation bar for the Courses View Controller and give it a title of Courses.

NSURL, NSURLRequest, NSURLSession, and NSURLSessionTask

The Nerdfeed application will fetch data from a web server using four handy classes: **NSURL**, **NSURLRequest**, **NSURLSessionTask**, and **NSURLSession** (Figure 18.4).

Figure 18.4 Relationship of web service classes



Each of these classes has an important role in communicating with a web server:

- An **NSURL** instance contains the location of a web application in URL format. For many web services, the URL will be composed of the base address, the web application you are communicating with, and any arguments that are being passed.
- An **NSURLRequest** instance holds all the data necessary to communicate with a web server. This includes an **NSURL** object as well as a caching policy, a limit on how long you will give the web server to respond,

and additional data passed through the HTTP protocol. (**NSMutableURLRequest** is the mutable subclass of **NSURLRequest**.)

- An **NSURLSessionTask** instance encapsulates the lifetime of a single request. It tracks the state of the request and has methods to cancel, suspend, and resume the request. Tasks will always be subclasses of **NSURLSessionTask**, specifically **NSURLSessionDataTask**, **NSURLSessionUploadTask**, or **NSURLSessionDownloadTask**.
- An **NSURLSession** instance acts as a factory for data transfer tasks. It is a configurable container that can set common properties on the tasks it creates. Some examples include header fields that all requests should have or whether requests work over cellular connections. **NSURLSession** also has a rich delegate model that can provide information on the status of a given task and handle authentication challenges, for example.

Formatting URLs and requests

The form of a web service request varies depending on who implements the web service; there are no set-in-stone rules when it comes to web services. You will need to find the documentation for the web service to know how to format a request. As long as a client application sends the server what it wants, you have a working exchange.

Big Nerd Ranch's courses web service wants a URL that looks like this:

`http://bookapi.bignerdranch.com/courses.json`

You can see that the base URL is `bookapi.bignerdranch.com` and the web application is located at `courses` on that server's filesystem, followed by the data format (`json`) you expect the response in.

Web service requests come in all sorts of formats, depending on what the creator of that web service is trying to accomplish. The `courses` web service, where each piece of information the web server needs to honor a request is broken up into arguments supplied as path components, is somewhat common. This particular web service call says, "Return all of the courses in a json format."

Another common web service format looks like this:

`http://baseURL.com/serviceName?argumentX=valueX&argumentY=valueY`

So, for example, you might imagine that you could specify the month and year for which you wanted a list of the courses having a URL like this:

`http://bookapi.bignerdranch.com/courses.json?year=2014&month=11`

At times, you will need to make a string "URL-safe." For example, space characters and quotes are not allowed in URLs; They must be replaced with escape-sequences. Here is how that is done.

```
let searchString = "Play some \"Abba\""  
let escapedString =  
    searchString.stringByAddingPercentEscapesUsingEncoding(NSUTF8StringEncoding)  
  
// escapedString is now "Play%20some%20%22Abba%22"
```

If you need to un-escape a percent-escaped string, **String** has the property:

```
var stringByRemovingPercentEncoding: String? { get }
```

When the request to the Big Nerd Ranch `courses` feed is processed, the server will return JSON data that contains the list of courses. The **CoursesViewController**, which made the request, will then populate its table view with the titles of these courses.

Working with **NSURLSession**

NSURLSession refers both to a specific class as well as a name for a collection of classes that form an API for network requests. To differentiate the two definitions, the book will refer to the class as just **NSURLSession**, or an instance thereof, and the API as the **NSURLSession** API.

In `CoursesViewController.swift` add a property to hold onto an instance of `NSURLSession`. Then implement `init(coder:)` to instantiate the session object.

```
class CoursesViewController: UITableViewController {

    var session: URLSession!

    required init(coder aDecoder: NSCoder) {
        super.init(coder: aDecoder)

        let config = URLSessionConfiguration.defaultSessionConfiguration()
        session = URLSession(configuration: config, delegate: nil, delegateQueue: nil)
    }
}
```

The `NSURLSession` is created with a configuration, a delegate, and a delegate queue. The defaults for these arguments are what you want for this application. You get a default configuration and pass that in for the first argument. For the second and third arguments, you simply pass in `nil` to get the defaults.

In `CoursesViewController.swift`, implement the `fetchFeed` method to create an `NSURLRequest` that connects to `bookapi.bignerdranch.com` and asks for the list of courses. Then, use the `NSURLSession` to create an `NSURLSessionDataTask` that transfers this request to the server.

```
func fetchFeed() {
    let urlString = "http://bookapi.bignerdranch.com/courses.json"
    if let url = NSURL(string: urlString) {
        let req = URLRequest(URL: url)

        let dataTask = session.dataTaskWithRequest(req) {
            (data, response, error) in
            if data != nil {
                if let jsonString = NSString(data: data, encoding: NSUTF8StringEncoding) {
                    println("\(jsonString)")
                }
            } else {
                println("Error fetching courses: \(error.localizedDescription)")
            }
        }
        dataTask.resume()
    }
}
```

Creating the `NSURLRequest` is fairly straightforward: create an `NSURL` instance and instantiate a request object with it.

The purpose of the `NSURLSession` is a bit hazier. `NSURLSession`'s job is to create tasks of a similar nature. For example, if your application had a set of requests that all required the same header fields, you could configure the `NSURLSession` with these additional header fields. Similarly, if a set of requests should not connect over cellular networks, then an `NSURLSession` could be configured to not allow cellular access. These shared behaviors and attributes are then configured on the tasks that the session creates.

The session object can now be used to create tasks. By giving the session a request and a completion block to call when the request finishes, it will return an instance of `NSURLSessionTask`. Since Nerdfeed is requesting data from a web service, the type of task will be an `NSURLSessionDataTask`. Tasks are always created in the suspended state, so calling `resume()` on the task will start the web service request. For now, the completion block will just print out the JSON data returned from the request.

Kick off the exchange whenever the `CoursesViewController` is created. In `CoursesViewController.swift`, update `init(coder:)`.

```
required init(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)

    let config = URLSessionConfiguration.defaultSessionConfiguration()
    session = URLSession(configuration: config, delegate: nil, delegateQueue: nil)

    fetchFeed()
}
```

Build and run the application. A string representation of the JSON data coming back from the web service will print to the console. (If you do not see anything print to the console, make sure you typed the URL correctly.)

JSON data

JSON data, especially when it is condensed like it is in your console, may seem daunting. However, it is actually a very simple syntax. JSON can contain the most basic objects we use to represent model objects: arrays, dictionaries, strings, and numbers. A dictionary contains one or more key-value pairs, where the key is a string, and the value can be another dictionary, string, number, or array. An array can consist of strings, numbers, dictionaries, and other arrays. Thus, a JSON document is a nested set of these types of values.

Here is an example of some really simple JSON:

```
{
    "name" : "Christian",
    "friends" : ["Aaron", "Mikey"],
    "job" : {
        "company" : "Big Nerd Ranch",
        "title" : "Senior Nerd"
    }
}
```

A JSON dictionary is delimited with curly braces ({ and }). Within curly braces are the key-value pairs that belong to that dictionary. The beginning of this JSON document is an open curly brace, which means the top-level object of this document is a JSON dictionary. This dictionary contains three key-value pairs (name, friends, and job).

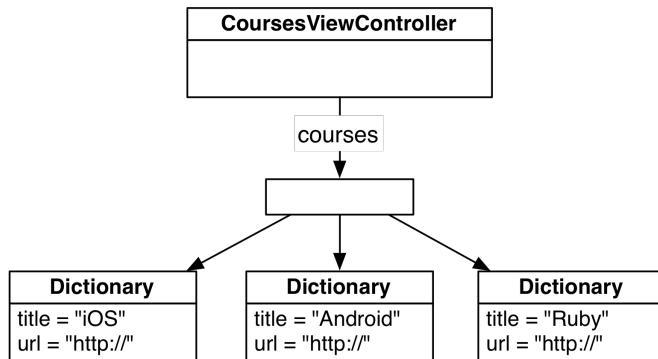
A string is represented by using text within quotations. Strings are used as the keys within a dictionary and can be used as values, too. Thus, the value of the name key in the top-level dictionary is the string Christian.

Arrays are represented with square brackets ([and]). An array can contain any other JSON information. In this case, the friends key holds an array of strings (Aaron and Mikey).

A dictionary can contain other dictionaries; the final key in the top-level dictionary, job, is associated with a dictionary that has two key-value pairs (company and title).

Nerdfeed will parse out the useful information from the JSON data and store this in its courses property.

Figure 18.5 Model object graph



Parsing JSON data

Apple has a built-in class for parsing JSON data, **NSJSONSerialization**. You can hand this class a bunch of JSON data and it will create instances of **Dictionary** for every JSON object, **Array** for every JSON array, **String** for every JSON string, and **NSNumber** for every JSON number.

In `CoursesViewController.swift`, modify the **NSURLSessionDataTask** completion handler to use the **NSJSONSerialization** class to convert the raw JSON data into the basic foundation objects.

```

let dataTask = session.dataTaskWithRequest(req) {
    (data, response, error) in

    if data != nil {
        var error: NSError?
        if let jsonObject = NSJSONSerialization.JSONObjectWithData(data,
            options: nil,
            error: &error) as? [NSObject: AnyObject] {
            println("\(jsonObject)")
        } else {
            if let error = error {
                println("Error parsing JSON: \(error)")
            }
        }

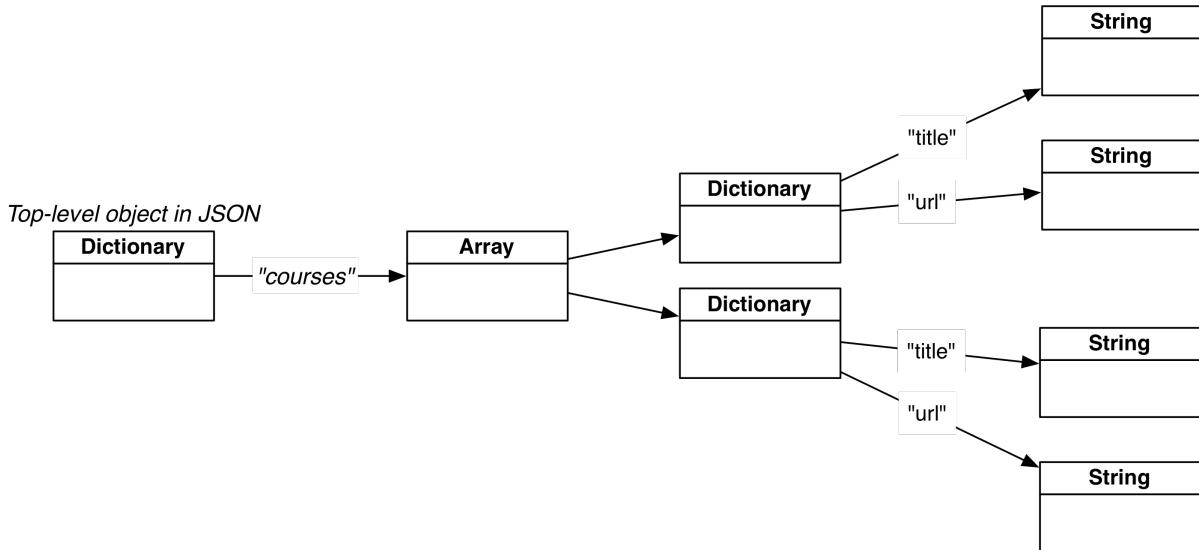
        if let jsonString = NSString(data: data, encoding: NSUTF8StringEncoding) {
            println("\(jsonString)")
        }
    } else {
        println("Error fetching courses: \(error.localizedDescription)")
    }
}

```

Build and run, then check the console. You will see the JSON data, except now it will be formatted slightly differently because `println` does a good job formatting dictionaries and arrays. The `jsonObject` is an instance of `Dictionary` and it has an `String` key with an associated value of type `Array`.

When the `NSURLSessionDataTask` finishes, you will use `NSJSONSerialization` to convert the JSON data into a `Dictionary`. Figure 18.6 shows how the data will be structured.

Figure 18.6 JSON objects



In `CoursesViewController.swift`, add a new property to hang on to that array, which is an array of `Dictionary` objects that describe each course.

```

class CoursesViewController: UITableViewController {

    var session: NSURLSession!
    var courses = [[NSObject: AnyObject]]()
}

```

Then, in the same file, change the implementation of the `NSURLSessionDataTask` completion handler:

```

if let jsonObject = NSJSONSerialization.JSONObjectWithData(data,
    options: nil,
    error: &error) as? [NSObject:AnyObject] {
    println("JSON: \(jsonObject)")

    if let courseArray: AnyObject = jsonObject["courses"] {
        if let cs = courseArray as? [[NSObject:AnyObject]] {
            self.courses = cs
            println("\(self.courses)")
        }
    }
} else {
    if let error = error {
        println("Error parsing JSON: \(error)")
    }
}

```

Now, still in CoursesViewController.swift, implement the data source methods so that each of the course titles are shown in the table.

```

override func tableView(tableView: UITableView,
    numberOfRowsInSection section: Int) -> Int {
    return courses.count
}

override func tableView(tableView: UITableView,
    cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {

    let cell = tableView.dequeueReusableCellWithIdentifier("UITableViewCell",
        forIndexPath: indexPath) as UITableViewCell
    let course = courses[indexPath.row]

    if let title: AnyObject = course["title"] {
        if let titleString = title as? String {
            cell.textLabel?.text = titleString
        }
    }

    return cell
}

```

Open Main.storyboard and select the Table View Cell. In its Attributes inspector, change the Style to Basic and set its Identifier to UITableViewCell.

The main thread

Modern iOS devices have multi-core processors that enable the devices to run multiple chunks of code concurrently. Fittingly, this is referred to as *concurrency*, and each chunk of code runs on a separate *thread*. So far in this book, all of our code has been running on the *main thread*. The main thread is sometimes referred to as the UI (user interface) thread, as any code that modifies the UI has to run on the main thread.

When the web service completes, you need to reload the table view data. By default, **NSURLSessionDataTask** runs the completion handler on a background thread. You need a way to force code to run on the main thread in order to reload the table view, and you can do that easily using the **NSOperationQueue** class.

In CoursesViewController.swift, update the completion handler to reload the table view data on the main thread.

```

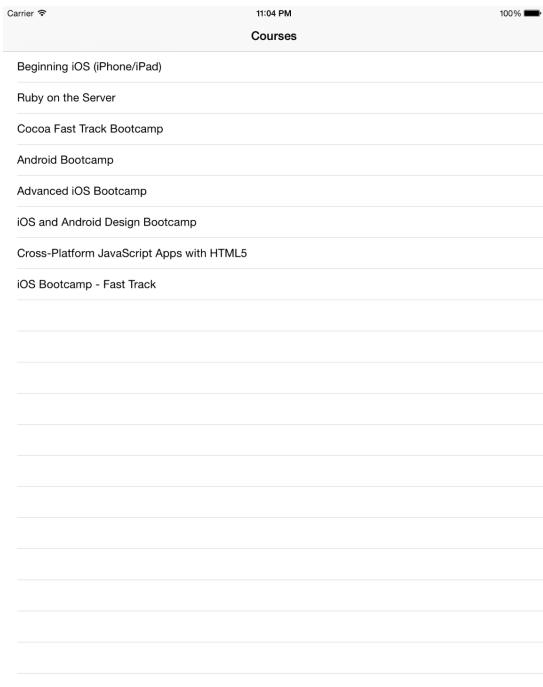
if let courseArray: AnyObject = jsonObject["courses"] {
    if let cs = courseArray as? [[NSObject:AnyObject]] {
        self.courses = cs
        println("\(self.courses)")

        NSOperationQueue.mainQueue().addOperationWithBlock() {
            self.tableView.reloadSections(NSIndexSet(index: 0),
                withRowAnimation: .Automatic)
        }
    }
}

```

Build and run the application. After the web service completes, you should see a list of Big Nerd Ranch's courses (Figure 18.7).

Figure 18.7 Course List



WKWebView

In addition to its title, each course dictionary also keeps a URL string that points to its web page. It would be neat if Nerdfeed could open up Safari to open that URL. It would be even neater if Nerdfeed could open the web page without having to leave Nerdfeed to open Safari. Good news – it can use the class **WKWebView**.

Instances of **WKWebView** render web content. In fact, the Safari application on your device uses a **WKWebView** to render its web content. In this part of the chapter, you will create a view controller whose view is an instance of **WKWebView**. When one of the items is selected from the table view of courses, you will push the web view's controller onto the navigation stack and have it load the URL string stored in the **Dictionary**.

Create a new **NSObject** subclass and name it **WebViewController**. In **WebViewController.swift**, change the superclass to **UIViewController**:

```
class WebViewController: NSObject {
class WebViewController: UIViewController {
```

Import **WebKit** and add an outlet for a web view and a property for a URL.

```
import UIKit
import WebKit

class WebViewController: UIViewController {

    var webView: WKWebView!
    var URL: NSURL?
}
```

As of Xcode 6.1, Interface Builder does not support adding instances of **WKWebView** to the canvas, so you'll set up the web view programmatically.

Override **viewDidLoad** to set up the web view and load the URL if it exists.

```

override func viewDidLoad() {
    super.viewDidLoad()

    webView = WKWebView(frame: view.bounds)
    webView.setTranslatesAutoresizingMaskIntoConstraints(false)
    view.addSubview(webView)

    // Setup web view constraints
    var formatString = "H:[webView]|"
    var constraints = NSLayoutConstraint.constraintsWithVisualFormat(formatString,
        options: nil,
        metrics: nil,
        views: ["webView": webView])
    view.addConstraints(constraints)

    formatString = "V:[webView]|"
    constraints = NSLayoutConstraint.constraintsWithVisualFormat(formatString,
        options: nil,
        metrics: nil,
        views: ["webView": webView])
    view.addConstraints(constraints)

    // Load URL if there is one
    if let urlToLoad = URL {
        let req = URLRequest(URL: urlToLoad)
        webView.loadRequest(req)
    }
}

```

Open Main.storyboard and drag a View Controller onto the canvas. Open its Identity inspector and change the Class to be `WebViewController`.

Then, Control-drag from the `UITableViewCell` to the Web View Controller and select the show selection segue. Open the Attributes inspector for this segue and give it an Identifier of `showDetail`.

In `CoursesViewController.swift`, implement `prepareForSegue(_:sender:)` to configure the `WebViewController`.

```

override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "showDetail" {
        // Make sure the destination is a WebViewController
        let wvc = segue.destinationViewController as WebViewController
        // Get the selected index path
        if let indexPath = tableView.indexPathForSelectedRow() {
            let course = courses[indexPath.row]

            if let urlString = course["url"] as? String {
                wvc.URL = NSURL(string: urlString)
            }

            if let titleString = course["title"] as? String {
                wvc.navigationItem.title = titleString
            }
        }
    }
}

```

Build and run the application. You should be able to select one of the courses, and it should take you to a new view controller that displays the web page for that course.

Credentials

When you try to access a web service, it will sometimes respond with an *authentication challenge*, which means “Who the heck are you?” You then need to send a username and password (a *credential*) before the server will send its genuine response.

When the challenge is received, the `NSURLSession` delegate is asked to authenticate that challenge, and the delegate will respond by supplying a username and password.

Open `CoursesViewController.swift` and update `fetchFeed` to hit a secure Big Nerd Ranch courses web service. (Do not forget to use https instead of http.)

```
func fetchFeed() {
    let urlString = "http://bookapi.bignerdranch.com/courses.json"
    let urlString = "https://bookapi.bignerdranch.com/private/courses.json"
    if let url = NSURL(string: urlString) {
        let req = NSURLRequest(URL: url)
```

The `NSURLSession` now needs its delegate to be set upon creation. Update `init(coder:)` to set the delegate of the session.

```
required init(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)

    let config = NSURLSessionConfiguration.defaultSessionConfiguration()
    session = NSURLSession(configuration: config, delegate: self, delegateQueue: nil)

    fetchFeed()
}
```

Then update the class declaration in `CoursesViewController.swift` to conform to the `NSURLSessionDataDelegate` protocol.

```
class CoursesViewController: UITableViewController, NSURLSessionDataDelegate {
```

Build and run the application. The web service will complete with an error (unauthorized access), and no data will be returned. Because of this, the `CoursesViewController` will not have any courses to display, and so the table view will remain empty.

To authorize this request, you will need to implement the authentication challenge delegate method. This method will supply a block that you can call, passing in the credentials as an argument.

In `CoursesViewController.swift` implement the `NSURLSessionDataDelegate` method to handle the authentication challenge.

```
func URLSession(session: NSURLSession,
               task: NSURLSessionTask,
               didReceiveChallenge challenge: NSURLAuthenticationChallenge,
               completionHandler:
                    (NSURLSessionAuthChallengeDisposition, NSURLCredential) -> Void) {

    let cred = NSURLCredential(user: "BigNerdRanch",
                               password: "AchieveNerdvana",
                               persistence: .ForSession)
    completionHandler(.UseCredential, cred)
}
```

The completion handler takes in two arguments. The first argument is the type of credentials you are supplying. Since you are supplying a username and password, the type of authentication is `NSURLSessionAuthChallengeDisposition.UseCredential`. The second argument is the credentials themselves, an instance of `NSURLCredential`, which is created with the username, password, and an enumeration specifying how long these credentials should be valid for.

Build and run the application and it will behave as it did before you changed to the secure web service, but it is now fetching the courses securely over SSL.

Silver Challenge: More WKWebView

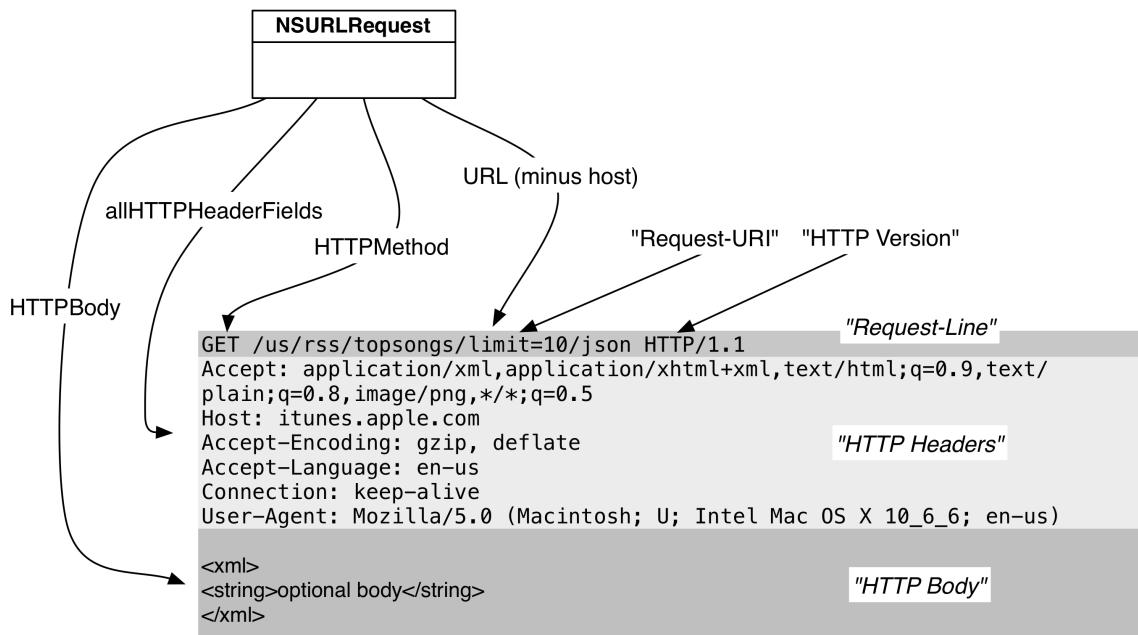
A `WKWebView` keeps its own history. You can send the messages `goBack` and `goForward` to a web view, and it will traverse through that history. Create a `UIToolbar` instance and add it to the `WebViewController`'s view

hierarchy. This toolbar should have back and forward buttons that will let the web view move through its history. Bonus: use two other properties of `WKWebView` to enable and disable the toolbar items.

