



A large, light-colored scroll or map is shown unrolled, revealing a detailed map of the Grand Canyon. The hands holding the scroll are visible at the top and bottom edges. The background is a dark, solid color.

iOS PROGRAMMING

THE BIG NERD RANCH GUIDE

4TH EDITION

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, first printing, February 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 Matthias, Scott Ritchie, Pouria Almassi, Step Christopher, TJ Usiyan, and Bolot Kerimbaev. 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.

This page intentionally left blank

Table of Contents

| | |
|---|------|
| Introduction | xiii |
| Prerequisites | xiii |
| What has Changed in the Fourth Edition? | xiii |
| Our Teaching Philosophy | xiv |
| How to Use This Book | xv |
| How This Book is Organized | xv |
| Style Choices | xvii |
| Typographical Conventions | xvii |
| Necessary Hardware and Software | xvii |
| 1. A Simple iOS Application | 1 |
| Creating an Xcode Project | 2 |
| Model-View-Controller | 4 |
| Designing Quiz | 5 |
| Creating a View Controller | 6 |
| Building an Interface | 8 |
| Creating view objects | 10 |
| Configuring view objects | 11 |
| NIB files | 13 |
| Making connections | 14 |
| Creating Model Objects | 18 |
| Using code-completion | 20 |
| Pulling it all Together | 21 |
| Implementing action methods | 21 |
| Getting the view controller on the screen | 22 |
| Running on the Simulator | 22 |
| Deploying an Application | 23 |
| Application Icons | 25 |
| Launch Images | 27 |
| 2. Objective-C | 29 |
| Objects | 29 |
| Using Instances | 30 |
| Creating objects | 30 |
| Sending messages | 31 |
| Destroying objects | 32 |
| Beginning RandomItems | 33 |
| Creating and populating an array | 36 |
| Iterating over an array | 37 |
| Format strings | 38 |
| Subclassing an Objective-C Class | 38 |
| Creating an NSObject subclass | 39 |
| Instance variables | 41 |
| Accessing instance variables | 42 |
| Class vs. instance methods | 47 |
| Overriding methods | 48 |
| Initializers | 49 |

| | |
|--|-----|
| Class methods | 55 |
| Testing your subclass | 57 |
| More on NSArray and NSMutableArray | 58 |
| Exceptions and Unrecognized Selectors | 60 |
| Challenges | 62 |
| Bronze Challenge: Bug Finding | 62 |
| Silver Challenge: Another Initializer | 62 |
| Gold Challenge: Another Class | 63 |
| Are You More Curious? | 63 |
| For the More Curious: Class Names | 63 |
| For the More Curious: #import and @import | 64 |
| 3. Managing Memory with ARC | 65 |
| The Stack | 65 |
| The Heap | 65 |
| ARC and memory management | 66 |
| Pointer Variables and Object Ownership | 66 |
| How objects lose owners | 67 |
| Ownership chains | 69 |
| Strong and Weak References | 70 |
| Properties | 75 |
| Declaring properties | 75 |
| Property attributes | 78 |
| Custom accessors with properties | 81 |
| For the More Curious: Property Synthesis | 81 |
| For the More Curious: Autorelease Pool and ARC History | 83 |
| 4. Views and the View Hierarchy | 85 |
| View Basics | 86 |
| The View Hierarchy | 86 |
| Subclassing UIView | 88 |
| Views and frames | 89 |
| Custom Drawing in drawRect: | 94 |
| Drawing a single circle | 96 |
| UIBezierPath | 96 |
| Using the developer documentation | 97 |
| Drawing concentric circles | 102 |
| More Developer Documentation | 105 |
| Bronze Challenge: Draw an Image | 106 |
| For the More Curious: Core Graphics | 106 |
| Gold Challenge: Shadows and Gradients | 108 |
| 5. Views: Redrawing and UIScrollView | 111 |
| The Run Loop and Redrawing Views | 112 |
| Class Extensions | 114 |
| Using UIScrollView | 114 |
| Panning and paging | 117 |
| 6. View Controllers | 119 |
| Subclassing UIViewController | 120 |
| The view of a view controller | 121 |
| Creating a view programmatically | 121 |