For the More Curious: The Request Body

When `NSURLSessionTask` talks to a web server, it uses the HTTP protocol. This protocol says that any data you send or receive must follow the HTTP specification. The actual data transferred to the server in this chapter is shown in Figure 18.8.

Figure 18.8 HTTP request format



`NSURLRequest` has a number of methods that allow you to specify a piece of the request and then properly format it for you.

Any service request has three parts: a request-line, the HTTP headers, and the HTTP body, which is optional. The request-line is the first line of the request and tells the server what the client is trying to do. In this request, the client is trying to GET the resource at `/courses.json`. (It also specifies the HTTP specification version that the data is in.)

The command `GET` is an HTTP method. While there are a number of supported HTTP methods, you most commonly see `GET` and `POST`. The default of `NSURLRequest`, `GET`, indicates that the client wants something *from* the server. The thing that it wants is called the Request-URI (`/courses.json`).

In the early days of the web, the Request-URI would be the path of a file on the server. For example, the request `http://www.website.com/index.html` would return the file `index.html`, and your browser would render that file in a window. Today, we also use the Request-URI to specify a service that the server implements. For example, in this chapter, you accessed the `courses` service, supplied parameters to it, and were returned a JSON document. You are still GETting something, but the server is more clever in interpreting what you are asking for.

In addition to getting things from a server, you can send it information. For example, many web servers allow you to upload photos. A client application would pass the image data to the server through a service request. In this situation, you use the HTTP method `POST`, which indicates to the server that you are including the optional HTTP body. The body of a request is data you can include with the request – typically XML, JSON, or Base-64 encoded data.

When the request has a body, it must also have the `Content-Length` header. Handily enough, `NSURLRequest` will compute the size of the body and add this header for you.

Here is some code that constructs an `NSMutableURLRequest` that HTTP POSTs an image.

```
if let someURL = NSURL(string: "http://www.photos.example.com/upload") {
    let image = profileImage()
    let data = UIImagePNGRepresentation(image)

    let req = NSMutableURLRequest(URL: someURL,
        cachePolicy: .ReloadIgnoringLocalCacheData,
        timeoutInterval: 90)

    // This adds the HTTP body data and automatically sets the Content-Length header
    req.HTTPBody = data

    // This changes the HTTP Method in the request-line
    req.HTTPMethod = "POST"

    // If you wanted to set the Content-Length programmatically...
    req.setValue("\(data.length)", forHTTPHeaderField: "Content-Length")
}
```


19

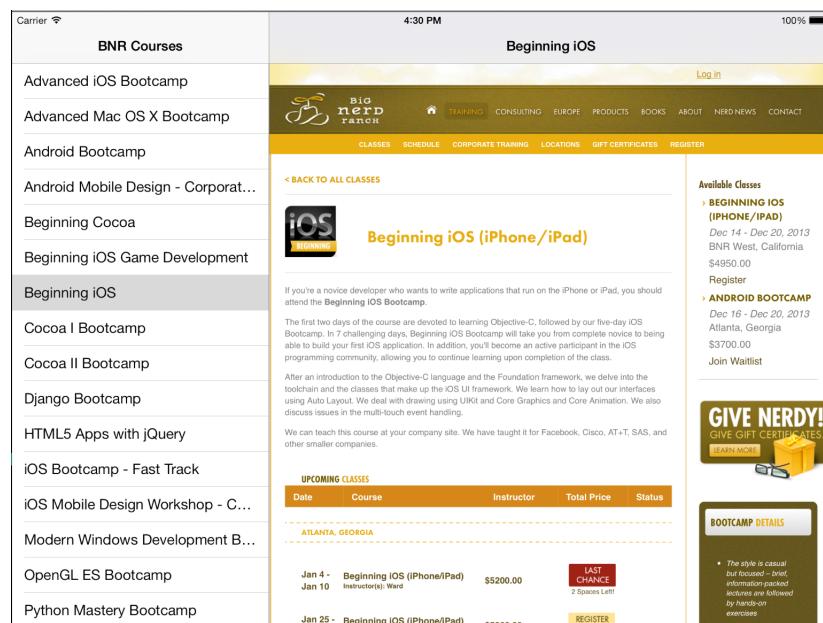
UISplitViewController

The iPhone and iPod touch have a limited amount of screen real estate. Given their small screen size, when presenting a drill-down interface, a **UINavigationController** is used to swap between a list of items and a detailed view for an item.

The iPad, on the other hand, has quite a bit more space on the screen, so it is able to take advantage of that space more effectively using a class called **UISplitViewController**. A **UISplitViewController** manages two view controllers in a master-detail relationship. On iPad, it will display both view controllers side-by-side with the master view controller on the left and the detail view controller on the right. **UISplitViewController** also works on iPhone, and it behaves almost identically to a **UINavigationController**. We refer to the **UISplitViewController** as an *adaptive view controller* because it adapts its interface depending on its enclosing trait collection.

In this chapter, you will have Nerdfeed present its view controllers in a split view controller (Figure 19.1).

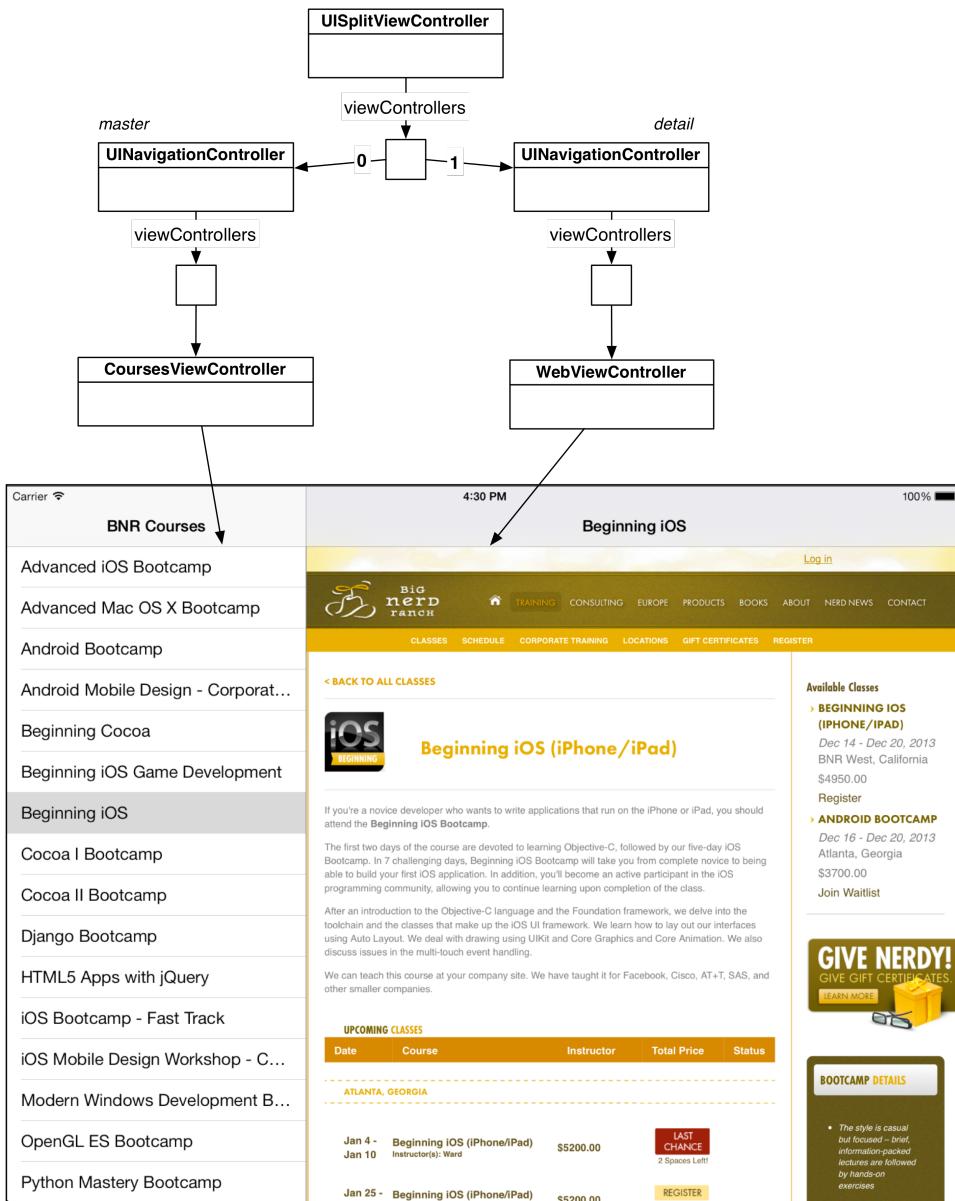
Figure 19.1 Nerdfeed with **UISplitViewController**



Splitting Up Nerdfeed

Creating a **UISplitViewController** is simple since you have already learned about navigation controllers and tab bar controllers. When you initialize a split view controller, you pass it an array of view controllers just like with a tab bar controller. However, a split view controller's array is limited to two view controllers: a master view controller and a detail view controller. The order of the view controllers in the array determines their roles in the split view; the first entry is the master view controller, and the second is the detail view controller. You will set up the split view controller hierarchy using the storyboard you worked on in the previous chapter.

Figure 19.2 Split view controller diagram



Open Nerdfeed.xcodeproj in Xcode, and then open Main.storyboard. Currently Nerdfeed is using a **`UINavigationController`** hierarchy. Let's update the hierarchy to use a **`UISplitViewController`**.

With the storyboard still open, reveal the Object library and drag a Split View Controller onto the canvas. Delete the three view controllers hooked up to the split view controller (a **`UINavigationController`**, a **`UITableViewController`**, and a **`UIViewController`**).

The split view controller needs to be set as the initial view controller for this storyboard. Select the Split View Controller and open the Attributes inspector. Check the box for **Is Initial View Controller**.

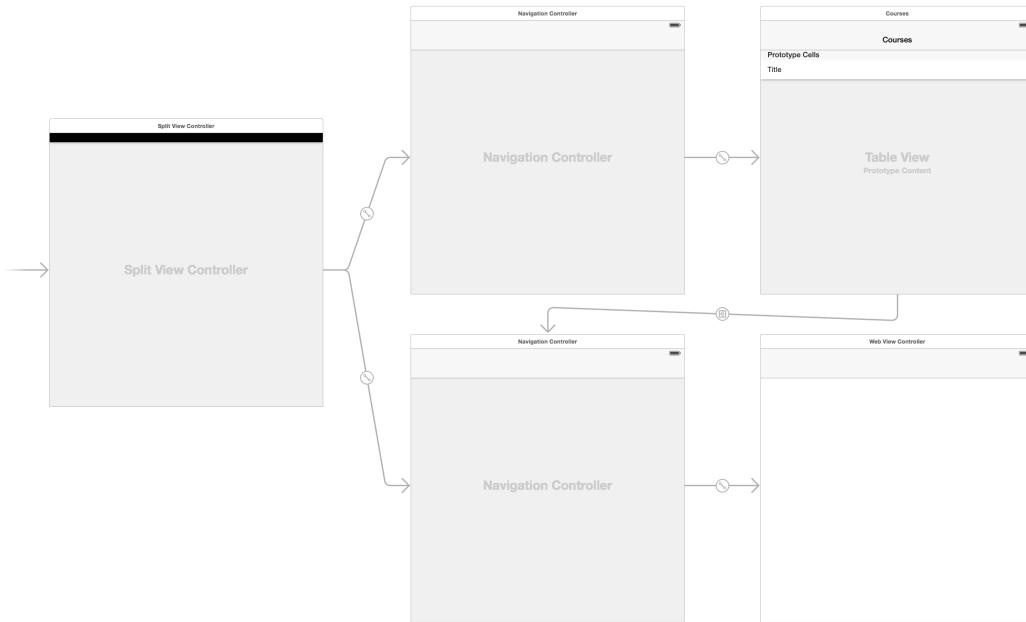
The master view controller for the split view controller will be the navigation controller with its **`CoursesViewController`**. Control-drag from the Split View Controller to the Navigation Controller and select **master view controller**.

The detail view controller will be the **`WebViewController`**, but it should be contained within a navigation controller (you'll see why later in this chapter). Select the Detail View Controller. From the Editor menu, select **Embed In → Navigation Controller**.

For split view controller, you don't push the detail view controller (which is represented by the **Show segue** type). Instead, you will use the **Show Detail segue**.

Select the segue linking the **CoursesViewController** to the navigation controller associated with the **WebViewController**. Open the Attributes inspector and change the Segue type to Show Detail. Finally, Control-drag from the Split View Controller to the newly created Navigation Controller and select detail view controller. Your storyboard canvas should look like Figure 19.3.

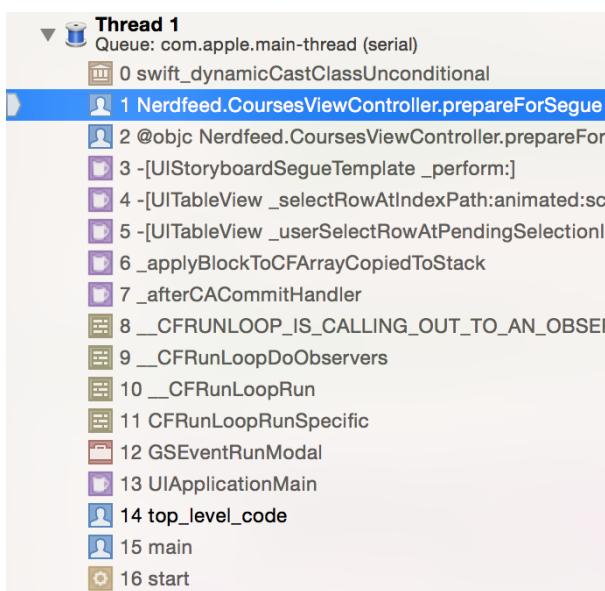
Figure 19.3 Split View Controller Storyboard



Build and run the application for iPad. You may not see anything yet if you are in portrait mode; however, if you rotate the device to landscape you will see both view controllers on the screen. This is how a **UISplitViewController** works: in landscape mode, the master view controller is shown in a small strip on the left hand side of the screen and the detail view controller takes over the rest of the screen.

Rotate the iPad into landscape and select a row in the table view. Uh oh! The application crashes. If you look at the stack trace in the Debug navigator, you will see that the application is crashing in **prepareForSegue(_:sender)** in **CoursesViewController** (Figure 19.4).

Figure 19.4 Stack Trace of Crash



In the Issue navigator, select the line that begins with **Nerdfeed.CoursesViewController.prepareForSegue**. This will take you to the line of code where the application is crashing:

```
let wvc = segue.destinationViewController as WebViewController
```

When you declared that line initially, you made a contract with the compiler. You said, “I know that when this segue happens, the destination view controller will be a **WebViewController**.” When the storyboard was updated for **UISplitViewController**, the segue’s destination view controller became the newly created navigation controller. You broke that contract and therefore when that segue executed, the cast failed and the application crashed.

Update **prepareForSegue(_:sender:)** to reflect the current storyboard scene.

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
    if segue.identifier == "showDetail" {
        let nc = segue.destinationViewController as UINavigationController
        let wvc = nc.topViewController as WebViewController
```

If a row is selected in the table view now, the application won’t crash and the **WebViewController** will display the course information.

Now build and run the application on iPhone. Notice that it behaves much like navigation controller with one exception - the application starts by displaying the **WebViewController**. This is because it is behaving like split view controller always has, showing the detail view controller. For Homeowner, it should behave just like a navigation controller by showing the master view controller (**ItemsViewController**) initially.

On iPhone, it makes sense to collapse that view controller so we see the primary view controller on launch. You can do this with a delegate method on **UISplitViewController**.

Open **AppDelegate.swift** and set the delegate for the **UISplitViewController**. Also declare that **AppDelegate** conforms to **UISplitViewControllerDelegate**.

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate, UISplitViewControllerDelegate {

    var window: UIWindow?

    func application(application: UIApplication,
                    didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
        let svc = window!.rootViewController as UISplitViewController
        svc.delegate = self
        return true
    }
}
```

Finally, implement the delegate method to collapse the secondary view controller, which you’ll do if the **WebViewController** does not have a URL.

```
func splitViewController(splitViewController: UISplitViewController,
                      collapseSecondaryViewController secondaryViewController: UIViewController!,
                      ontoPrimaryViewController primaryViewController: UIViewController!) -> Bool {
    let nc = secondaryViewController as UINavigationController
    let wvc = nc.topViewController as WebViewController
    if wvc.URL == nil {
        return true
    } else {
        return false
    }
}
```

Build and run on iPhone and you’ll land at the **CoursesViewController** as expected.

Let’s take a detour and look at iPhone 6 Plus in more detail. If you haven’t already ran the application on iPhone 6 Plus, go ahead and do that now. Notice that it currently behaves just like the other iPhones. Currently, the application doesn’t know that it is built with the iPhone 6 Plus in mind, so it is scaling the traditional iPhone interface up to fill the screen. To get it working natively for iPhone 6 Plus, the application needs an iPhone 6 Plus launch image. You can add one to the project as you always have, but let’s take a look at a new solution: using the storyboard to generate one for us.

Apple's human interface guidelines says, "Design a launch image that is identical to the first screen of the app ... to enhance the user's perception of your app as quick to launch and immediately ready for use." With many different screen sizes and orientations to support, it would be cumbersome to manually design each of these; however, since the storyboard knows the initial interface will be laid out in every permutation, it can do the heavy lifting for you.

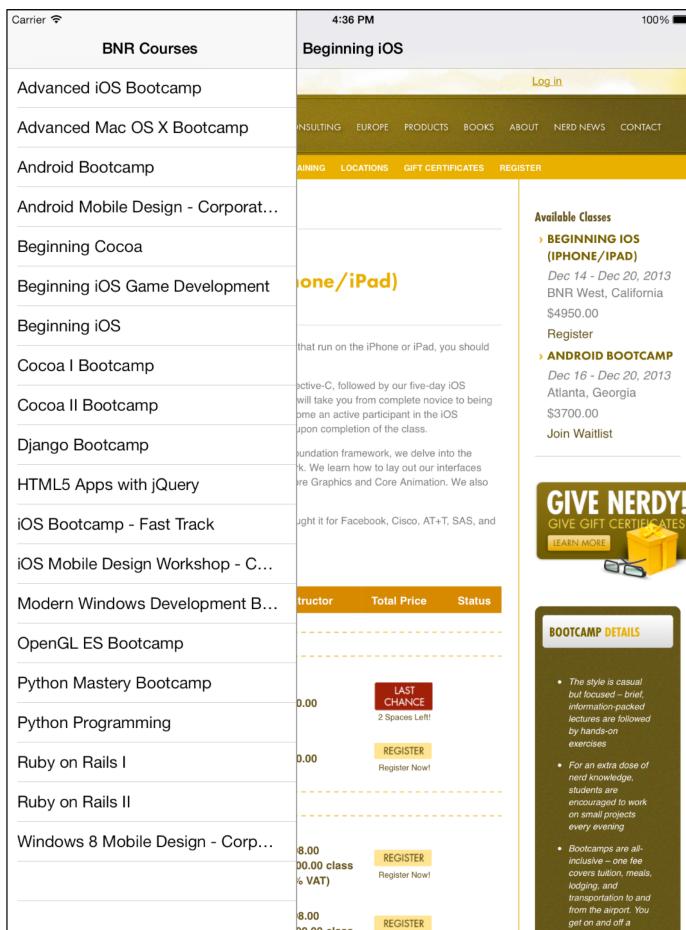
Open the project settings by clicking on Nerdfeed in the Project navigator. Under App Icons and Launch Images, choose Main from the Launch Screen File drop-down. Build and run the application on iPhone 6 Plus and notice the split that occurs in landscape. In portrait, iPhone 6 Plus behaves just as it does on other iPhones. This makes sense since all iPhones have a compact width in portrait orientation.

Displaying the Master View Controller in Portrait Mode

While in portrait mode, the master view controller is missing in action. It would be nice if you could see the master view controller to select a new post from the list without having to rotate the device.

UISplitViewController lets you do just that by supplying its delegate with a **UIBarButton Item**. Tapping this button shows the master view controller in a specialized **UIPopoverController** (Figure 19.5).

Figure 19.5 Master view controller in **UIPopoverController**



```
func application(application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?) -> Bool {
    if let svc = window!.rootViewController as? UISplitViewController {
        svc.delegate = self
        let nc = svc.viewControllers[1] as UINavigationController
        let wvc = nc.topViewController as WebViewController
        wvc.navigationItem.leftBarButtonItem = svc.displayModeButtonItem()
    }
    return true
}
```

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {  
    if segue.identifier == "showDetail" {  
        let nc = segue.destinationViewController as UINavigationController  
        let wvc = nc.topViewController as WebViewController  
  
        wvc.navigationItem.leftBarButtonItem =  
            splitViewController?.displayModeButtonItem()  
        wvc.navigationItem.leftItemsSupplementBackButton = true
```

Build and run the application. Rotate to portrait mode, and you will see the bar button item appear on the left of the navigation bar. Tap that button, and the master view controller's view will appear in a **UIPopoverController**.

This bar button item is why we always had you put the detail view controller inside a navigation controller. You do not have to use a navigation controller to put a view controller in a split view controller, but it makes using the bar button item much easier. (If you do not use a navigation controller, you can instantiate your own **UINavigationBar** or **UIToolbar** to hold the bar button item and add it as a subview of the **WebViewController**'s view.)

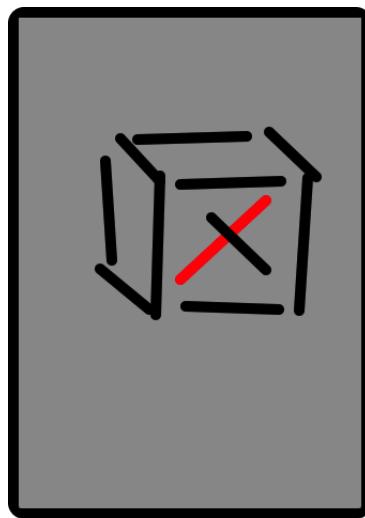
20

Touch Events and UIResponder

For the next three chapters, you are going to step away from Homeowner and build a new application named TouchTracker to learn more about touch events and gestures, as well as debugging applications.

In this chapter, you will create a view that lets the user draw lines by dragging across the view (Figure 20.1). Using multi-touch, the user will be able to draw more than one line at a time.

Figure 20.1 A drawing program



Touch Events

As a subclass of **UIResponder**, a **UIView** can override four methods to handle the four distinct touch events:

- a finger or fingers touches the screen

```
func touchesBegan(touches: NSSet, withEvent event: UIEvent)
```

- a finger or fingers moves across the screen (this message is sent repeatedly as a finger moves)

```
func touchesMoved(touches: NSSet, withEvent event: UIEvent)
```

- a finger or fingers is removed from the screen

```
func touchesEnded(touches: NSSet, withEvent event: UIEvent)
```

- a system event, like an incoming phone call, interrupts a touch before it ends

```
func touchesCancelled(touches: NSSet, withEvent event: UIEvent)
```

When a finger touches the screen, an instance of **UITouch** is created. The **UIView** that this finger touched is sent the message **touchesBegan(_:withEvent:)** and the **UITouch** is in the **NSSet** of touches.

As that finger moves around the screen, the touch object is updated to contain the current location of the finger on the screen. Then, the same **UIView** that the touch began on is sent the message

`touchesMoved(_:withEvent:)`. The **NSSet** that is passed as an argument to this method contains the same **UITouch** that originally was created when the finger it represents touched the screen.

When a finger is removed from the screen, the touch object is updated one last time to contain the current location of the finger, and the view that the touch began on is sent the message `touchesEnded(_:withEvent:)`. After that method finishes executing, the **UITouch** object is destroyed.

From this information, we can draw a few conclusions about how touch objects work:

- One **UITouch** corresponds to one finger on the screen. This touch object lives as long as the finger is on the screen and always contains the current position of the finger on the screen.
- The view that the finger started on will receive every touch event message for that finger no matter what. If the finger moves outside of the **UIView**'s frame that it began on, that view still receives the `touchesMoved(_:withEvent:)` and `touchesEnded(_:withEvent:)` messages. Thus, if a touch begins on a view, then that view owns the touch for the life of the touch.
- You do not have to – nor should you ever – keep a reference to a **UITouch** object. The application will give you access to a touch object when it changes state.

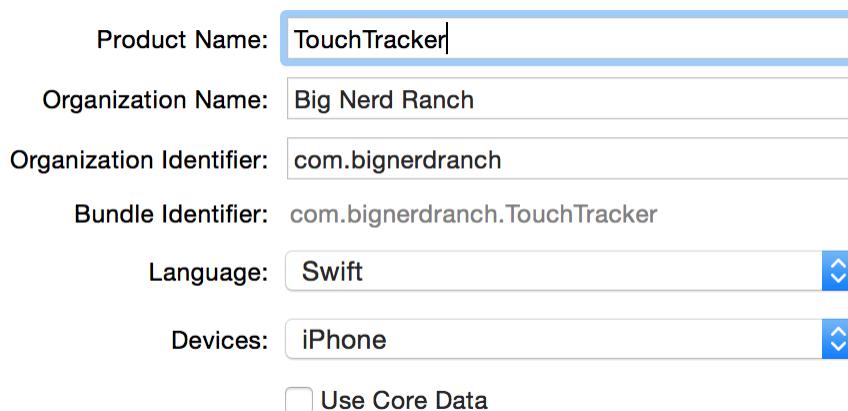
Every time a touch does something, like begins, moves, or ends, a *touch event* is added to a queue of events that the **UIApplication** object manages. In practice, the queue rarely fills up, and events are delivered immediately. The delivery of these touch events involves sending one of the **UIResponder** messages to the view that owns the touch. (If your touches are sluggish, then one of your methods is hogging the CPU, and events are waiting in line to be delivered. Chapter 22 will show you how to catch these problems.)

What about multiple touches? If multiple fingers do the same thing at the exact same time to the same view, all of these touch events are delivered at once. Each touch object – one for each finger – is included in the **NSSet** passed as an argument in the **UIResponder** messages. However, the window of opportunity for the “exact same time” is fairly short. So, instead of one responder message with all of the touches, there are usually multiple responder messages with one or more of the touches.

Creating the TouchTracker Application

Now let's get started with your application. In Xcode, create a new Single View iPhone project and name it TouchTracker (Figure 20.2).

Figure 20.2 Creating TouchTracker



First, you will need a model object that describes a line. Create a new subclass of **NSObject** and name it **Line**. In **Line.swift**, declare two **CGPoint** properties and an initializer:

```

import UIKit

class Line: NSObject {
    var begin: CGPoint = CGPointZero
    var end: CGPoint = CGPointZero

    init(begin: CGPoint, end: CGPoint) {
        self.begin = begin
        self.end = end

        super.init()
    }
}

```

Next, create a new `NSObject` subclass called `DrawView`. In `DrawView.swift`, change the superclass to `UIView`.

```

import UIKit

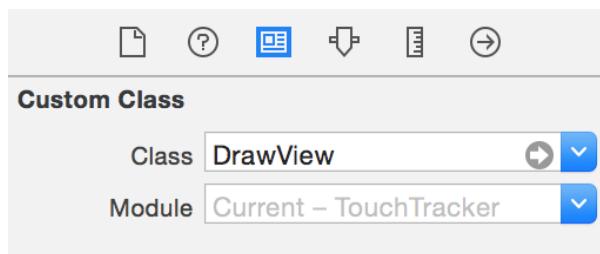
class DrawView: NSObject {
    class DrawView: UIView {
        }
}

```

An instance of `DrawView` will be the view of the application's `rootViewController`.

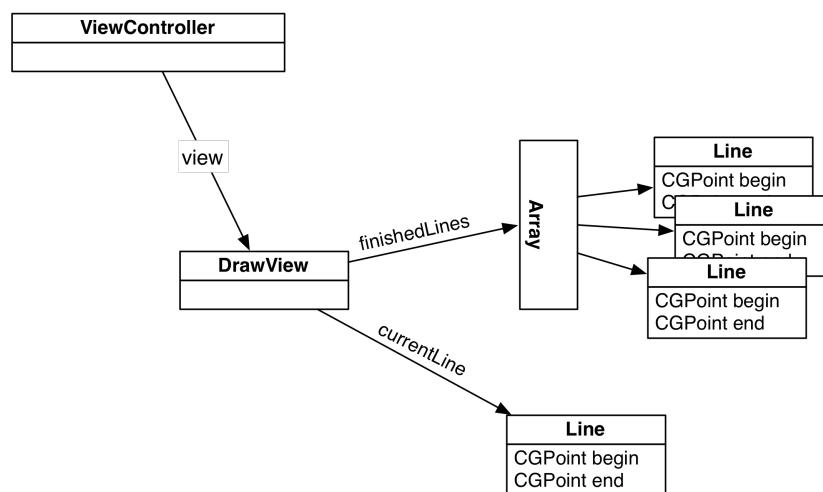
Open `Main.storyboard`. Select the View and open its Identity inspector (Command-Option-3). Under Custom Class, change the Class to `DrawView` (Figure 20.3).

Figure 20.3 Change View Class



The major objects you have just set up for `TouchTracker` are shown in Figure 20.4.

Figure 20.4 Object diagram for `TouchTracker`



Drawing with `DrawView`

`DrawView` will keep track of all of the lines that have been drawn and the line that is currently being drawn. In `DrawView.swift`, create two properties that will hold the lines in their two states.

```
import UIKit

class DrawView: UIView {

    var currentLine: Line?
    var finishedLines = [Line]()

}
```

We will get to how lines are created in a moment, but in order to test that the line creation code is written correctly, you need the `DrawView` to be able to draw lines.

In `DrawView.swift`, implement `drawRect(_:_)` to draw the current and finished lines as well as a helper method for stroking lines.

```
func strokeLine(line: Line) {
    let path = UIBezierPath()
    path.lineWidth = 10
    path.lineCapStyle = kCGLineCapRound

    path.moveToPoint(line.begin)
    path.addLineToPoint(line.end)
    path.stroke()
}

override func drawRect(rect: CGRect) {
    // Draw finished lines in black
    UIColor.blackColor().setStroke()
    for line in finishedLines {
        strokeLine(line)
    }

    if let line = currentLine {
        // If there is a line currently being drawn, do it in red
        UIColor.redColor().setStroke()
        strokeLine(line)
    }
}
```

Turning Touches into Lines

A line is defined by two points. Your `Line` stores these points as properties named `begin` and `end`. When a touch begins, you will create a line and set both `begin` and `end` to the point where the touch began. When the touch moves, you will update its `end`. When the touch ends, you will have your complete line.

In `DrawView.swift`, implement `touchesBegan(_:_withEvent:)` to create a new line.

```
override func touchesBegan(touches: NSSet, withEvent event: UIEvent) {
    let touch = touches.anyObject() as UITouch

    // Get location of the touch in view's coordinate system
    let location = touch.locationInView(self)

    currentLine = Line(begin: location, end: location)

    setNeedsDisplay()
}
```

Then, in `DrawView.swift`, implement `touchesMoved(_:_withEvent:)` so that it updates the `end` of the `currentLine`.

```
override func touchesMoved(touches: NSSet, withEvent event: UIEvent) {
    let touch = touches.anyObject() as UITouch
    let location = touch.locationInView(self)

    currentLine?.end = location

    setNeedsDisplay()
}
```

Finally, in `DrawView.swift`, add the `currentLine` to the `finishedLines` when the touch ends.

```
override func touchesEnded(touches: NSSet, withEvent event: UIEvent) {
    if let currentLine = currentLine {
        finishedLines.append(currentLine)
    }
    currentLine = nil

    setNeedsDisplay()
}
```

Build and run the application and draw some lines on the screen. While you are drawing, the lines will appear in red and once finished, they will appear in black.

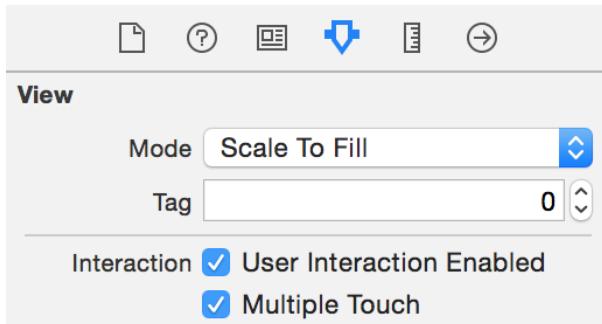
Handling multiple touches

When drawing lines, you may have noticed that having more than one finger on the screen does not do anything – that is, you can only draw one line at a time. Let's update `DrawView` so that you can draw as many lines as you can fit fingers on the screen.

By default, a view will only accept one touch at a time. If one finger has already triggered `touchesBegan(_:withEvent:)` but has not finished – and therefore has not triggered `touchesEnded(_:withEvent:)` – subsequent touches are ignored. In this context, “ignore” means that the `DrawView` will not be sent `touchesBegan(_:withEvent:)` or any other `UIResponder` messages related to the extra touches.

In `Main.storyboard`, select the Draw View and open its Attributes inspector. Check the box labeled Multiple Touch, which will set the `DrawView` instances's `multipleTouchEnabled` property to `true`.

Figure 20.5 Multiple Touch Enabled



Now that `DrawView` will accept multiple touches, each time a finger touches the screen, moves, or is removed from the screen, the view will receive the appropriate `UIResponder` message. However, this now presents a problem: your `UIResponder` code assumes there will only be one touch active and one line being drawn at a time.

Notice, first, that each touch handling method you have already implemented calls the method `anyObject()` on the `NSSet` of touches it receives. In a single-touch view, there will only ever be one object in the set, so asking for any object will always give you the touch that triggered the event. In a multiple touch view, that set could contain more than one touch.

Then, notice that there is only one property (`currentLine`) that hangs on to a line in progress. Obviously, you will need to hold as many lines as there are touches currently on the screen. While you could create a few more properties, like `currentLine1` and `currentLine2`, you would have to go to considerable lengths to manage which instance variable corresponds to which touch.

Instead of the multiple property approach, you can use a `Dictionary` to hang on to each `Line` in progress. The key to store the line in the dictionary will be derived from the `UITouch` object that the line corresponds to. As more touch events occur, you can use the same algorithm to derive the key from the `UITouch` that triggered the event and use it to look up the appropriate `Line` in the dictionary.

In `DrawView.swift`, add a new instance variable to replace the `currentLine`.

```
class DrawView: UIView {
    var currentLine: Line?
    var currentLines = [NSValue:Line]()
}
```

Now you need to update the `UIResponder` methods to add lines that are currently being drawn to this dictionary. In `DrawView.swift`, update the code in `touchesBegan(_:withEvent:)`.

```
override func touchesBegan(touches: NSSet, withEvent event: UIEvent) {
    // Let's put in a log statement to see the order of events
    println(__FUNCTION__)

    for touch in touches.allObjects as [UITouch] {
        let location = touch.locationInView(self)

        let.newLine = Line(begin: location, end: location)

        let key = NSValue(nonretainedObject: touch)
        currentLines[key] = newLine
    }

    let touch = touches.anyObject() as UITouch
    let location = touch.locationInView(self)
    currentLine = Line(begin: location, end: location)

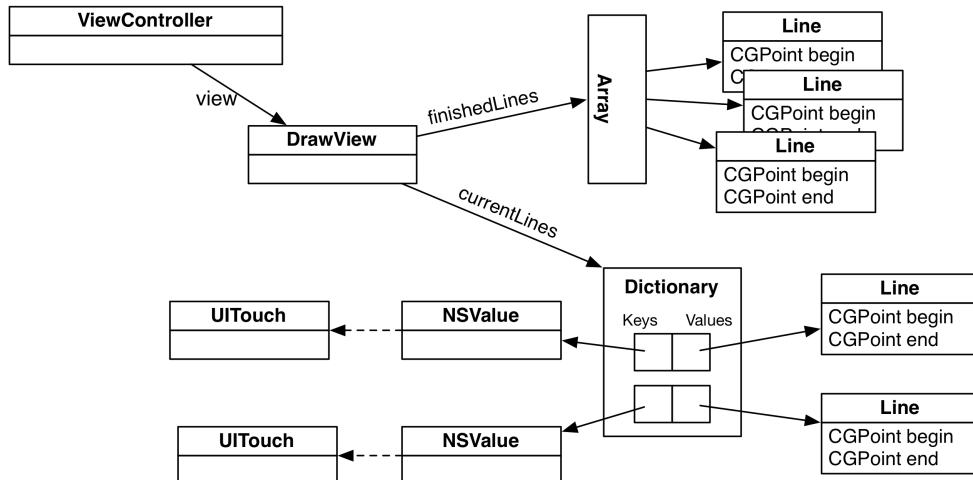
    setNeedsDisplay()
}
```

First, you print out the name of the method, which is represented by the `__FUNCTION__` **String** literal.

Second, notice that you use fast enumeration to loop over all of the touches that began, because it is possible that more than one touch can begin at the same time. (Although typically touches begin at different times and `DrawView` will receive multiple `touchesBegan(_:withEvent:)` messages containing each touch.)

Next, notice the use of `valueWithNonretainedObject(_:)` to derive the key to store the `Line`. This method creates an `NSValue` instance that holds on to the address of the `UITouch` object that will be associated with this line. Since a `UITouch` is created when a touch begins, updated throughout its lifetime, and destroyed when the touch ends, the address of that object will be constant through each touch event message.

Figure 20.6 Object diagram for Multitouch TouchTracker



Update `touchesMoved(_:withEvent:)` in `DrawView.swift` so that it can look up the right `Line`.

```

override func touchesMoved(touches: NSSet, withEvent event: UIEvent) {
    // Let's put in a print statement to see the order of events
    println(__FUNCTION__)

    for touch in touches.allObjects as [UITouch] {
        let key = NSValue(nonretainedObject: touch)
        currentLines[key]?.end = touch.locationInView(self)
    }

    let touch = touches.anyObject() as UITouch
    let location = touch.locationInView(self)

    currentLine?.end = location

    setNeedsDisplay()
}

```

Then, update `touchesEnded(_:withEvent:)` to move any finished lines into the `finishedLines` array.

```

override func touchesEnded(touches: NSSet, withEvent event: UIEvent) {
    // Let's put in a log statement to see the order of events
    println(__FUNCTION__)

    for touch in touches.allObjects as [UITouch] {
        let key = NSValue(nonretainedObject: touch)
        if var line = currentLines[key] {
            line.end = touch.locationInView(self)

            finishedLines.append(line)
            currentLines.removeValueForKey(key)
        }
    }

    if let currentLine = currentLine {
        finishedLines.append(currentLine)
    }
    currentLine = nil

    setNeedsDisplay()
}

```

Finally, update `drawRect(_:)` to draw each line in `currentLines`.

```

override func drawRect(rect: CGRect) {
    // Draw finished lines in black
    for line in finishedLines {
        strokeLine(line)
    }

    // Draw current lines in red
    UIColor.redColor().setStroke()
    for (key,line) in currentLines {
        strokeLine(line)
    }

    if let line = currentLine {
        UIColor.redColor().setStroke()
        strokeLine(line)
    }
}

```

Build and run the application and start drawing lines with multiple fingers. (You can simulate multiple fingers on the simulator by holding down the Option key as you drag.)

Also, you should know that when a `UIResponder` message like `touchesMoved(_:withEvent:)` is sent to a view, only the touches that have moved will be in the `NSSet` of touches. Thus, it is possible for three touches to be on a view, but only one touch inside the set of touches passed into one of these methods if the other two did not move. Additionally, once a `UITouch` begins on a view, all touch event messages are sent to that same view over the touch's lifetime, even if that touch moves off of the view it began on.

The last thing left for the basics of TouchTracker is to handle what happens when a touch is cancelled. A touch can be cancelled when an application is interrupted by the operating system (for example, a phone call comes in) when a touch is currently on the screen. When a touch is cancelled, any state it set up should be reverted. In this case, you should remove any lines in progress.

In DrawView.swift, implement `touchesCancelled(_:withEvent:)`.

```
override func touchesCancelled(touches: NSSet, withEvent event: UIEvent) {
    // Let's put in a log statement to see the order of events
    println(__FUNCTION__)

    for touch in touches.allObjects as [UITouch] {
        let key = NSValue(nonretainedObject: touch)
        currentLines.removeValueForKey(key)
    }

    setNeedsDisplay()
}
```

Bronze Challenge: Saving and Loading

Save the lines when the application terminates. Reload them when the application resumes.

Silver Challenge: Colors

Make it so the angle at which a line is drawn dictates its color once it has been added to `currentLines`.

Gold Challenge: Circles

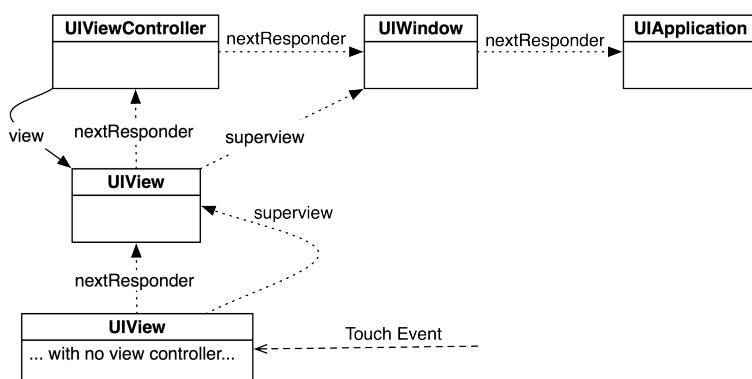
Use two fingers to draw circles. Try having each finger represent one corner of the bounding box around the circle. Recall that you can simulate two fingers on the simulator by holding down the Option key. (Hint: This is much easier if you track touches that are working on a circle in a separate dictionary.)

For the More Curious: The Responder Chain

In ???, we talked briefly about **UIResponder** and the first responder. A **UIResponder** can receive touch events. **UIView** is one example of a **UIResponder** subclass, but there are many others, including **UIViewController**, **UIApplication**, and **UIWindow**. You are probably thinking, “But you can’t touch a **UIViewController**. It’s not an on-screen object.” You are right – you cannot send a touch event *directly* to a **UIViewController**, but view controllers can receive events through the *responder chain*.

Every **UIResponder** has a method called `nextResponder()`, and together these objects make up the responder chain (Figure 20.7). A touch event starts at the view that was touched. The `nextResponder` of a view is typically its **UIViewController** (if it has one) or its superview (if it does not). The `nextResponder()` of a view controller is typically its view’s superview. The top-most superview is the window. The window’s `nextResponder()` is the singleton instance of **UIApplication**.

Figure 20.7 Responder chain



How does a **UIResponder** *not* handle an event? It forwards the same message to its **nextResponser()**. That is what the default implementation of methods like **touchesBegan(_:withEvent:)** do. So if a method is not overridden, its next responder will attempt to handle the touch event. If the application (the last object in the responder chain) does not handle the event, then it is discarded.

You can explicitly send a message to a next responder, too. Let's say there is a view that tracks touches, but if a double tap occurs, its next responder should handle it. The code would look like this:

```
override func touchesBegan(touches: NSSet, withEvent event: UIEvent) {
    let touch = touches.anyObject() as UITouch
    if touch.tapCount == 2 {
        nextResponser()?.touchesBegan(touches, withEvent: event)
    }
    else {
        // Go on to handle touches that are not double taps
    }
}
```

For the More Curious: UIControl

The class **UIControl** is the superclass for several classes in Cocoa Touch, including **UIButton** and **UISlider**. You have seen how to set the targets and actions for these controls. Now we can take a closer look at how **UIControl** overrides the same **UIResponder** methods you implemented in this chapter.

In **UIControl**, each possible *control event* is associated with a constant. Buttons, for example, typically send action messages on the **UIControlEvents.TouchUpInside** control event. A target registered for this control event will only receive its action message if the user touches the control and then lifts the finger off the screen inside the frame of the control. Essentially, it is a tap.

For a button, however, you can have actions on other event types. For example, you might trigger a method if the user removes the finger *inside or outside* the frame. Assigning the target and action programmatically would look like this:

```
button.addTarget(self,
    action: "resetTemperature:",
    forControlEvents: .TouchUpInside | .TouchUpOutside)
```

Now consider how **UIControl** handles **UIControlEvents.TouchUpInside**.