| | |
|---|-----|
| Setting the root view controller | 122 |
| Another UIViewController | 123 |
| Creating a view in Interface Builder | 124 |
| UITabBarController | 130 |
| Tab bar items | 132 |
| UIViewController Initializers | 134 |
| Adding a Local Notification | 135 |
| Loaded and Appearing Views | 136 |
| Accessing subviews | 137 |
| Interacting with View Controllers and Their Views | 138 |
| Bronze Challenge: Another Tab | 138 |
| Silver Challenge: Controller Logic | 138 |
| For the More Curious: Key-Value Coding | 138 |
| For the More Curious: Retina Display | 140 |
| 7. Delegation and Text Input | 143 |
| Text Fields | 143 |
| UIResponder | 144 |
| Configuring the keyboard | 145 |
| Delegation | 146 |
| Protocols | 148 |
| Adding the Labels to the Screen | 150 |
| Motion Effects | 151 |
| Using the Debugger | 151 |
| Using breakpoints | 152 |
| Stepping through code | 154 |
| For the More Curious: main() and UIApplication | 156 |
| Silver Challenge: Pinch to Zoom | 156 |
| 8. UITableView and UITableViewController | 159 |
| Beginning the Homepwner Application | 159 |
| UITableViewController | 160 |
| Subclassing UITableViewController | 161 |
| UITableView's Data Source | 164 |
| Creating BNRIItemStore | 165 |
| Implementing data source methods | 169 |
| UITableViewCells | 170 |
| Creating and retrieving UITableViewCells | 171 |
| Reusing UITableViewCells | 173 |
| Code Snippet Library | 174 |
| Bronze Challenge: Sections | 177 |
| Silver Challenge: Constant Rows | 177 |
| Gold Challenge: Customizing the Table | 177 |
| 9. Editing UITableView | 179 |
| Editing Mode | 179 |
| Adding Rows | 185 |
| Deleting Rows | 187 |
| Moving Rows | 188 |
| Bronze Challenge: Renaming the Delete Button | 190 |
| Silver Challenge: Preventing Reordering | 190 |

| | |
|---|-----|
| Gold Challenge: Really Preventing Reordering | 190 |
| 10. UINavigationController | 191 |
| UINavigationController | 192 |
| An Additional UIViewController | 196 |
| Navigating with UINavigationController | 202 |
| Pushing view controllers | 202 |
| Passing data between view controllers | 203 |
| Appearing and disappearing views | 205 |
| UINavigationBar | 205 |
| Bronze Challenge: Displaying a Number Pad | 210 |
| Silver Challenge: Dismissing a Number Pad | 210 |
| Gold Challenge: Pushing More View Controllers | 210 |
| 11. Camera | 211 |
| Displaying Images and UIImageView | 212 |
| Adding a camera button | 213 |
| Taking Pictures and UIImagePickerController | 216 |
| Setting the image picker's sourceType | 216 |
| Setting the image picker's delegate | 218 |
| Presenting the image picker modally | 218 |
| Saving the image | 219 |
| Creating BNRIImageStore | 220 |
| NSDictionary | 222 |
| Creating and Using Keys | 225 |
| Wrapping up BNRIImageStore | 227 |
| Dismissing the Keyboard | 228 |
| Bronze Challenge: Editing an Image | 230 |
| Silver Challenge: Removing an Image | 230 |
| Gold Challenge: Camera Overlay | 230 |
| For the More Curious: Navigating Implementation Files | 230 |
| #pragma mark | 231 |
| For the More Curious: Recording Video | 232 |
| 12. Touch Events and UIResponder | 235 |
| Touch Events | 235 |
| Creating the TouchTracker Application | 236 |
| Drawing with BNRDrawView | 238 |
| Turning Touches into Lines | 240 |
| Handling multiple touches | 241 |
| Bronze Challenge: Saving and Loading | 245 |
| Silver Challenge: Colors | 245 |
| Gold Challenge: Circles | 245 |
| For the More Curious: The Responder Chain | 245 |
| For the More Curious: UIControl | 246 |
| 13. UIGestureRecognizer and UIMenuController | 249 |
| UIGestureRecognizer Subclasses | 250 |
| Detecting Taps with UITapGestureRecognizer | 250 |
| Multiple Gesture Recognizers | 252 |
| UIMenuController | 254 |
| UILongPressGestureRecognizer | 256 |