```
// Not the exact code. There is a bit more going on!
override func touchesEnded(touches: NSSet, withEvent event: UIEvent) {

    // Reference to the touch that is ending
    let touch = touches.anyObject() as UITouch

    // Location of that point in this control's coordinate system
    let touchLocation = touch.locationInView(self)

    // Is that point still in my viewing bounds?
    if CGRectContainsPoint(self.bounds, touchLocation) {
        // Send out action messages to all targets registered for this event!
        sendActionsForControlEvents(.TouchUpInside)
    }
    else {
        // The touch ended outside the bounds, different control event
        sendActionsForControlEvents(.TouchUpOutside)
    }
}
```

So how do these actions get sent to the right target? At the end of the **UIResponder** method implementations, the control sends the message **sendActionsForControlEvents(_:)** to itself. This method looks at all of the target-action pairs the control has, and if any of them are registered for the control event passed as the argument, those targets are sent an action message.

However, a control never sends a message directly to its targets. Instead, it routes these messages through the **UIApplication** object. Why not have controls send the action messages directly to the targets? Controls can

also have `nil`-targeted actions. If a **UIControl**'s target is `nil`, the **UIApplication** finds the *first responder* of its **UIWindow** and sends the action message to it.

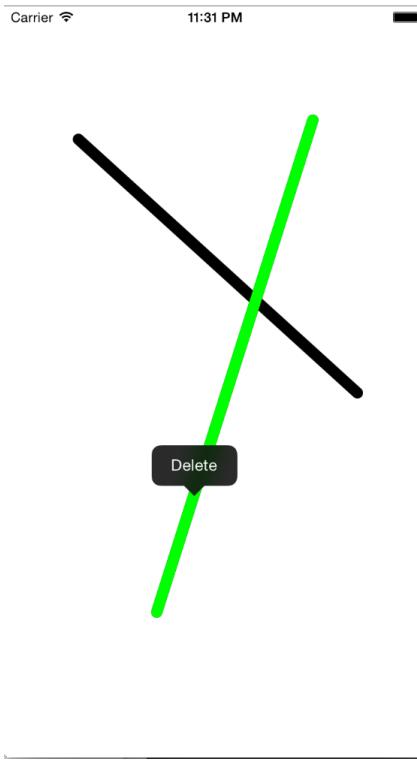
21

UIGestureRecognizer and UIMenuController

In Chapter 20, you handled raw touches and determined their course by implementing methods from **UIResponder**. Sometimes you want to detect a specific pattern of touches that make a gesture, like a pinch or a swipe. Instead of writing code to detect common gestures yourself, you can use instances of **UIGestureRecognizer**.

A **UIGestureRecognizer** intercepts touches that are on their way to being handled by a view. When it recognizes a particular gesture, it sends a message to the object of your choice. There are several types of gesture recognizers built into the SDK. In this chapter, you will use three of them to allow TouchTracker users to select, move, and delete lines (Figure 21.1). You will also see how to use another interesting iOS class, **UIMenuController**.

Figure 21.1 TouchTracker by the end of the chapter



UIGestureRecognizer Subclasses

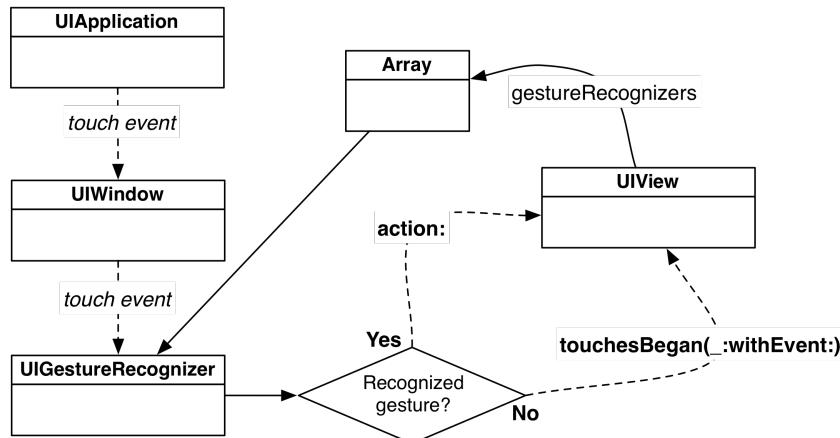
You do not instantiate **UIGestureRecognizer** itself. Instead, there are a number of subclasses of **UIGestureRecognizer**, and each one is responsible for recognizing a particular gesture.

To use an instance of a **UIGestureRecognizer** subclass, you give it a target-action pair and attach it to a view. Whenever the gesture recognizer recognizes its gesture on the view, it will send the action message to its target. All **UIGestureRecognizer** action messages have the same form:

```
func action(gestureRecognizer: UIGestureRecognizer) { }
```

When recognizing a gesture, the gesture recognizer intercepts the touches destined for the view (Figure 21.2). Thus, a view with gesture recognizers may not receive the typical **UIResponder** messages like `touchesBegan(_:withEvent:)`.

Figure 21.2 Gesture recognizers intercept touches



Detecting Taps with UITapGestureRecognizer

The first **UIGestureRecognizer** subclass you will use is **UITapGestureRecognizer**. When the user taps the screen twice, all of the lines on the screen will be cleared. Open TouchTracker.xcodeproj from Chapter 20.

In `DrawView.swift`, instantiate a **UITapGestureRecognizer** that requires two taps to fire in `init(coder:)`.

```
required init(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)

    let doubleTapRecognizer = UITapGestureRecognizer(target: self, action: "doubleTap:")
    doubleTapRecognizer.numberOfTapsRequired = 2
    addGestureRecognizer(doubleTapRecognizer)
}
```

When a double tap occurs on an instance of `DrawView`, the method `doubleTap(_)` will be called on that instance. Implement this method in `DrawView.swift`.

```
func doubleTap(gestureRecognizer: UIGestureRecognizer) {
    println("Recognized a double tap")

    currentLines.removeAll(keepCapacity: false)
    finishedLines.removeAll(keepCapacity: false)
    setNeedsDisplay()
}
```

Notice that the argument to the action method for a gesture recognizer is the instance of **UIGestureRecognizer** that called the method. In the case of a double tap, you do not need any information from the recognizer, but you will need information from the other recognizers you install later in the chapter.

Build and run the application, draw a few lines, and double-tap the screen to clear them.

You may have noticed (especially on the simulator) that during a double tap the first tap draws a small red dot. This dot appears because `touchesBegan(_:withEvent:)` is called on the `DrawView` on the first tap, creating a small line. Check the console and you will see the following sequence of events:

```
touchesBegan(_:withEvent:)
Recognized a double tap
touchesCancelled(_:withEvent:)
```

Gesture recognizers work by inspecting touch events to determine if their particular gesture occurred. Before a gesture is recognized, all **UIResponder** messages will be delivered to a view as normal. Since a tap gesture recognizer is recognized when a touch begins and ends within a small area in a small amount of time, the **UITapGestureRecognizer** cannot claim the touch is a tap just yet and `touchesBegan(_:withEvent:)` is sent

to the view. When the tap is finally recognized, the gesture recognizer claims the touch involved in the tap for itself and no more **UIResponder** messages will be sent to the view for that particular touch. In order to communicate this touch take-over to the view, **touchesCancelled(_:withEvent:)** is sent to the view and the **NSSet** of touches contains that **UITouch** instance.

To prevent this red dot from appearing temporarily, you can tell a **UIGestureRecognizer** to delay the sending of **touchesBegan(_:withEvent:)** to its view if it is still possible for the gesture to be recognized.

In **DrawView.swift**, modify **init(coder:)** to do just this.

```
required init(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)

    let doubleTapRecognizer = UITapGestureRecognizer(target: self, action: "doubleTap:")
    doubleTapRecognizer.numberOfTapsRequired = 2
    doubleTapRecognizer.delaysTouchesBegan = true
    addGestureRecognizer(doubleTapRecognizer)
}
```

Build and run the application, draw some lines, and then double-tap to clear them. You will no longer see the red dot while double tapping.

Multiple Gesture Recognizers

Let's add another gesture recognizer that allows the user to select a line. (Later, a user will be able to delete the selected line.) You will install another **UITapGestureRecognizer** on the **DrawView** that only requires one tap.

In **DrawView.swift**, modify **init(coder:)** to add this additional gesture recognizer.

```
required init(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)

    let doubleTapRecognizer = UITapGestureRecognizer(target: self, action: "doubleTap:")
    doubleTapRecognizer.numberOfTapsRequired = 2
    doubleTapRecognizer.delaysTouchesBegan = true
    addGestureRecognizer(doubleTapRecognizer)

    let tapRecognizer = UITapGestureRecognizer(target: self, action: "tap:")
    tapRecognizer.delaysTouchesBegan = true
    addGestureRecognizer(tapRecognizer)
}
```

Now, implement **tap(_:)** to log the tap to the console in **DrawView.swift**.

```
func tap(gestureRecognizer: UIGestureRecognizer) {
    println("Recognized a tap")
}
```

Build and run the application. Tapping once will log the appropriate message to the console. The only problem, however, is that tapping twice will trigger both **tap(_:)** and **doubleTap(_:)**.

In situations where you have multiple gesture recognizers, it is not uncommon to have a gesture recognizer fire when you really want another gesture recognizer to handle the work. In these cases, you set up dependencies between recognizers that say, “Just wait a moment before you fire, because this gesture might be mine!”

In **init(coder:)**, make it so the **tapRecognizer** must wait for the **doubleTapRecognizer** to fail before it can assume that a single tap is not just the first of a double tap.

```
required init(coder aDecoder: NSCoder) {
    super.init(coder: aDecoder)

    let doubleTapRecognizer = UITapGestureRecognizer(target: self, action: "doubleTap:")
    doubleTapRecognizer.numberOfTapsRequired = 2
    doubleTapRecognizer.delaysTouchesBegan = true
    addGestureRecognizer(doubleTapRecognizer)

    let tapRecognizer = UITapGestureRecognizer(target: self, action: "tap:")
    tapRecognizer.delaysTouchesBegan = true
    tapRecognizer.requireGestureRecognizerToFail(doubleTapRecognizer)
    addGestureRecognizer(tapRecognizer)
}
```

Build and run the application. A single tap now takes a small amount of time to fire after the tap occurs, but double-tapping no longer triggers the `tap(_:)` message.

Now, let's build on the `DrawView` so that the user can select lines when they are tapped. First, add a property to hold onto a selected line at the top of `DrawView.swift`.

```
class DrawView: UIView {  
  
    var currentLines = [NSValue:Line]()  
    var finishedLines = [Line]()  
    weak var selectedLine: Line?  
}
```

(Notice that this property is weak: the `finishedLines` array will hold the strong reference to the line and `selectedLine` will be set to nil if the line is removed from `finishedLines` by clearing the screen.)

Now, in `drawRect(_:)`, add some code to the bottom of the method to draw the selected line in green.

```
override func drawRect(rect: CGRect) {  
    for line in finishedLines {  
        strokeLine(line)  
    }  
  
    UIColor.redColor().setStroke()  
    for (key,line) in currentLines {  
        strokeLine(line)  
    }  
  
    if let line = selectedLine {  
        UIColor.greenColor().setStroke()  
        strokeLine(line)  
    }  
}
```

Implement `lineAtPoint(_:)` in `DrawView.swift` to get a `Line` close to the given point.

```
func lineAtPoint(point: CGPoint) -> Line? {  
  
    // Find a line close to point  
    for line in finishedLines {  
        let begin = line.begin  
        let end = line.end  
  
        // Check a few points on the line  
        for var t: CGFloat = 0; t < 1.0; t += 0.05 {  
            let x = begin.x + ((end.x - begin.x) * t)  
            let y = begin.y + ((end.y - begin.y) * t)  
  
            // If the tapped point is within 20 points, let's return this line  
            if hypot(x - point.x, y - point.y) < 20.0 {  
                return line  
            }  
        }  
    }  
  
    // If nothing is close enough to the tapped point, then we did not select a line  
    return nil  
}
```

(There are better ways to implement `lineAtPoint(_:)`, but this simplistic implementation is OK for your current purpose.)

The point you are interested in, of course, is where the tap occurred. You can easily get this information. Every `UIGestureRecognizer` has a `locationInView(_:)` method. Calling this method on the gesture recognizer will give you the coordinate where the gesture occurred in the coordinate system of the view that is passed as the argument.

In `DrawView.swift`, call the `locationInView(_:)` method on the gesture recognizer, pass the result to `lineAtPoint(_:)`, and make the returned line the `selectedLine`.

```

func tap(gestureRecognizer: UIGestureRecognizer) {
    println("Recognized a tap")

    let point = gestureRecognizer.locationInView(self)
    selectedLine = lineAtPoint(point)

    setNeedsDisplay()
}

```

Build and run the application. Draw a few lines and then tap on one. The tapped line should appear in green, but remember that it takes a short moment before the tap is known not to be a double tap.

UIMenuController

Next you are going to make it so that when the user selects a line, a menu appears right where the user tapped that offers the option to delete that line. There is a built-in class for providing this sort of menu called **UIMenuController** (Figure 21.3). A menu controller has a list of **UIMenuItem** objects and is presented in an existing view. Each item has a title (what shows up in the menu) and an action (the message it sends the first responder of the window).

Figure 21.3 A **UIMenuController**



There is only one **UIMenuController** per application. When you wish to present this instance, you fill it with menu items, give it a rectangle to present from, and set it to be visible.

Do this in `DrawView.swift`'s `tap(_:`) method if the user has tapped on a line. If the user tapped somewhere that is not near a line, the currently selected line will be deselected, and the menu controller will hide.

```

func tap(gestureRecognizer: UIGestureRecognizer) {
    println("Recognized a tap")

    let point = gestureRecognizer.locationInView(self)
    selectedLine = lineAtPoint(point)

    // Grab the menu controller
    let menu = UIMenuController.sharedMenuController()

    if let line = selectedLine {

        // Make ourselves the target of menu item action messages
        becomeFirstResponder()

        // Create a new "Delete" UIMenuItem
        let deleteItem = UIMenuItem(title: "Delete", action: "deleteLine:")
        menu.menuItems = [deleteItem]

        // Tell the menu where it should come from and show it
        menu.setTargetRect(CGRect(x: point.x, y: point.y, width: 2, height: 2), inView: self)
        menu.setMenuVisible(true, animated: true)
    }
    else {
        // Hide the menu if no line is selected
        menu.setMenuVisible(false, animated: true)
    }

    setNeedsDisplay()
}

```

For a menu controller to appear, a view that responds to at least one action message in the **UIMenuController**'s menu items must be the first responder of the window – this is why you called the method `becomeFirstResponder()` to the `DrawView` before setting up the menu controller.

If you have a custom view class that needs to become the first responder, you must override `canBecomeFirstResponder()`. In `DrawView.swift`, override this method to return `true`.

```
override func canBecomeFirstResponder() -> Bool {
    return true
}
```

You can build and run the application now, but when you select a line, the menu will not appear. When being presented, the menu controller goes through each menu item and asks the first responder if it implements the action message for that item. If the first responder does not implement that method, then the menu controller will not show the associated menu item. If no menu items have their action messages implemented by the first responder, the menu is not shown at all.

To get the Delete menu item (and the menu itself) to appear, implement `deleteLine(_:)` in `DrawView.swift`.

```
func deleteLine(sender: AnyObject) {
    // Remove the selected line from the list of finishedLines
    if let line = selectedLine {
        if let index = find(finishedLines, line) {
            finishedLines.removeAtIndex(index)

            // Redraw everything
            setNeedsDisplay()
        }
    }
}
```

Build and run the application. Draw a line, tap on it, and then select Delete from the menu item.

UILongPressGestureRecognizer

Let's test out two other subclasses of `UIGestureRecognizer`: `UILongPressGestureRecognizer` and `UIPanGestureRecognizer`. When you hold down on a line (a long press), that line will be selected and you can then drag it around by dragging your finger (a pan).

In this section, let's focus on the long press recognizer. In `DrawView.swift`, instantiate a `UILongPressGestureRecognizer` in `init(coder:)` and add it to the `DrawView`.

```
addGestureRecognizer(tapRecognizer)

let longPressRecognizer = UILongPressGestureRecognizer(target: self, action: "longPress:")
addGestureRecognizer(longPressRecognizer)
}
```

Now when the user holds down on the `DrawView`, the method `longPress(_:)` will be called on it. By default, a touch must be held 0.5 seconds to become a long press, but you can change the `minimumPressDuration` of the gesture recognizer if you like.

So far, you have worked with tap gestures. A tap is a simple gesture. By the time it is recognized, the gesture is over, and the action message has been delivered. A long press, on the other hand, is a gesture that occurs over time and is defined by three separate events.

For example, when the user touches a view, the long press recognizer notices a *possible* long press but must wait to see whether the touch is held long enough to become a long press gesture.

Once the user holds the touch long enough, the long press is recognized and the gesture has *begun*. When the user removes the finger, the gesture has *ended*.

Each of these events causes a change in the gesture recognizer's `state` property. For instance, the `state` of the long press recognizer described above would be `UIGestureRecognizerState.Possible`, then `UIGestureRecognizerState.Began`, and finally `UIGestureRecognizerState.Ended`.

When a gesture recognizer transitions to any state other than the possible state, it sends its action message to its target. This means the long press recognizer's target receives the same message when a long press begins and when it ends. The gesture recognizer's state allows the target to determine why it has been sent the action message and take the appropriate action.

Here is the plan for implementing your action method `longPress(_:)`. When the view receives `longPress(_:)` and the long press has begun, you will select the closest line to where the gesture occurred. This allows the user to select a line while keeping the finger on the screen (which is important in the next section when you implement panning). When the view receives `longPress(_:)` and the long press has ended, you will deselect the line.

In `DrawView.swift`, implement `longPress(_:)`.

```
func longPress(gestureRecognizer: UIGestureRecognizer) {
    if gestureRecognizer.state == .Began {
        let point = gestureRecognizer.locationInView(self)
        selectedLine = lineAtPoint(point)

        if selectedLine != nil {
            currentLines.removeAll(keepCapacity: false)
        }
    } else if gestureRecognizer.state == .Ended {
        selectedLine = nil
    }

    setNeedsDisplay()
}
```

Build and run the application. Draw a line and then hold down on it; the line will turn green and be selected and will stay that way until you let go.

UIPanGestureRecognizer and Simultaneous Recognizers

Once a line is selected during a long press, you want the user to be able to move that line around the screen by dragging it with a finger. So you need a gesture recognizer for a finger moving around the screen. This gesture is called *panning*, and its gesture recognizer subclass is `UIPanGestureRecognizer`.

Normally, a gesture recognizer does not share the touches it intercepts. Once it has recognized its gesture, it “eats” that touch, and no other recognizer gets a chance to handle it. In your case, this is bad: the entire pan gesture you want to recognize happens within a long press gesture. You need the long press recognizer and the pan recognizer to be able to recognize their gestures simultaneously. Let’s see how to do that.

First, in `DrawView.swift`, declare that `DrawView` conforms to the `UIGestureRecognizerDelegate` protocol. Then, declare a `UIPanGestureRecognizer` as a property so that you have access to it in all of your methods.

```
class DrawView: UIView, UIGestureRecognizerDelegate {

    var currentLines = [NSValue:Line]()
    var finishedLines = [Line]()
    weak var selectedLine: Line?
    var moveRecognizer: UIPanGestureRecognizer!
```

In `DrawView.swift`, add code to `init(coder:)` to instantiate a `UIPanGestureRecognizer`, set two of its properties, and attach it to the `DrawView`.

```
let longPressRecognizer = UILongPressGestureRecognizer(target: self, action: "longPress:")
addGestureRecognizer(longPressRecognizer)

moveRecognizer = UIPanGestureRecognizer(target: self, action: "moveLine:")
moveRecognizer.delegate = self
moveRecognizer.cancelsTouchesInView = false
addGestureRecognizer(moveRecognizer)
}
```

There are a number of methods in the `UIGestureRecognizerDelegate` protocol, but you are only interested in one – `gestureRecognizer(_:shouldRecognizeSimultaneouslyWithGestureRecognizer:)`. A gesture recognizer will call this method on its delegate when it recognizes its gesture but realizes that another gesture recognizer has recognized its gesture, too. If this method returns `true`, the recognizer will share its touches with other gesture recognizers.

In DrawView.swift, return true when the moveRecognizer calls the method on its delegate.

```
func gestureRecognizer(gestureRecognizer: UIGestureRecognizer,
    shouldRecognizeSimultaneouslyWithGestureRecognizer
    otherGestureRecognizer: UIGestureRecognizer) -> Bool {
    if moveRecognizer == gestureRecognizer {
        return true
    }
    return false
}
```

Now when the user begins a long press, the **UIPanGestureRecognizer** will be allowed to keep track of this finger, too. When the finger begins to move, the pan recognizer will transition to the began state. If these two recognizers could not work simultaneously, the long press recognizer would start, and the pan recognizer would never transition to the began state or send its action message to its target.

In addition to the states you have already seen, a pan gesture recognizer supports the *changed* state. When a finger starts to move, the pan recognizer enters the began state and sends a message to its target. While the finger moves around the screen, the recognizer transitions to the changed state and sends its action message to its target repeatedly. Finally, when the finger leaves the screen, the recognizer's state is set to ended, and the final message is delivered to the target.

Now you need to implement the **moveLine(_:)** method that the pan recognizer sends its target. In this implementation, you will send the message **translationInView(_:)** to the pan recognizer. This **UIPanGestureRecognizer** method returns how far the pan has moved as a **CGPoint** in the coordinate system of the view passed as the argument. When the pan gesture begins, this property is set to the zero point (where both x and y equal zero). As the pan moves, this value is updated – if the pan goes very far to the right, it has a high x value; if the pan returns to where it began, its translation goes back to the zero point.

In DrawView.swift, implement **moveLine(_:)**. Notice that because you will send the gesture recognizer a method from the **UIPanGestureRecognizer** class, the parameter of this method must be a reference to an instance of **UIPanGestureRecognizer** rather than **UIGestureRecognizer**.

```
func moveLine(gestureRecognizer: UIPanGestureRecognizer) {
    // If we have a selected line ...
    if var line = selectedLine {
        // When the pan recognizer changes its position...
        if gestureRecognizer.state == .Changed {
            // How far has the pan moved?
            let translation = gestureRecognizer.translationInView(self)

            // Add the translation to the current beginning and end points of the line
            // Make sure there are no copy and paste typos!
            line.begin.x += translation.x
            line.begin.y += translation.y
            line.end.x += translation.x
            line.end.y += translation.y

            // Redraw the screen
            setNeedsDisplay()
        }
    }
    else {
        // If we have not selected a line, we do not do anything here
        return
    }
}
```

Build and run the application. Touch and hold on a line and begin dragging – and you will immediately notice that the line and your finger are way out of sync. This makes sense because you are adding the current translation over and over again to the line's original end points. You really need the gesture recognizer to report the change in translation since the last time this method was called instead. Fortunately, you can do this. You can set the translation of a pan gesture recognizer back to the zero point every time it reports a change. Then, the next time it reports a change, it will have the translation since the last event.

Near the bottom of **moveLine(_:)** in DrawView.swift, add the following line of code.

```

line.end.x += translation.x
line.end.y += translation.y

gestureRecognizer.setTranslation(CGPointZero, inView: self)

// Redraw the screen
setNeedsDisplay()

```

Build and run the application and move a line around. Works great!

Before moving on, let's take a look at a property you set in the pan gesture recognizer – `cancelsTouchesInView`. Every `UIGestureRecognizer` has this property and, by default, this property is true. This means that the gesture recognizer will eat any touch it recognizes so that the view will not have a chance to handle it via the traditional `UIResponder` methods, like `touchesBegan(_:withEvent:)`.

Usually, this is what you want, but not always. In this case, the gesture that the pan recognizer recognizes is the same kind of touch that the view handles to draw lines using the `UIResponder` methods. If the gesture recognizer eats these touches, then users will not be able to draw lines.

When you set `cancelsTouchesInView` to false, touches that the gesture recognizer recognizes also get delivered to the view via the `UIResponder` methods. This allows both the recognizer and the view's `UIResponder` methods to handle the same touches. If you are curious, comment out the line that sets `cancelsTouchesInView` to false and build and run again to see the effects.

For the More Curious: UIMenuController and UIResponderStandardEditActions

The `UIMenuController` is typically responsible for showing the user an “edit” menu when it is displayed; think of a text field or text view when you press and hold. Therefore, an unmodified menu controller (one that you do not set the menu items for) already has default menu items that it presents, like Cut, Copy, and other familiar options. Each item has an action message wired up. For example, `cut:` is sent to the view presenting the menu controller when the Cut menu item is tapped.

All instances of `UIResponder` implement these methods, but, by default, these methods do not do anything. Subclasses like `UITextField` override these methods to do something appropriate for their context, like cut the currently selected text. The methods are all declared in the `UIResponderStandardEditActions` protocol.

If you override a method from `UIResponderStandardEditActions` in a view, its menu item will automatically appear in any menu you show for that view. This works because the menu controller sends the message `canPerformAction(_:withSender:)` to its view, which returns `true` or `false` depending on whether the view implements this method.

If you want to implement one of these methods but *do not* want it to appear in the menu, you can override `canPerformAction(_:withSender:)` to return `false`.

```

override func canPerformAction(action: Selector, withSender sender: AnyObject?) -> Bool {
    if action == "copy:" {
        return false
    }
    else {
        // Else return the default behavior
        return super.canPerformAction(action, withSender: sender)
    }
}

```

For the More Curious: More on UIGestureRecognizer

We have only scratched the surface of `UIGestureRecognizer`; there are more subclasses, more properties, and more delegate methods, and you can even create recognizers of your own. This section will give you an idea of what `UIGestureRecognizer` is capable of, and then you can study the documentation for `UIGestureRecognizer` to learn even more.

When a gesture recognizer is on a view, it is really handling all of the `UIResponder` methods, like `touchesBegan(_:withEvent:)`, for you. Gesture recognizers are pretty greedy, so they typically do not let

a view receive touch events or they at least delay the delivery of those events. You can set properties on the recognizer, like `delaysTouchesBegan`, `delaysTouchesEnded`, and `cancelsTouchesInView`, to change this behavior. If you need finer control than this all-or-nothing approach, you can implement delegate methods for the recognizer.

At times, you may have two gesture recognizers looking for very similar gestures. You can chain recognizers together so that one is required to fail for the next one to start using the method `requireGestureRecognizerToFail(_ :)`.

One thing you must understand to master gesture recognizers is how they interpret their state. Overall, there are seven states a recognizer can enter:

- `UIGestureRecognizerState.Possible`
- `UIGestureRecognizerState.Began`
- `UIGestureRecognizerState.Changed`
- `UIGestureRecognizerState.Ended`
- `UIGestureRecognizerState.Failed`
- `UIGestureRecognizerState.Cancelled`
- `UIGestureRecognizerState.Recognized`

Most of the time, a recognizer will stay in the possible state. When a recognizer recognizes its gesture, it goes into the began state. If the gesture is something that can continue, like a pan, it will go into and stay in the changed state until it ends. When any of its properties change, it sends another message to its target. When the gesture ends (typically when the user lifts the finger), it enters the ended state.

Not all recognizers begin, change, and end. For gesture recognizers that pick up on a discrete gesture like a tap, you will only ever see the recognized state (which has the same value as the ended state).

Finally, a recognizer can be cancelled (by an incoming phone call, for example) or fail (because no amount of finger contortion can make the particular gesture from where the fingers currently are). When these states are transitioned to, the action message of the recognizer is sent, and the state property can be checked to see why.

The four built-in recognizers you did not implement in this chapter are `UIPinchGestureRecognizer`, `UISwipeGestureRecognizer`, `UIScreenEdgePanGestureRecognizer`, and `UIRotationGestureRecognizer`. Each of these have properties that allow you to fine-tune their behavior. The documentation will show you the way.

Finally, if there is a gesture you want to recognize that is not implemented by the built-in subclasses of `UIGestureRecognizer`, you can subclass `UIGestureRecognizer` yourself. This is an intense undertaking and outside the scope of this book. You can read the Subclassing Notes in the `UIGestureRecognizer` documentation to learn what is required.

Silver Challenge: Mysterious Lines

There is a bug in the application. If you tap on a line and then start drawing a new one while the menu is visible, you will drag the selected line *and* draw a new line at the same time. Fix this bug.

Gold Challenge: Speed and Size

Piggy-back off of the pan gesture recognizer to record the velocity of the pan when you are drawing a line. Adjust the thickness of the line being drawn based on this speed. Make no assumptions about how small or large the velocity value of the pan recognizer can be. (In other words, log a variety of velocities to the console first.)

Mega-Gold Challenge: Colors

Have a three-finger swipe upwards bring up a panel that shows some colors. Selecting one of those colors should make any lines you draw afterwards appear in that color. No extra lines should be drawn by putting up that panel – or at least any lines drawn should be immediately deleted when the application realizes that it is dealing with a three-finger swipe.

22

Debugging Tools

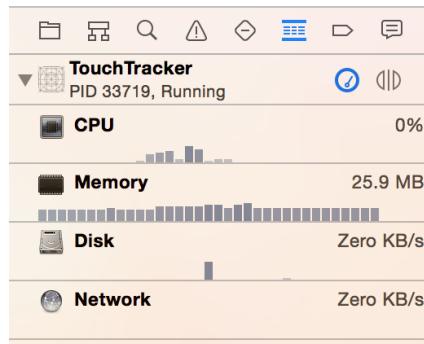
In ???, you learned about using the debugger to find and fix problems in your code. Now we are going to look at other tools available to iOS programmers and how you can integrate them into your application development.

Gauges

Xcode 5 introduced *debug gauges* that provide at-a-glance information about your application's CPU and memory usage.

Open your TouchTracker project and run it, preferably on a provisioned iOS device rather than the iOS Simulator. In the navigator, select the  tab to open the debug navigator.

Figure 22.1 Gauges

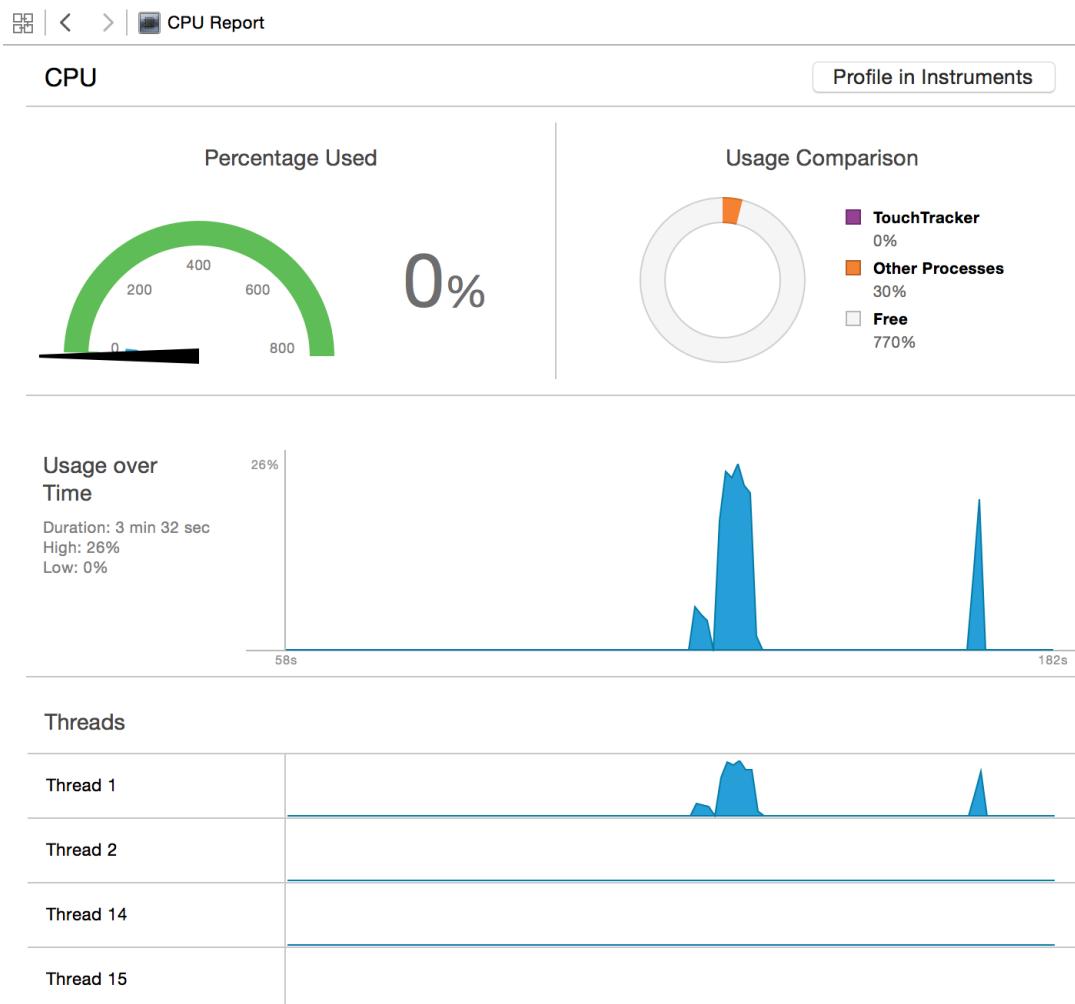


While the application is running (not paused or crashed), the debug navigator shows CPU and memory gauges, as well as a couple others (Figure 22.1). Each of these shows a live-updating graph of resource usage over time, as well as a numerical figure describing that resource's current usage.

Important note: these gauges scale based on the hardware that is actually running your application. Your Mac has much more available RAM and likely more CPU cores than iOS devices do, so if you run your application in the iOS Simulator, your CPU and memory usage will appear to be very low.

Click on the CPU Debug Gauge. This will present the CPU Report in the Editor pane (Figure 22.2).

Figure 22.2 CPU report



Percentage Used

shows your CPU utilization relative to the number of CPU cores your device has. For example, dual-core devices will show CPU usage out of 200%. While your application is idle, this should read 0%.

Usage Comparison

allows you to see your application's CPU usage as it impacts the rest of the system. At any given time, your application is not the only cause of activity on the device. Some applications may be running in the background, putting their own pressure on the system. If your app feels slow but is not using much CPU on its own, this may be why.

Usage over Time

graphs your application's CPU usage and shows how long the application has been running, as well as peak and trough usage values over the course of the current run.

Threads

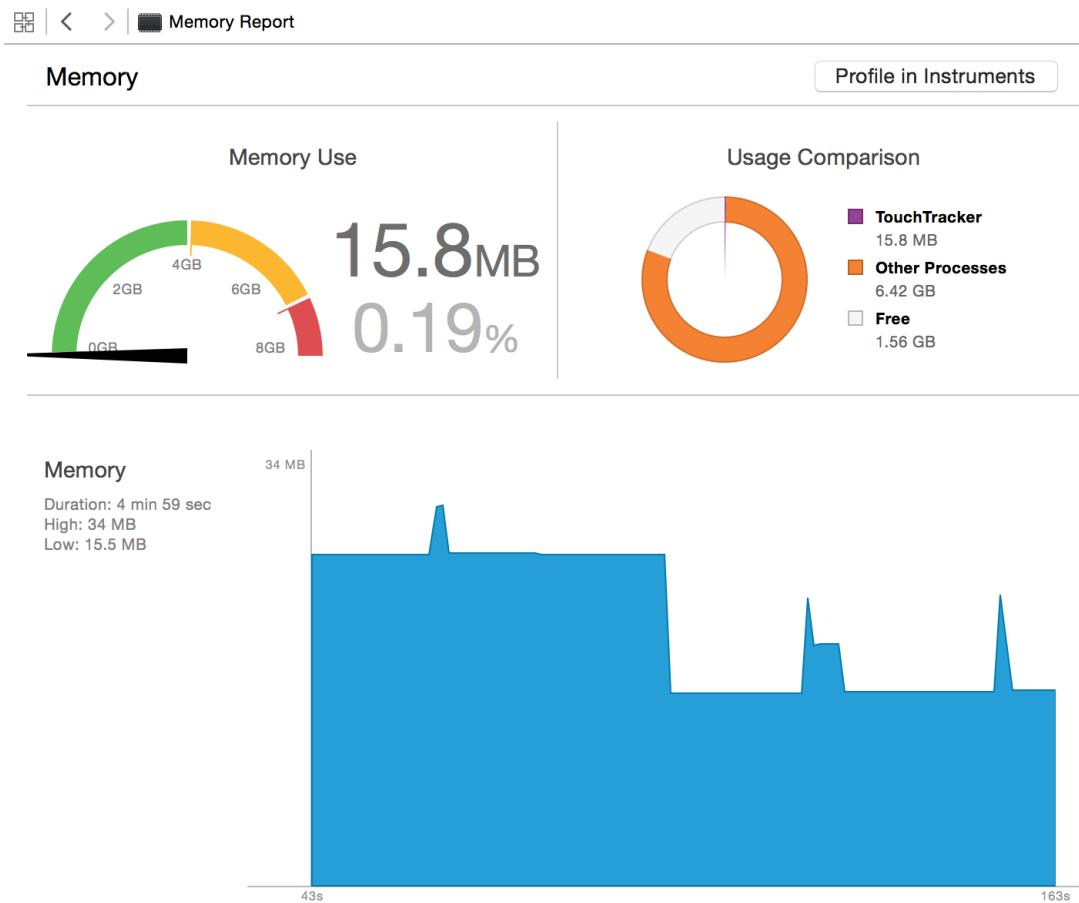
shows a the breakdown of the Usage over Time graph on a per-thread basis. Multithreading is outside the scope of this book, but this information will become useful to you as you continue your iOS development education and career.

To make the graph a bit less boring, begin drawing a line but continue moving your finger without ever letting the line lock into place. This will cause a sustained spike in CPU usage.

Why? Each point on the screen that your finger moves on causes a turn of the application's run loop beginning with a `touchesMoved(_:withEvent:)` message, which in turn causes `drawRect(_:)` to be called on your `DrawView` instance. The more work that you do in these methods, the more CPU utilization your application will require while lines are being drawn.

Next, in the debug navigator, click on the Memory Debug Gauge to present the Memory Report (Figure 22.3).

Figure 22.3 Memory report



Like the CPU Report, the Memory Report is broken down into easy-to-read sections. Do not be alarmed if your Memory graph (the bottom section) appears to be at 100%; this graph scales so that your peak memory usage represents 100% visually.

It is a general goal of software development for any platform to keep both CPU and memory utilization as low as possible, to maximize application performance for the user. It is a good idea to get in the habit of checking these gauges and reports early and often in your projects so that you will be more likely to notice when a change that you have made to your code has resulted in an unexpected change in your application's resource usage.

Instruments

The gauges and reports provide easy and quick access to a high-level understanding of your application's resource usage. If your CPU or memory usage seems higher than it should be, or if your application feels sluggish, you need more information than the gauges and reports provide.

Instruments is an application bundled with Xcode that you can use to monitor your application while it is running and gather fine-grained statistics about your application's performance. Instruments is made up of

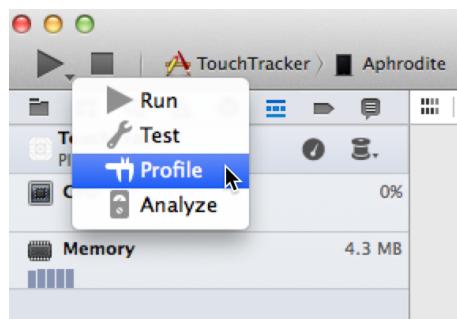
several plug-ins that enable you to inspect object allocations, CPU utilization per function or method, file I/O, network I/O, and much more. Each plug-in is known as an **Instrument**. Together, they help you track down performance deficits in your application.

Allocations instrument

The Allocations instrument tells you about every object that has been created and how much memory it takes up. When you use an instrument to monitor your application, you are *profiling* the application. As with the debug gauges, you can profile the application running on the simulator, but you will get more accurate data on a device.

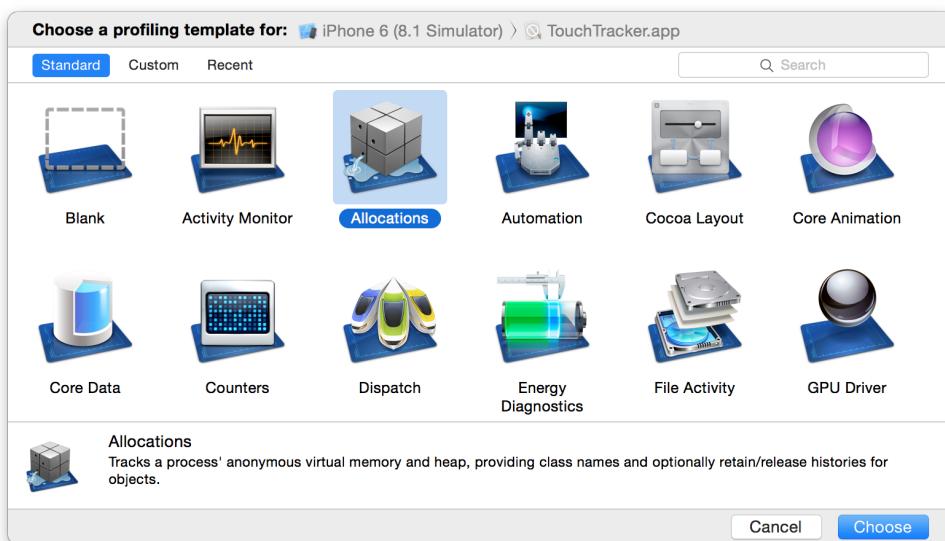
To profile an application, click and hold the Run button in the top left corner of the workspace. In the pop-up menu that appears, select Profile (Figure 22.4).

Figure 22.4 Profiling an application



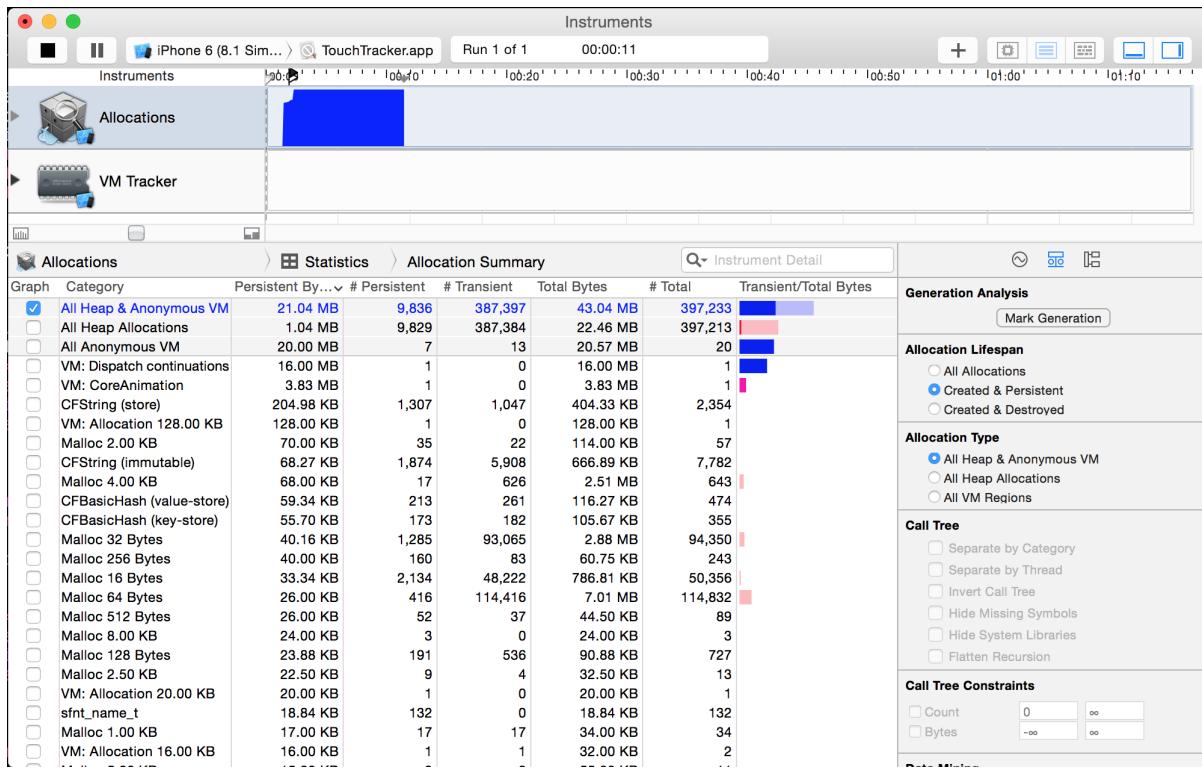
Instruments will launch and ask which instrument template to use. Note that there are more than twelve choices; you will see more if you scroll down. Choose Allocations and click Choose (Figure 22.5).

Figure 22.5 Choosing an instrument



TouchTracker will launch, and a window will open in Instruments (Figure 22.6). The interface may be overwhelming at first, but, like Xcode's workspace window, it will become familiar with time and use. First, make sure you can see everything by turning on all of the areas in the window. In the View control at the top of the window, click the two buttons to reveal the two main areas. The window should look like Figure 22.6.

Figure 22.6 Allocations instrument



To begin profiling your application, click on the red Record button in the top left corner. The application will start up on the device or simulator you selected.

The table shows every memory allocation in the application. There are a lot of objects here, but let's look at the objects that your code is responsible for creating. First, draw some lines in TouchTracker. Then, type Line in the Instrument Detail search box in the top right corner of the window.

This will filter the list of objects in the Object Summary table so that it only shows instances of **Line** (Figure 22.7).

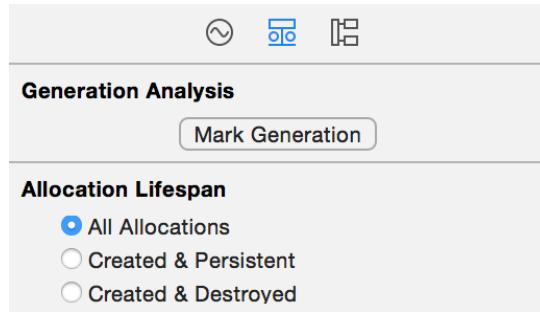
Figure 22.7 Allocated lines

Graph	Category	Persistent Bytes	# Persistent	# Transient	Total Bytes	# Total	# Transient/Total
<input type="checkbox"/>	TouchTracker.Line	256 Bytes	4	0	256 Bytes	4	

The **# Persistent** column shows you how many line objects are currently allocated. **Persistent Bytes** shows how much memory these living instances take up. The **# Total** column shows you how many lines have been created during the course of the application – even if they have since been deallocated.

As you would expect, the number of lines living and the number of lines overall are equal at the moment. Now double-tap the screen in TouchTracker and erase your lines. In Instruments, notice that the **Line** instances disappear from the table. The Allocations instrument is currently set to show only objects that are created and still living. To change this, select All Allocations from the Allocation Lifespan section of the righthand panel (Figure 22.8).

Figure 22.8 Allocations options



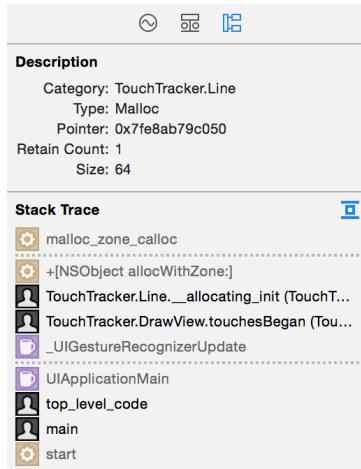
Let's see what else the Allocations instrument can tell you about your lines. First, draw a few more lines in TouchTracker. Then, in the table, select the row that says `TouchTracker.Line`. An arrow will appear in the Category column; click that arrow (known as a focus button) to see more details about these allocations (Figure 22.9).

Figure 22.9 Line summary

#	Address	Category	Timestamp	Live	Size	Responsible Library	Responsible Caller
0	0x7fe8ab79c050	TouchTracker.Line	01:19.237.654	•	64 Bytes	TouchTracker	TouchTracker.Line._alloc...
1	0x7fe8ab45d860	TouchTracker.Line	01:19.858.212	•	64 Bytes	TouchTracker	TouchTracker.Line._alloc...
2	0x7fe8acb13db0	TouchTracker.Line	01:20.405.486	•	64 Bytes	TouchTracker	TouchTracker.Line._alloc...
3	0x7fe8ab78ba90	TouchTracker.Line	01:21.042.700	•	64 Bytes	TouchTracker	TouchTracker.Line._alloc...

Each row in this table shows a single instance of `Line` that has been created. Select one of the rows and check out the stack trace that appears in the Extended Detail area on the right side of the Instruments window (Figure 22.10). This stack trace shows you where that instance of `Line` was created. Grayed-out items in the trace are system library calls. Items in black text are your code. Find the item that is your code that says `TouchTracker.DrawView.touchesBegan ...` and double-click it.

Figure 22.10 Stack trace



The source code for this implementation will replace the table of `Line` instances (Figure 22.11). The percentages you see are the amount of memory these method calls allocate compared to the other calls in `touchesBegan(_:_withEvent:)`. For example, the `Line` instance makes up about 1.7 percent of the memory allocated by `touchesBegan(_:_withEvent:)`. If you don't see percentages, you can click the gear icon above the source code and select View as Percentage.

Figure 22.11 Source code in Instruments

The screenshot shows the Xcode Instruments interface with the Allocations instrument selected. The timeline at the top shows a single event from 196 to 223. The left pane displays the Swift code for `DrawView.swift`. The right pane, titled "Annotations", shows two memory allocations:

- A large allocation at line 201, size 80.2KB, labeled "15.12%".
- A smaller allocation at line 205, size 4.7KB, labeled "4.65%".

The code itself is as follows:

```
196 }
197 }
198 override func touchesBegan(touches: NSSet, withEvent event: UIEvent) {
199 //    println("touchesBegan:_withEvent:")
200     for touch in touches.allObjects as [UITouch] {
201         let location = touch.locationInView(self)
202
203         //        let newLine = Line(begin: location, end: location)
204         let newLine = Line(begin: location, end: location)
205         let key = NSIndexPath(nonretainedObject: touch)
206         currentLines[key] = newLine
207
208         setNeedsDisplay()
209     }
210
211     override func touchesMoved(touches: NSSet, withEvent event: UIEvent) {
212 //        Let's put in a print statement to see the order of events
213 //        println("touchesMoved:_withEvent:")
214
215         for touch in touches.allObjects as [UITouch] {
216             let key = NSIndexPath(nonretainedObject: touch)
217             currentLines[key]!.end = touch.locationInView(self)
218
219         }
220
221         setNeedsDisplay()
222     }
223 }
```

Notice that above the summary area is a breadcrumb navigation bar (Figure 22.12). You can click on an item in this bar to return to a previous set of information.

Figure 22.12 Navigation for summary area



Click on the TouchTracker.Line item in the breadcrumb bar to get back to the list of all **Line** instances. Click on a single instance and then click the arrow icon on that row. This will show you the history of this object. There are two events: when the **Line** was created and when it was destroyed. You can select an event row to see the stack trace that resulted in the event in the extended detail area.

Generation analysis

The last item we will examine in the Allocations instrument is Generation Analysis (a.k.a. Heapshot Analysis). First, clear the search box so that you are not filtering results anymore. Then, find the Generation Analysis category on the left side of the Instruments window and click Mark Generation. A category named Generation A will appear in the table. You can click the disclosure button next to this category to see all of the allocations that took place before you marked the generation. Now draw a line in TouchTracker and click Mark Generation again. Another category will appear named Generation B. Click the disclosure button next to Generation B (Figure 22.13).

Figure 22.13 Generation analysis

Snapshot	Timestamp	Growth	# Persistent
▶ Generation A ↴	00:09.129.546	17.15 MB	9,664
▼ Generation B	00:22.286.202	4.00 MB	790
▶ TouchTracker.Line		64 Bytes	1

Every allocation that took place after the first generation is in this category. You can see the **Line** instance that you just created as well as a few objects that were used to handle other code during this time. You can mark as many generations as you like; they are very useful for seeing what objects get allocated for a specific event. Double-tap the screen in TouchTracker to clear the lines and notice that the objects in this generation disappear.

Generation analysis is most useful for identifying trends in memory usage by creating a closed circuit test and repeating it while marking the generation after each iteration. For example, you might draw four lines and then double-tap to dismiss them, and then mark a generation. Draw four more lines, dismiss them, and mark another generation.

In a perfect world, you would see net zero still-alive allocations between the two generations. In reality, there will be lots of small allocations from Apple's frameworks present. You only need to worry about your own objects. If, for example, you notice that your lines are not deallocating between generations, that represents a problem (memory leak) that you need to fix.

To return to the full object list where you started, select the pop-up button in the breadcrumb bar that currently says Generations and change it to Statistics.

Time Profiler instrument

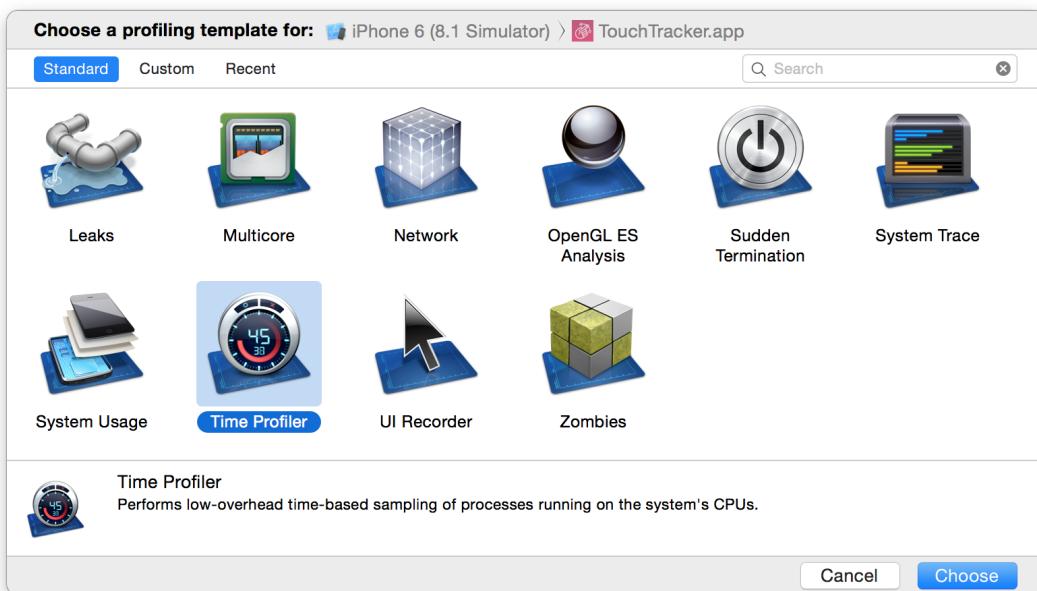
The Time Profiler instrument provides exhaustive statistics about the CPU utilization of your application. Right now, TouchTracker does not abuse the CPU enough to provide meaningful results.

In `DrawView.swift`, make things more interesting by adding the following CPU cycle-wasting code to the end of your `drawRect(_)` method:

```
var f = 0.0
for i in 0..<1_000_000 {
    f = f + sin(sin(sin(Double(time(nil)) + i)))
}
println("f = \(f)")
```

Build and profile the application. When Instruments asks which instrument to use, choose Time Profiler (Figure 22.14). When Instruments launches the application and its window appears, make sure that all three areas are visible by clicking the buttons in the View control to blue.

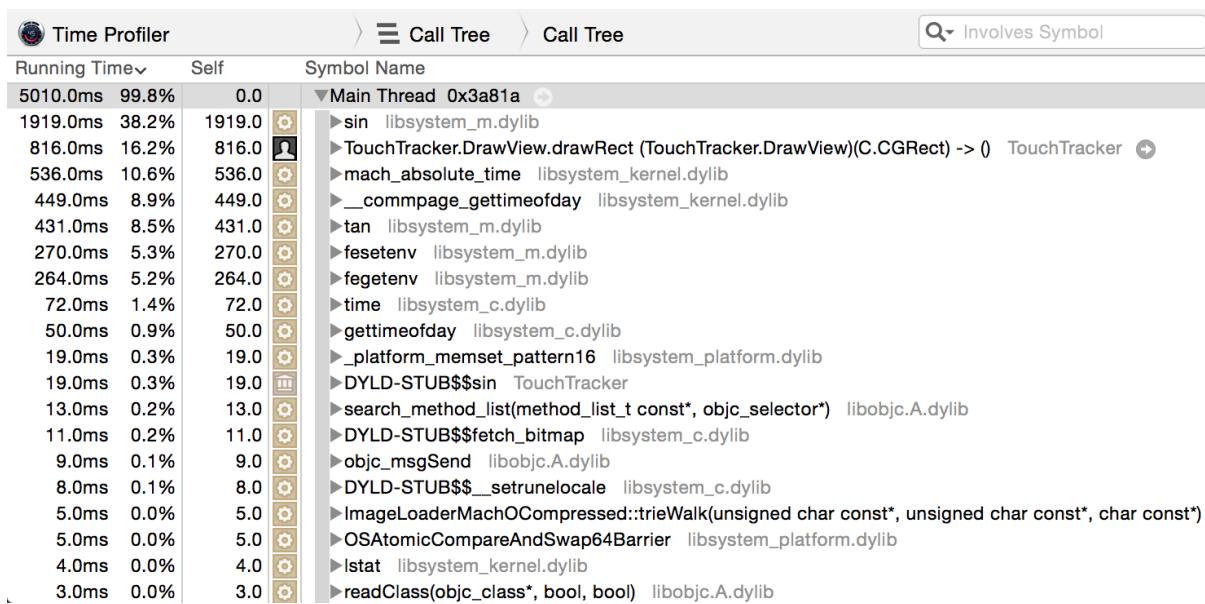
Figure 22.14 Time Profiler instrument



Start recording (you can use the shortcut Command-R), and then touch and hold your finger on the TouchTracker screen. Move your finger around but keep it on the screen. This sends `touchesMoved(_:_withEvent:)` over and over to the `DrawView`. Each time `touchesMoved(_:_withEvent:)` is called, it causes `drawRect(_)` to be called, which in turn causes the silly `sin` code to run repeatedly.

It looks like not much is happening in this instrument, but that is because you are looking at it from the wrong angle: you only see how much time is spent in each of the threads this application is employing. Click the pause button in the top lefthand corner of Instruments and then, in the righthand panel, check the box titled **Invert Call Tree**. Each row in the table is now one function or method call. In the left column, the amount of time spent in that function (expressed in milliseconds and as a percentage of the total run time) is displayed (Figure 22.15). This gives you an idea of where your application is spending its execution time.

Figure 22.15 Time Profiler results



There is no rule that says, “If X percentage of time is spent in this function, your application has a problem.” Instead, use Time Profiler if you notice your application acting sluggish while testing it as a user. For example, you should notice that drawing in TouchTracker is less responsive since you added the wasteful sin code.

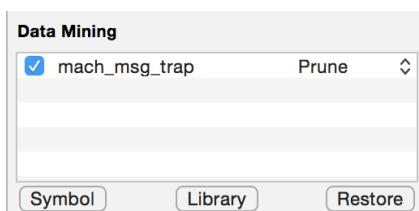
You know that when drawing a line, two things are happening: `touchesMoved(_:withEvent:)` and `drawRect(_:)` are being sent to the `DrawView` view. In Time Profiler, you can check to see how much time is spent in these two methods relative to the rest of the application. If an inordinate amount of time is being spent in one of these methods, you know that is where the problem is.

(Keep in mind that some things just take time. Redrawing the entire screen every time the user’s finger moves, as is done in TouchTracker, is an expensive operation. If it was hindering the user experience, you could find a way to reduce the number of times the screen is redrawn. For example, you could redraw only every tenth of a second regardless of how many touch events were sent.)

Time Profiler shows you nearly every function and method call in the application. If you want to focus on certain parts of the application’s code, you can prune down its results. For example, sometimes the `mach_msg_trap` function will be very high on the sample list. This function is where the main thread sits when it is waiting for input. It is not a bad thing to spend time in this function, so you might want to ignore this time when looking at your Time Profiler results.

Use the search box in the top right corner of the Instruments window to find `mach_msg_trap`. Then, select it from the table. On the right side of the screen, find the Data Mining section and then click the Symbol button at the bottom of that section. The `mach_msg_trap` function appears in the table under Data Mining, and the pop-up button next to it displays Charge. Click on Charge and change it to Prune. Then, clear the text from the search box. Now the list is adjusted so that any time spent in `mach_msg_trap()` is ignored. You can click on Restore while `mach_msg_trap` is selected in the Data Mining table to add it back to the total time.

Figure 22.16 Pruning a symbol



Other options for reducing the list of symbols in Time Profiler include hiding system libraries, flattening recursion, and charging calls to callers. The first two are obvious, but let’s look at charging calls to callers.

Select the row that holds `mach_absolute_time` (or some method that begins with that name). Then, click the Symbol button. This function disappears from the main table and reappears in the Data Mining table. Notice that it is listed as a Charge. This means that the time spent in this function will be attributed to the function or method that called it.

Back in the main table, notice that `mach_absolute_time` has been replaced with the function that calls it, `gettimeofday`. If you take the same steps to charge `gettimeofday`, it will be replaced with its caller, `time`. If you charge `time`, it will be replaced with its caller, `TouchTracker.DrawView.drawRect`. The `drawRect(_)` method will move to near the top of the list; it now is now charged with `time`, `gettimeofday`, and `mach_absolute_time`.

Some common function calls always use a lot of CPU time. Most of the time, these are harmless and unavoidable. For example, the `objc_msgSend()` function is the central dispatch function for any Objective-C message. Since Apple's frameworks are still build with Objective-C, `objc_msgSend()` function occasionally creeps to the top of the list when you are sending lots of messages to objects. Usually, it is nothing to worry about. However, if you are spending more time dispatching messages than actually doing work in the triggered methods *and* your application is not performing well, you have a problem that needs solving.

Do not forget to remove the CPU cycle-wasting code in `drawRect(_)`!

Leaks instrument

Another useful instrument is Leaks. Although this instrument is less useful now that ARC handles memory management, there is still a possibility of leaking memory with a strong reference cycle. Leaks can help you find strong reference cycles.

First, you need to introduce a strong reference cycle into your application. Pretend that every `Line` needs to know what array of lines it belongs to. Add a new property to `Line.swift`:

```
class Line: NSObject {
    var begin: CGPoint = CGPointZero
    var end: CGPoint = CGPointZero

    var containingArray: [Line]?
```

In `DrawView.swift`, set every completed line's `containingArray` property in `touchesEnded(_:_withEvent:)`.

```
override func touchesEnded(touches: NSSet, withEvent event: UIEvent) {
    // Remove ending touches from dictionary
    for touch in touches.allObjects as [UITouch] {
        let key = NSValue(nonretainedObject: touch)
        if var line = currentLines[key] {
            line.end = touch.locationInView(self)

            finishedLines.append(line)
            currentLines.removeValueForKey(key)

            line.containingArray = finishedLines
        }
    }
    setNeedsDisplay()
}
```

Finally, in `doubleTap(_)` of `DrawView.swift`, comment out the code that removes all of the objects from `finishedLines` and create a new instance of `Array` instead.

```
func doubleTap(gestureRecognizer: UIGestureRecognizer) {
    println("Recognized a double tap")

    currentLines.removeAll(keepCapacity: false)
    finishedLines.removeAll(keepCapacity: false)

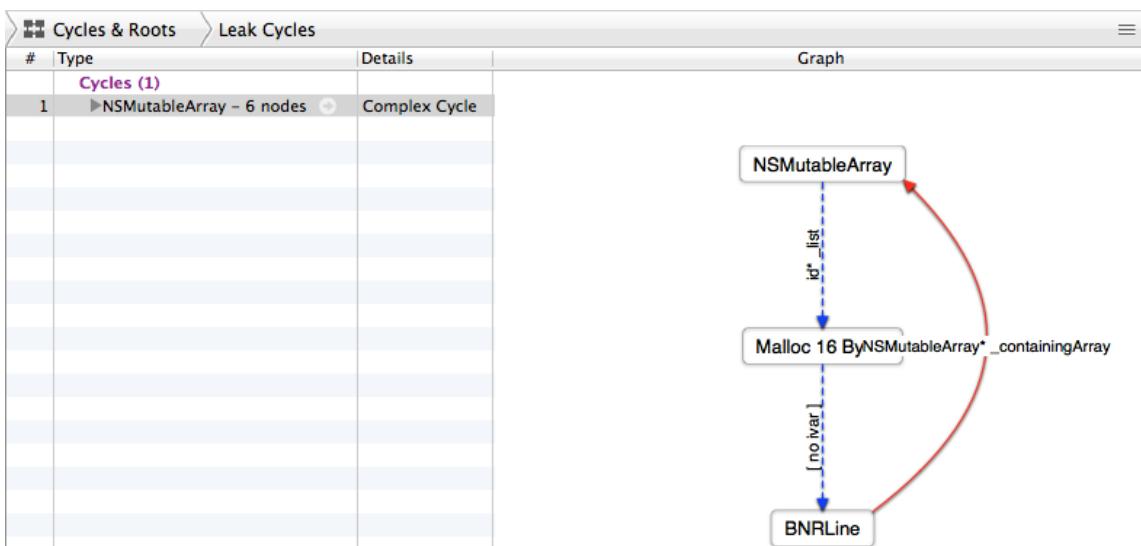
    finishedLines = [Line]()
    setNeedsDisplay()
}
```

Build and profile the application. Choose **Leaks** as the instrument to use, and then start recording a new session.

Draw a few lines and then double-tap the screen to clear it. Select the **Leaks** instrument from the top left table and wait a few seconds. You will see an odd name in the Leaked Objects column: `_TtC12TouchTracker4Line`. (If you don't see anything show up, try again; Instruments does not seem to work perfectly with Swift yet.) This is the *mangled name* of our `Line` class. The mangled name is how Swift actually refers to classes, methods, and variables. You'll notice the project name and the class name which is all you really need to know and look for.

Select the **Leaks** pop-up button in the breadcrumb bar and change it to **Cycles & Roots** (Figure 22.17). This view gives you a lovely graphical representation of the strong reference cycle. You can see the mangled names for the `Line` instances and some `NSArray` information, all of which forms a loop, our strong reference cycle.

Figure 22.17 Cycles and roots



You can of course fix this problem by making the `containingArray` property a weak reference. Or just remove the property and undo your changes to `touchesEnded(_:_withEvent:)` and `doubleTap(_:_)`.

This should give you a good start with the Instruments application. The more you play with it, the more adept at using it you will become. One final word of warning before you invest a significant amount of your development time using Instruments: If there is no performance problem, do not fret over every little row in Instruments. It is a tool for diagnosing existing problems, not for finding new ones. Write clean code that works first; then, if there is a problem, you can find and fix it with the help of Instruments.

Projects, Targets, and Build Settings

An Xcode project is a file that contains a list of references to other files (source code, resources, frameworks, and libraries) as well as a number of settings that lay out the rules for items within the project. Projects end in `.xcodeproj`, as in `TouchTracker.xcodeproj`.

A project always has at least one *target*. When you build and run, you build and run the target, not the project. A target uses the files in the project to build a particular *product*. The product that the target builds is typically an application, although it can be a compiled library or a unit test bundle.

When you create a new project and choose a template, Xcode automatically creates a target for you. When you created the TouchTracker project, you selected an iOS application template, so Xcode created an iOS application target and named it `TouchTracker`.

To see this target, select the `TouchTracker` item at the very top of the project navigator's list. In the editor area just to the right of this item, find a toggle button (Figure 22.18). Click this button to show the project and targets list for `TouchTracker`.

Figure 22.18 TouchTracker project and targets list

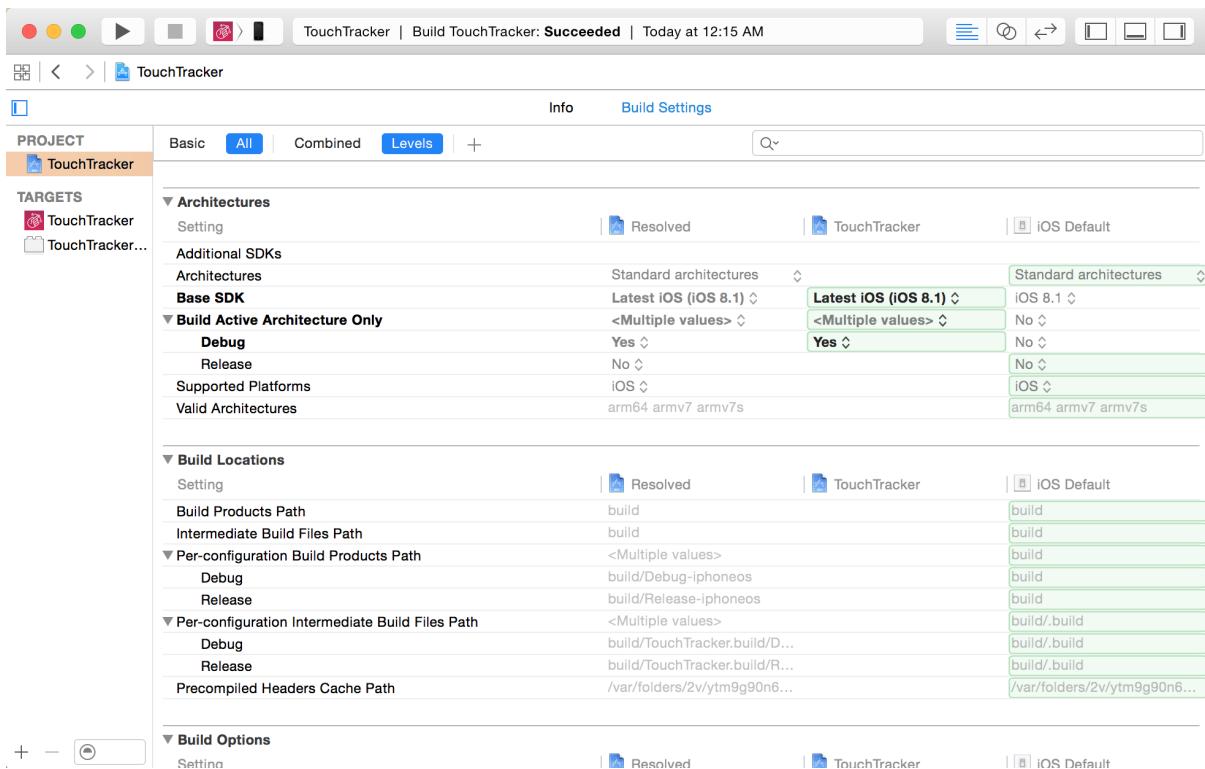
Click to show/hide
project and targets list



Every target includes *build settings* that describe how the compiler and linker should build your application. Every project also has build settings that serve as defaults for the targets within the project.

Let's look at the project build settings for TouchTracker first. In the project and targets list, select the TouchTracker project. Then click the Build Settings tab at the top of the editor area and the Levels button (Figure 22.19).

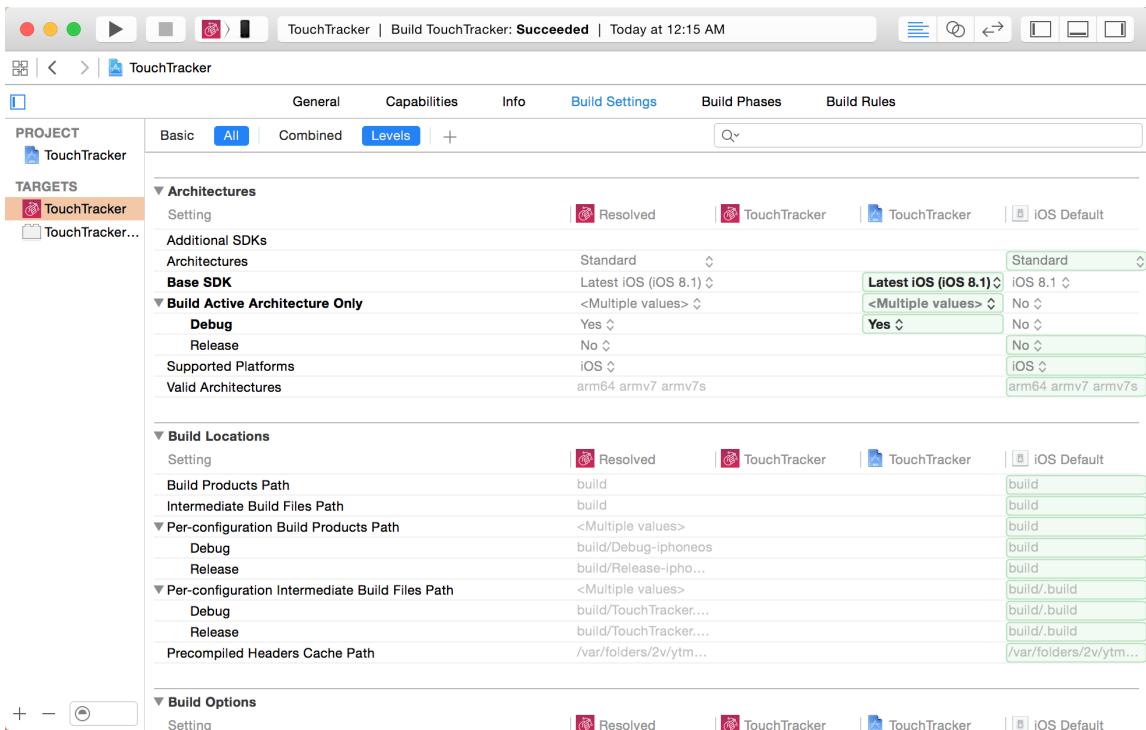
Figure 22.19 TouchTracker project build settings



These are the project-level build settings – the default values that targets inherit. In the top-right corner is a search box that you can use to search for a specific setting. Start typing “Base SDK” in the box, and the list will adjust to show this setting. (The Base SDK setting specifies the version of the iOS SDK that should be used to build your application. It should always be set to the latest version.)

Now let's look at the target's build settings. In project and targets list, select the TouchTracker target and then the Build Settings tab. These are the build settings for this specific target. Make sure the Levels option is selected (Figure 22.20).

Figure 22.20 TouchTracker target build settings



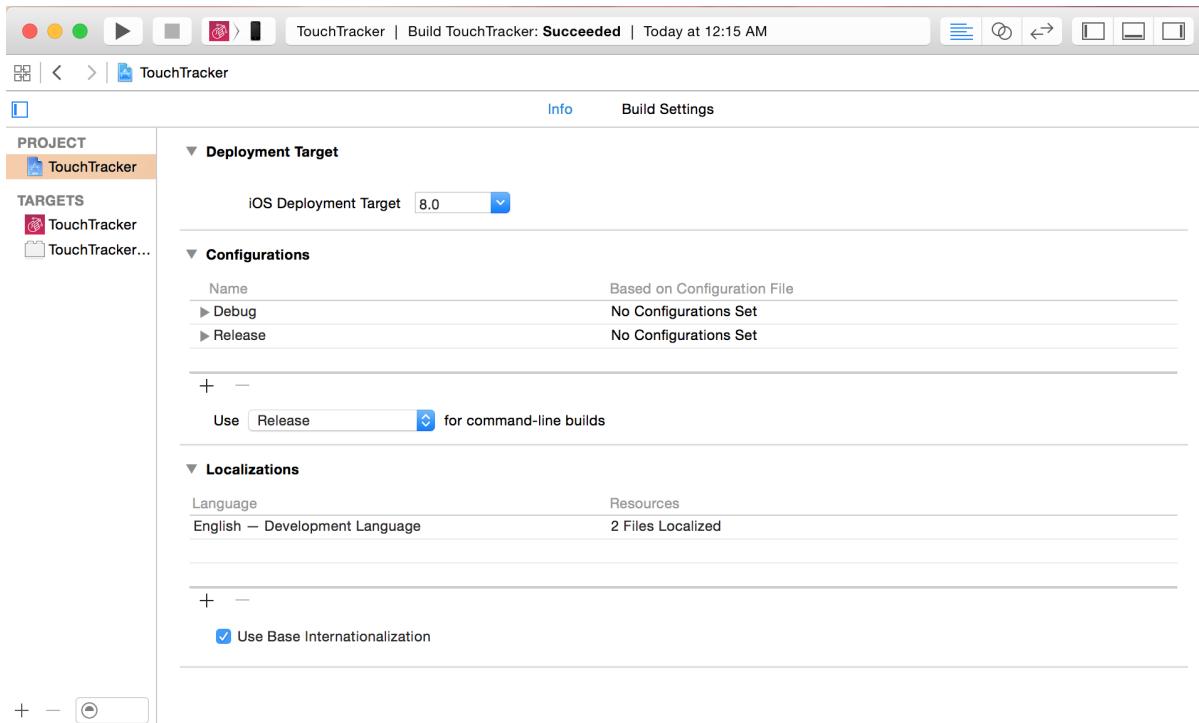
When viewing the build settings with this option, you can see each setting's value at the three different levels: OS, project, and target. The far right column shows the iOS Default settings; these serve as the project's defaults, which it can override. The previous column shows the project's settings, and the one before that shows the currently selected target's settings. The Resolved column shows which setting will actually be used; it is always equal to the left-most specified value. You can click in each column to set the value for that level.

Build configurations

Each target and project has multiple *build configurations*. A build configuration is a set of build settings. When you create a project, there are two build configurations: debug and release. The build settings for the debug configuration make it easier to debug your application, while the release settings turn on optimizations to speed up execution.

To see the build settings and configurations for TouchTracker, select the project from the project navigator and the TouchTracker project. Then, select the Info tab (Figure 22.21).

Figure 22.21 Build configurations list



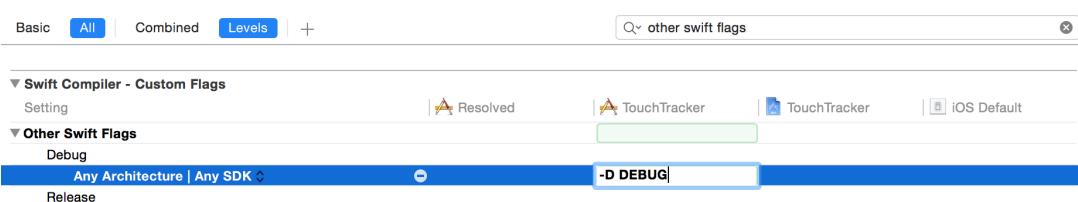
The Configurations section shows you the available build configurations in the project and targets. You can add and remove build configurations with the buttons at the bottom of this section.

Changing a build setting

Enough talk – time to do something useful. You are going to change the value of the target build setting Preprocessor Macros. Preprocessor macros allow you to compile code conditionally. They are either defined or not defined at the start of a build. If you wrap a block of code in a preprocessor directive, it will only be compiled if that macro has been defined. The Preprocessor Macros setting lists preprocessor macros that are defined when a certain build configuration is used by a scheme to build a target.

In the project and targets list, select the TouchTracker target and the Build Settings tab. Then search for the Other Swift Flags build setting. Click the disclosure triangle next to Other Swift Flags to reveal the different build configurations, and then double-click on the value column for the Debug configuration under Preprocessor Macros. In the table that appears, add a new item: -D DEBUG, as shown in Figure 22.22.

Figure 22.22 Changing a build setting



Adding this value to this setting says, “When you are building the TouchTracker target with the debug configuration, a flag named DEBUG is defined.” The -D stands for -Define, indicating that we are defining this flag named DEBUG.

Let’s add some debugging code to TouchTracker that will only be compiled when the target is built with the debug configuration. Printing to the console is very useful, but you usually don’t want to do this in release builds. You will add a function that only prints a **String** to the console in debug configurations.

In `AppDelegate.swift`, add the following code to the top of the file.

```
import UIKit

func printStringInDebug(text: String) {
    #if DEBUG
        println(text)
    #endif
}

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {
```

Now you can call this function from your code. Open `DrawView.swift` and replace any `println()` function calls with calls to `printStringInDebug()`.

Now build and run the application. Check out the console and you will still see the log messages, but they won't be included (and therefore be visible to the rest of the world) when building the app for release.

23

Afterword

Welcome to the end of the book! You should be very proud of all your work and all that you have learned. Now there is good news and bad news:

- *The good news:* The stuff that leaves programmers befuddled when they come to the iOS platform is behind you now. You are an iOS developer.
- *The bad news:* You are probably not a very good iOS developer.

What to do Next

It is now time to make some mistakes, read some really tedious documentation, and be humbled by the heartless experts who will ridicule your questions. Here is what we recommend:

Write apps now. If you do not immediately use what you have learned, it will fade. Exercise and extend your knowledge. Now.

Go deep. This book has consistently favored breadth over depth; any chapter could have been expanded into an entire book. Find a topic that you find interesting and really wallow in it – do some experiments, read Apple’s docs on the topic, read a posting on a blog or on StackOverflow.

Connect. There is an iOS Developer Meetup in most cities, and the talks are surprisingly good. There are CocoaHeads chapters around the world. There are discussion groups online. If you are doing a project, find people to help you: designers, testers (AKA guinea pigs), and other developers.

Make mistakes and fix them. You will learn a lot on the days when you say, “This application has become a ball of crap! I’m going to throw it away and write it again with an architecture that makes sense.” Polite programmers call this *refactoring*.

Give back. Share the knowledge. Answer a dumb question with grace. Give away some code.

Shameless Plugs

You can find all of us on Twitter, where we keep you informed about programming and entertained about life: @aaronhillegass, @cbkeur and @joeconwaystk.

Keep an eye out for future guides from Big Nerd Ranch. We also offer week-long courses for developers. And if you just need some code written, we do contract programming. For more information, visit our website at <http://www.bignerdranch.com/>.

It is you, dear reader, who makes our lives of writing, coding, and teaching possible. So thank you for buying our book.

Index

Symbols

(**drawRect(_:)**, 27-35
.xcassets (asset catalog), 17
.xcdatamodeld (data model file), 198
// MARK:, 125, 126
@selector(), 106
`_cmd`, 139

A

accessor methods,
(see also properties)
accessory indicator (**UITableViewCell**), 75
action methods, 12-13
connecting in XIB file, 112-113
and **UIControl**, 277-278
active state, 134
addKeyframeWithRelativeStartTime(_:relativeDuration:animations:), 232, 233
addSubview(view:), 24, 26
alignment rectangles, 147, 148
Allocations instrument, 292-296
ambiguous layout, 159-162
analyzing (code), 301
animateKeyframesWithDuration(_:delay:options:animations:completion:), 232
animateWithDuration(_:delay:options:animations:completion:), 231
animateWithDuration:animations:, 230-231
animations
basic, 229-231
with closures, 230
keyframe, 232-234
timing functions, 231-231
anti-aliasing, 67
API Reference, 29-30, 128
App ID, 16
application bundle
explained, 141-143
and internationalization, 226
mainBundle, 86
NSBundle, 86
application sandbox, 129-130, 142
application states, 133-135, 139-140
applicationDidBecomeActive:, 135, 139
applicationDidEnterBackground:, 135, 139
applications,
(see also application bundle, debugging, projects, universal applications)
build settings for, 300-303
building, 15, 221
cleaning, 221
data storage, 129-130, 211
deploying, 15-16
directories in, 129-130

icons for, 16-18
multiple threads in, 256
optimizing CPU usage, 296-298
profiling, 292-292
running on iPad, 2
running on simulator, 15
targets and, 299
universalizing, 145-146
applicationWillEnterForeground:, 135, 139
applicationWillResignActive:, 135, 139
ARC (Automatic Reference Counting),
(see also memory management)
archiving
vs. Core Data, 197, 197
described, 127
implementing, 127-129
with **NSKeyedArchiver**, 130-133
when to use, 211
and XIB files, 129
arrays
writing to filesystem, 140
asset catalogs (Xcode), 17
assistant editor (Xcode), 99-102, 112-113
attributes (Core Data), 198, 198
attributes inspector, 9
authentication challenge, 258
Auto Layout,
(see also constraints, Interface Builder)
alignment rectangles, 147, 148
ambiguous layout, 159-162
autoresizing masks and, 173, 173
debugging, 158-165
layout attributes, 147, 148
misplaced views, 163
missing constraints, 159-162
purpose, 147
unsatisfiable constraints, 163
autoresizing masks, 167, 173, 173
autorotation, 185, 187
awakeFromNib, 201

B

background state, 134-135, 134, 139-140
backgroundColor (**UIView**), 24, 32, 33
Base internationalization, 216
baselines, 148
basic animations, 229-231
bounds, 27
build configurations, 301
build settings, 300-303
bundles
application (see application bundle)
identifiers for, 16
NSBundle, 141, 226

C

CALayer, 21

callbacks, , 43, 44, 138
 (see also delegation, notifications, target-action pairs)

camera,
 (see also images)
 taking pictures, 111-116

`cancelTouchesInView`, 287

`canPerformAction(_:withSender:)`, 287

canvas (Interface Builder), 6

cells (see `UITableViewCell`)

`CGPoint`, 23

`CGRect`, 23-25

`CGSize`, 23

class extensions,
 (see also header files)

classes,
 (see also *individual class names*)
 copying files, 72
 reusing, 72

closures
 animation and, 230

`_cmd`, 139

concurrency, 256

connections (in Interface Builder), 10-14

connections inspector, 14

constraints (Auto Layout)
 activating programmatically, 169
 align, 155-156
 creating in Interface Builder, 149-158
 creating programmatically, 168
 creating with VFL, 168-169
 creating without VFL, 172
 debugging, 158-165
 deleting, 153
 intrinsic content size, 170-172
 missing, 159-162
 nearest neighbor and, 149
 overview, 148
 pin, 150-154
 priorities, 158, 171
 unsatisfiable, 163

`constraintsWithVisualFormat(_:options:m... ...etrics:views:)`, 169

`constraintsWithVisualFormat:options:met... ...rics:views:`, 169

`constraintWithItem:attribute:relatedBy:... ...toItem:attribute:multiplier:constant:`, 172

content compression resistance priority, 171

content hugging priority, 171

contentMode (`UIImageView`), 110-110

contentView (`UITableViewCell`), 76-76, 79-80

contentViewController (`UIPopoverController`), 191

control events, 277

controller objects, 4, 138

copying files, 72

Core Data
 vs. archiving, 197, 197

attributes, 198, 198

entities, 198-200, 206-209

faults, 210-210

fetch requests, 203, 211

fetched property, 211

lazy fetching, 210

model configurations, 211

model file, 197-200, 202

`NSManagedObjectContext`, 202-205

`NSManagedObjectModel`, 202-203

`NSPersistentStoreCoordinator`, 202-203

as ORM, 197-198

relationships, 199-200, 210-210

and SQLite, 197, 202-203

subclassing `NSManagedObject`, 200-201

versioning, 211, 211

when to use, 197, 211

credentials (web services), 258-259

`currentLocale`, 215

D

data model file (Core Data), 197-200, 202

data source methods, 74

data storage,
 (see also archiving, Core Data)
 for application data, 129-130

binary, 136, 140

choosing, 211

with `NSData`, 135

`dataSource (UITableView)`, 71, 73-75, 74

debug gauges, 289-291

debugging,
 (see also debugging tools, exceptions)
 Auto Layout, 158-165
`NSError`, 140-140

debugging tools
 Allocations instrument, 292-296
 debug gauges, 289-291
 generation analysis, 295, 296
 Instruments, 291-299
 issue navigator, 15
 static analyzer, 301
 Time Profiler, 296-298

declarations
 protocol, 46

delegation
 protocols used for, 45-46
 vs. target-action pairs, 44

`deleteRowsAtIndexPaths(_:withRowAnimation:... ...on:)`, 90

detail view controllers, 263

developer certificates, 15

developer documentation, 29-30, 37, 128

device orientation, 185, 187

device type
 setting, 145

devices
 checking for camera, 114-115

- deploying to, 15, 16
 display resolution, 25
 provisioning, 15-16, 16
 Retina display, 16, 66-67
 dictionaries,
 (see also JSON data)
 memory management of, 138
 using, 120
 writing to filesystem, 140
 directories
 application, 129-130
 Documents, 129
 Library/Caches, 130
 Library/Preferences, 130
 lproj, 216, 226
 temporary, 130
dismissViewControllerAnimated:completion:
...n:, 191
 display resolution, 25
 dock (Interface Builder), 5
 documentation, developer, 29-30, 37, 128
 Documents directory, 129
 drawing (see views)
drawRect(_:)
 and run loop, 36, 37
drawRect:
 and UITableViewCell, 79
 drill-down interface
 with UINavigationController, 93
 with UISplitViewController, 263
- E**
- editButtonItem**, 107
editing (UITableView, UITableViewDelegate), 83, 87
 editor area (Xcode), 5
encodeInt(_:forKey:), 128
encodeObject(_:forKey:), 128
encodeWithCoder(_:), 131
encodeWithCoder:, 127, 127-128
endEditing(_:), 105, 122
 entities (Core Data), 198-200, 206-209
 errors
 and NSError, 140-140
 event loop, 36, 37
 events
 callbacks and, 138
 control, 277
 run loop and, 36, 37
 touch (see touch events)
 exceptions
 internal inconsistency, 88
exerciseAmbiguousLayout (UIView), 161, 162
- F**
- faults, 210-210
 fetch requests, 203, 211
 fetched property, 211
 file inspector, 216
 file paths, retrieving, 130-130
 File's Owner, 56-59, 56
 first responder
 and nil-targeted actions, 277
 resigning, 105, 122
 and responder chain, 276-277
 and UIMenuController, 283
frame (UIView), 23-25, 27
 frameworks
 Core Data (see Core Data)
 functions,
 (see also *individual function names*)
 callback, 43, 44
- G**
- generation analysis, 295, 296
 genstrings, 224
 gestures,
 (see also UIGestureRecognizer, UIScrollView)
 long press, 284-285
 panning, 39, 284, 285-287
 pinching, 47
 taps, 280-284
 GUIDs, 120
- H**
- .h files (see header files)
hasAmbiguousLayout (UIView), 161
 header view (UITableView), 83-87
 heapshots, 295, 296
 hierarchies
 view, 21-27
 Homeowner application
 adding an image store, 117
 adding Auto Layout constraints, 147, 168
 adding drill-down interface, 93-108, 96
 adding item images, 109-123
 adding modal presentation, 187-191
 customizing cells, 79
 enabling editing, 83-91
 localizing, 213-226
 moving to Core Data, 197
 object diagrams, 73, 96
 reusing BNRItem class, 72
 storing images, 135-137
 universalizing, 145-146
 HTTP protocol, 260-261
 HypnoNerd application
 adding tab bar controller, 59-63
 Hypnosister application
 adding a local notification, 63-64
 adding second view controller, 53-59
 creating HypnosisView, 22-23

I

IBAction, 12, 13, 112-113
IBOutlet, 11, 11, 100-102
 icons,
 (see also images)
 application, 16-18
 asset catalogs for, 17
 camera, 111
 identity inspector, 57
 image picker (see **UIImagePickerController**)
imageNamed:, 67
imagePickerController: didFinishPicking...
...MediaWithInfo:, 114
imagePickerControllerDidCancel:, 114
 images,
 (see also camera, icons, **UIImageView**)
 caching, 135-137
 displaying in **UIImageView**, 109-110
 for Retina display, 67
 storing, 117-120
 wrapping in **NSData**, 136
imageWithContentsOfFile(_:), 136
 inactive state, 134
initWithCoder:, 128-129
initWithFrame:, 23
 initializers,
 (see also **init**)
initWithCoder:, 127
initWithContentsOfFile:, 140-140
initWithStyle:, 72
 inspectors (Xcode)
 attribute, 9
 connections, 14
 file, 216
 identity, 57
 size, 171
 instance variables,
 (see also pointers, properties)
 Instruments, 291-299
 Interface Builder,
 (see also Xcode)
 canvas, 5
 connecting objects, 10-14
 connecting with source files, 81, 81, 99-102
 creating objects in, 7-10
 dock, 5
 explained, 5
 making connections in, 99-102
 and properties, 81, 81
 setting outlets in, 11-12, 100, 101
 setting target-action in, 13-13
 simulated metrics, 100
 when to use, 53
 interface files (see header files)
 interface orientation, 185, 187
 internal inconsistency exception, 88
 internationalization, , 213, 226
 (see also localization)
 intrinsic content size, 170-172
 iOS simulator
 low-memory warnings and, 138
 rotating in, 186
 running applications on, 15
 sandbox location, 132
 saving images to, 115
 viewing application bundle in, 142
 iPad,
 (see also devices)
 application icons for, 16
 running iPhone applications on, 2
isSourceTypeAvailable:, 114
 issue navigator, 15

J

JSON data, 254-256

K

key-value pairs, 118-119
 keyboard
 dismissing, 122
 number pad, 108
 keyframe animations, 232-234
 keys (in dictionaries), 117-122

L

language settings, 213
 layers (of views), 21-22
 layout attributes, 147, 148
 lazy loading, 50, 64, 65
 Leaks instrument, 298-299
 libraries,
 (see also frameworks)
 object, 6
 Library/Caches directory, 130
 Library/Preferences directory, 130
loadView, 50, 50, 51, 87
 local notifications, 63, 64
Localizable.strings, 224
 localization
 Base internationalization and, 216
 internationalization, 213, 226
 lproj directories, 216, 226
NSBundle, 226
 resources, 216
 strings tables, 222-224
 user settings for, 213
 XIB files, 216
localizedDescription, 140
locationInView:, 282
 low-memory warnings, 116
 lproj directories, 216, 226

M

mach_msg_trap(), 297

- macros, preprocessor, 302-303
 main bundle, 86 (see application bundle)
 main thread, 256
mainBundle, 86, 143
 master view controllers, 263, 267
 memory management
 dictionaries, 119, 138
 and Leaks instrument, 298-299
 optimizing with Allocations instrument, 292-296
UITableViewCell, 77
 memory warnings, 116
 menus (**UIMenuController**), 283-284, 287
 messages,
 (see also methods)
 methods,
 (see also *individual method names*)
 action, 12-13, 277-278
 data source, 74
 protocol, 45
minimumPressDuration, 284
 misplaced views, 163
 missing constraints, 159-162
.mobileprovision files, 16-16
 modal view controllers
 defined, 115
 dismissing, 188-190
 and non-disappearing parent view, 191
 relationships of, 194
 in storyboards, 240-243
 styles of, 190
 transitions for, 191
modalPresentationStyle, 190
modalTransitionStyle, 191
modalViewControllerAnimated, 188-190
 model file (Core Data), 197-200, 202
 model objects, 4
 Model-View-Controller (MVC), 4-5, 138
 Model-View-Controller-Store (MVCS), 138
 multi-threading, 256
 multi-touch, enabling, 273
multipleTouchEnabled (**UIView**), 273
 MVC (Model-View-Controller), 4-5, 138
 MVCS (Model-View-Controller-Store), 138
- ## N
- naming conventions
 cell reuse identifiers, 78
 delegate protocols, 45
 navigation controllers (see **UINavigationController**)
navigationController, 103, 194
navigationItem (**UIViewController**), 105
 navigators (Xcode)
 defined, 3
 issue, 15
 project, 3-3, 3
 nearest neighbor, 149
 Nerdfeed application
 adding **WKWebView**, 257-258
 fetching data, 251-253
 using **UISplitViewController**, 263-265
nextResponder, 276
 NIB files,
 (see also XIB files)
 and archiving, 129
 loading, 55
 loading manually, 81
nibWithNibName:bundle:, 81
nil
 -targeted actions, 277
 notifications (**NSNotificationCenter**), 131-138
 of low-memory warnings, 131
 notifications, local, 63, 64
 notifications, push, 63
NSArray,
 (see also arrays)
NSBundle, 86, 226
NSCoder, 127, 129
 NSCoding protocol, 127-129
NSData, 135
NSDate, 104, 140
NSDateFormatter, 104, 214
NSDictionary,
 (see also dictionaries)
NSError, 140-140
NSExpression, 211
NSFetchRequest, 203, 211
NSIndexPath, 77, 90
NSJSONSerialization, 254
NSKeyedArchiver, 130-133
NSKeyedUnarchiver, 133
NSLayoutConstraint, 168
NSLocale, 215
NSLocalizedString(), 222, 224
NSManagedObject, 200-201, 211
NSManagedObjectContext, 202-205, 211
NSManagedObjectModel, 202-203
NSMutableArray,
 (see also arrays)
NSMutableDictionary,
 (see also dictionaries)
NSNotificationCenter, 131-138
NSNumber, 140
NSPersistentStoreCoordinator, 202-203
NSPredicate, 204
NSSearchPathDirectory, 130
NSSearchPathForDirectoriesInDomains, 130
NSSortOrdering, 211
NSString
 property list serializable, 140
NSStringFromSelector, 139
NSTemporaryDirectory, 130
NSURL, 251-253
NSURLCredential, 259
NSURLRequest, 251-253, 260-261
NSURLSession, 252, 252-254, 258
NSURLSessionAuthChallengeUseCredential, 259

NSURLSessionConfiguration, 253
NSURLSessionDataDelegate (protocol), 259
NSURLSessionDataTask, 253-254, 254, 255, 256
NSURLSessionTask, 252, 260
NSUserDefaults, 130
NSUUID, 120
 number pad, 108

O

objc_msgSend(), 298
 object library, 6, 8
 Object-Relational Mapping (ORM), 197
objectForKey:, 117-119
 objects,
 (see also classes, memory management)
 property list serializable, 140
optional, 45
 optional methods (protocols), 45
 Organizer window (Xcode), 16
orientation (UIDevice), 185
 ORM (Object-Relational Mapping), 197
 outlets
 connecting with source files, 81, 81
 declared as weak, 59
 defined, 10
 setting, 10-12, 99

P

parentViewController, 188-190
pathForResource(_:ofType:), 226
 pixels, 25
 placeholder objects, 56
 pointers
 in Interface Builder (see outlets)
 setting in XIB files, 11-12
 points (vs. pixels), 25
 popover controllers, 191, 267
`// MARK:`, 125, 126
 predicates (fetch requests), 204
predicateWithFormat:, 204
 preferences,
 (see also Dynamic Type, localization)
prepareForSegue:sender:, 246
 preprocessor macros, 302-303
presentedViewController, 194
presentingViewController, 194
presentViewController:animated:completi...
`...on:`, 115
 products, 299
 profiling (applications), 292-292
 project and targets list (Xcode), 299
 project navigator, 3-3, 3
 projects
 build settings for, 300-303
 cleaning and building, 221
 copying files to, 72
 creating, 1-3

defined, 299
 target settings in, 141
 properties
 creating from XIB file, 81, 81
 creating in Interface Builder, 81
 property list serializable objects, 140
protocol, 45
 protocols
 declaring, 46
 delegate, 45-46
 described, 45-45
 NSCoding, 127-129
 NSURLSessionDataDelegate, 259
 optional methods in, 45
 required methods in, 45
 structure of, 45
 UIApplicationDelegate, 135
 UIGestureRecognizerDelegate, 285
 UIImagePickerControllerDelegate, 114, 116-116
 UINavigationControllerDelegate, 116
 UIResponderStandardEditActions, 287
 UIScrollViewDelegate, 45, 46
 UITableViewDataSource, 71, 74-75, 76, 90, 90
 UITableViewDelegate, 71
 UITextFieldDelegate, 122
 provisioning profiles, 16-16
 push notifications, 63
pushViewController:animated:, 102-103

Q

Quartz (see Core Graphics)
 Quick Help, 128
 Quiz application, 1-18

R

reference pages, 29-30, 128
 region settings, 213
 relationships (Core Data), 199-200, 210-210
 reordering controls, 91
 required methods (protocols), 45
requireGestureRecognizerToFail(_:), 288
resignFirstResponder, 105
 resources
 asset catalogs for, 17
 defined, 17, 141
 localizing, 216
 responder chain, 276-277
 responders (see first responder, **UIResponder**)
respondsToSelector:, 45
 Retina display, 16, 66-67
reuseIdentifier (UITableViewCell), 78
 reusing
 classes, 72
 table view cells, 77-78
rootViewController (UINavigationController), 94-97
rootViewController (UIWindow), 51, 52

-
- rotation, handling, 185, 187
rows (`UITableView`)
 adding, 88-89
 deleting, 89-90
 moving, 90-91
run loop, 36, 37
- S**
- Sample Code (documentation), 37
 sandbox, application, 129-130, 142
 schemes, 15, 16
 SDK Guides (documentation), 37
 sections (`UITableView`), 75, 83-83
SEL, 106
`@selector()`, 106
`sendAction(_:to:from:forEvent:)`, 277
`sendActionsForControlEvents(_:)`, 277
`setEditing:animated:`, 87, 107
`setNeedsDisplay() (UIView)`, 37
`setNeedsDisplayInRect(rect:) (UIView)`, 37
`setObjectForKey:`, 117-119
`setPagingEnabled:`, 48
`setStroke() (UIColor)`, 32
 settings (see preferences)
 Settings application, 130
 simulated metrics, 100
 simulator
 low-memory warnings and, 138
 rotating in, 186
 running applications on, 15
 sandbox location, 132
 saving images to, 115
 simulating two fingers, 47
 viewing application bundle in, 142
 size inspector, 171
 sort descriptors (`NSFetchRequest`), 203
 sourceType (`UIImagePickerController`), 113-114
 split view controllers (see `UISplitViewController`)
`splitViewController`, 194
 SQLite, 197, 202-203
 SSL (Secure Sockets Layer), 258, 259
 states, application, 133-135
 static analyzer, 301
 static tables, 238-239
 store objects, 138
 storyboards
 creating, 235-238
 segues, 239-248
 static tables in, 238-239
 vs. XIB files, 235
String
 internationalizing, 222
 writing to filesystem, 136-140
strings (see `NSString`)
 strings tables, 222-224
 strong reference cycles
 finding with Leaks instrument, 298-299
 subclassing,
- (see also overriding methods)
 subviews, 21
 superview, 25
 suspended state, 134, 135
- T**
- tab bar controllers (see `UITabBarController`)
 tab bar items, 61-63
`tabBarController`, 194
 table view cells (see `UITableViewCell`)
 table view controllers (see `UITableViewController`)
 table views (see `UITableView`)
 tables (database), 197
`tableView`, 89
`tableView(_:commitEditingStyle:forRowAt...IndexPath:)`, 90
`tableView(_:didSelectRowAtIndexPath:)`, 104
`tableView(_:moveRowAtIndexPath:toIndexPath...at:)`, 90, 90
`tableView:cellForRowAtIndexPath:`, 75, 76-78
`tableView:numberOfRowsInSection:`, 75-75
 target-action pairs
 defined, 12-13
 vs. delegation, 44
 setting programmatically, 106
 and `UIControl`, 277-278
 and `UIGestureRecognizer`, 279
 targets
 build settings for, 141, 300-303
 defined, 299
 templates (Xcode), xiv
`textFieldShouldReturn:`, 122
 threads, 256
 Time Profiler instrument, 296-298
 timing functions, 231-231
 tmp directory, 130
`toggleEditMode:`, 87
`topViewController (UINavigationController)`, 95
 touch events,
 (see also `UIGestureRecognizer`)
 basics of, 269-270
 enabling multi-touch, 273-276
 and responder chain, 276-277
 and target-action pairs, 277-278
 and `UIControl`, 277-278
`touchesBegan(_:withEvent:)`, 269
`touchesBegan:withEvent:`, 269
`touchesCancelled:withEvent:`, 269
`touchesEnded:withEvent:`, 269, 270
`touchesMoved(_:withEvent:)`, 269
`touchesMoved:withEvent:`, 269
 TouchTracker application
 drawing lines, 270-276
 recognizing gestures, 279-287
`translationInView(_:)`, 286

U

UI thread, 256
UIApplication
 and events, 270
 and responder chain, 276, 277
UIApplicationDelegate, 135
UIApplicationDidEnterBackgroundNotification, 131
UIBarButtonItem, 106-107, 111-113, 123
UIBezierPath, 28-35
UIColor, 24, 32, 32
UIControl, 122-123, 277-278
UIControlEvents.TouchUpInside, 277
UIControlEvents.TouchUpInside, 277
UIGestureRecognizer
 action messages of, 279, 284
 cancelsTouchesInView, 287
 chaining recognizers, 288
 delaying touches, 287
 described, 279
 detecting taps, 280-284
 enabling simultaneous recognizers, 285
 implementing multiple, 281-283, 285-287
 intercepting touches from view, 280, 285, 287
locationInView:, 282
 long press, 284-285
 panning, 284, 285-287
 state (property), 284, 286, 288
 subclasses, 279, 288
 subclassing, 288
translationInView(_:), 286
 and **UIResponder** methods, 287
UIGestureRecognizerDelegate, 285
UIImage, 136
 (see also images, **UIImageView**)
UIImageJPEGRepresentation, 136
UIImagePickerController
 instantiating, 113-114
 on iPad, 191
 presenting, 115-116
 in **UIPopoverController**, 191
UIImagePickerControllerDelegate, 114, 116-116
UIImageView, 109-110
UIInterpolatingMotionEffect, 230
UILocalNotification, 64
UILongPressGestureRecognizer, 284-285
UIMenuController, 283-284, 287
UIModalPresentationStyle.FormSheet, 190
UIModalPresentationStyle.PageSheet, 190
UIModalTransitionStyleCoverVertical, 191
UIModalTransitionStyleCrossDissolve, 191
UIModalTransitionStyleFlipHorizontal, 191
UIModalTransitionStylePartialCurl, 191
UINavigationBar, 95, 97-108
UINavigationController,
 (see also view controllers)
 adding view controllers to, 102-103, 104
 described, 94-98
 instantiating, 97
 managing view controller stack, 94
navigationController, 194
pushViewController:animated:, 102-103
rootViewController, 94-96
 in storyboards, 240, 240
topViewController, 94-96
 and **UINavigationBar**, 105-107
view, 95
viewControllers, 95
viewWillAppear:, 104
viewWillDisappear:, 104
UINavigationControllerDelegate, 116
UINavigationItem, 105-107
UINib, 81
UIPanGestureRecognizer, 284, 285-287
UIPopoverController, 191, 267
UIResponder
 menu actions, 287
 and responder chain, 276-277
 and touch events, 269
UIResponderStandardEditActions (protocol), 287
UIScrollView
 zooming, 46-47
UIScrollViewDelegate, 45, 46
UISplitViewController
 master and detail view controllers, 263
 overview, 263-265
 in portrait mode, 267
splitViewController, 194
UIStoryboard, 235
UIStoryboardSegue, 239-248
UITabBarController
 implementing, 59-63
tabBarController, 194
 vs. **UINavigationController**, 93
view, 60
UITabBarItem, 61-63
UITableView, 69-71
 (see also **UITableViewCell**,
UITableViewcontroller)
 adding rows to, 88-89
 deleting rows from, 89-90
 editing mode of, 79, 83, 87-88, 107
 editing property, 83, 87-88
 footer view, 83
 header view, 83-87
 moving rows in, 90-91
 populating, 73
 sections, 75, 83-83
view, 72
UITableViewCell
 cell styles, 76
contentView, 76-76, 79-80
 creating interface with XIB file, 80-81
 editing styles, 90
 retrieving instances of, 76

reusing instances of, 77-78
 subclassing, 79-82
 subviews, 75-76
UITableViewCellStyle, 76
UITableViewCellStyleDelete, 90
UITableViewController,
 (see also **UITableView**)
 adding rows, 88-89
 creating in storyboard, 238-239
 creating static tables, 238-239
 data source methods, 74
dataSource, 73-75
 deleting rows, 89-90
 described, 71-71
 designated initializer, 72
 editing property, 87
init(style:), 72
 moving rows, 90-91
 returning cells, 76-78
 subclassing, 71-72
tableView, 89
UITableViewStyleGrouped, 72
UITableViewStylePlain, 72
UITableViewDataSource (protocol), 71, 74-75, 76, 90, 90
UITableViewDelegate, 71
UITapGestureRecognizer, 280-284
UITextField
 as first responder, 122, 277
 setting attributes of, 108
UITextFieldDelegate, 122
UIToolbar, 106, 111
UITouch, 269-270, 272, 273, 273-276
UIView,
 (see also **UIViewController**, views)
backgroundColor, 24, 32, 33
bounds, 27
 defined, 20-20
drawRect(_:), 27-35, 36, 37
endEditing:, 105
exerciseAmbiguousLayout, 161, 162
frame, 23-25, 27
hasAmbiguousLayout, 161
 instantiating, 23
setNeedsDisplay(), 37
setNeedsDisplayInRect(rect:), 37
 subclassing, 22-27
 superview, 25
UIViewController,
 (see also **UIView**, view controllers)
 instantiating, xiv
loadView, 50, 50, 51, 87
modalTransitionStyle, 191
modalViewController, 188-190
navigationController, 103
navigationItem, 105
parentViewController, 188-190
tabBarItem, 61
view, 50, 56, 65, 276
viewControllers, 193
viewDidLayoutSubviews, 161
viewDidLoad, 65
viewWillAppear:, 65, 116
 and XIB files, xiv
UIWindow, 20
 and responder chain, 276
rootViewController, 51, 52
unarchiveObjectWithFile(_:), 133
 universal applications
 creating, 145-146
 defined, 145
 unsatisfiable constraints, 163
URLs, , 252
 (see also **NSURL**)
 user interface,
 (see also Auto Layout, views)
 drill-down, 93, 263
 handling rotation, 185, 187
 keyboard, 122
 orientation of, 185, 187
 zooming (views), 46-47
 user settings (see preferences)
 utility area (Xcode), 7
 UUIDs, 120

V

variables,
 (see also instance variables, local variables, pointers, properties)
VFL (Visual Formal Language), 168-168
view (UIViewController), 50, 65
 view controllers,
 (see also **UIViewController**, views)
 adding to navigation controller, 102-103
 adding to split view controller, 263-265
 creating in a storyboard, 235
 defined, 49
 detail, 263
 families of, 193, 194
 lazy loading of views, 50, 64, 65
 loading views, 56
 master, 263, 267
 modal, 115
 passing data between, 103-104
 presenting, 59
 relationships between, 193-195
 reloading subviews, 116
 role in application, 49
 and view hierarchy, 50, 50
 view hierarchy, 21-27, 50
 view controllers, 193
viewControllers (UINavigationController), 95
viewDidLayoutSubviews (UIViewController), 161
viewDidLoad, 65
viewForZoomingInScrollView:, 47
 views,

(see also Auto Layout, touch events, **UIView**, view controllers)
 adding to window, 21, 50
 animating, 229
 creating custom, 22-27
 defined, 20-20
 drawing shapes, 28-35
 drawing to screen, 21-22, 27, 27, 36, 37
 in hierarchy, 21-22
 layers and, 21-22
 lazy loading of, 50, 64
 loading, 56
 modal presentation of, 115
 in Model-View-Controller, 4
 redrawing, 36, 37
 rendering, 21-22, 27, 27, 36, 37
 resizing, 110-110
 and run loop, 36, 37
 size and position of, 23-25, 27
 and subviews, 21-27
 zooming, 46-47
viewWillAppear:, 65, 104-105, 104, 116
viewWillDisappear:, 104
 Visual Formal Language (VFL), 168-168

W

web services
 authentication, 258-259
 credentials, 258-259
 for data storage, 211
 and HTTP protocol, 260-261
 implementing, 251-257
 with JSON data, 254-256
NSURLSession, 252-254
 overview, 250-250
 requesting data from, 251-254
 SSL (Secure Sockets Layer), 258, 259
WKWebView, 257-258
 workspaces (Xcode), 3
writeToFile(_:atomically:), 136
writeToFile:atomically:encoding:error:, 140

X

.xcassets (asset catalog), 17
 .xcdatamodeld (data model file), 198
 Xcode,
 (see also debugging tools, Instruments, Interface Builder, projects, iOS simulator)
 API Reference, 29-30, 128
 asset catalogs, 17
 assistant editor, 99-102, 112-113
 attributes inspector, 9
 build settings, 300-303
 building interfaces, 5-14
 canvas, 6
 creating projects in, 1-3
 debug gauges, 289-291

documentation browser, 29-30
 editor area, 5
 file inspector, 216
 identity inspector, 57
 inspectors, 6
 issue navigator, 15
 keyboard shortcuts, 102
 library, 6
 navigator area, 3
 navigators, 3
 object library, 6, 8
 Organizer window, 16
 products, 299
 profiling applications in, 292-292
 project and targets list, 299
 project navigator, 3
 projects, 299
 Quick Help, 128
 schemes, 15, 16
 size inspector, 171
 static analyzer, 301
 tabs, 102
 targets, 299
 templates, 22
 utility area, 7
 versions, 2
 workspaces, 3
XIB files,
 (see also Interface Builder, NIB files)
 and archiving, 129
 bad connections in, 102
 Base internationalization and, 216
 connecting with source files, 81, 81, 81, 99-102, 112-113
 creating properties from, 81, 81
 File's Owner, 56-59
 loading manually, 86
 localizing, 216
 making connections in, 112-113
 placeholders in, 56
 and properties, 81
 simulated metrics, 100
 vs. storyboards, 235
 when to use, 53
 XML property lists, 140

Z

zooming (views), 46-47