| | |
|---|------------|
| UIPanGestureRecognizer and Simultaneous Recognizers | 257 |
| For the More Curious: UIMenuController and UIResponderStandardEditActions | 260 |
| For the More Curious: More on UIGestureRecognizer | 260 |
| Silver Challenge: Mysterious Lines | 261 |
| Gold Challenge: Speed and Size | 261 |
| Mega-Gold Challenge: Colors | 261 |
| 14. Debugging Tools | 263 |
| Gauges | 263 |
| Instruments | 265 |
| Allocations instrument | 266 |
| Time Profiler instrument | 271 |
| Leaks instrument | 274 |
| Static Analyzer | 276 |
| Projects, Targets, and Build Settings | 278 |
| Build configurations | 280 |
| Changing a build setting | 281 |
| 15. Introduction to Auto Layout | 283 |
| Universalizing Homepwner | 283 |
| The Auto Layout System | 285 |
| Alignment rectangle and layout attributes | 286 |
| Constraints | 287 |
| Adding Constraints in Interface Builder | 289 |
| Adding more constraints | 293 |
| Adding even more constraints | 296 |
| Priorities | 298 |
| Debugging Constraints | 298 |
| Ambiguous layout | 299 |
| Unsatisfiable constraints | 303 |
| Misplaced views | 304 |
| Bronze Challenge: Practice Makes Perfect | 306 |
| Silver Challenge: Universalize Quiz | 307 |
| For the More Curious: Debugging Using the Auto Layout Trace | 307 |
| For the More Curious: Multiple XIB Files | 308 |
| 16. Auto Layout: Programmatic Constraints | 309 |
| Visual Format Language | 310 |
| Creating Constraints | 311 |
| Adding Constraints | 312 |
| Intrinsic Content Size | 315 |
| The Other Way | 316 |
| For the More Curious: NSAutoresizingMaskLayoutConstraint | 318 |
| 17. Autorotation,Popover Controllers, and Modal View Controllers | 321 |
| Autorotation | 321 |
| Rotation Notification | 324 |
| UIPopoverController | 326 |
| More Modal View Controllers | 329 |
| Dismissing modal view controllers | 332 |
| Modal view controller styles | 334 |
| Completion blocks | 335 |

| | |
|---|-----|
| Modal view controller transitions | 337 |
| Thread-Safe Singletons | 337 |
| Bronze Challenge: Another Thread-Safe Singleton | 339 |
| Gold Challenge: Popover Appearance | 339 |
| For the More Curious: Bitmasks | 339 |
| For the More Curious: View Controller Relationships | 340 |
| Parent-child relationships | 341 |
| Presenting-presenter relationships | 342 |
| Inter-family relationships | 342 |
| 18. Saving, Loading, and Application States | 345 |
| Archiving | 345 |
| Application Sandbox | 348 |
| Constructing a file path | 349 |
| NSKeyedArchiver and NSKeyedUnarchiver | 350 |
| Application States and Transitions | 353 |
| Writing to the Filesystem with NSData | 356 |
| NSNotificationCenter and Low-Memory Warnings | 358 |
| More on NSNotificationCenter | 360 |
| Model-View-Controller-Store Design Pattern | 361 |
| Bronze Challenge: PNG | 361 |
| For the More Curious: Application State Transitions | 361 |
| For the More Curious: Reading and Writing to the Filesystem | 362 |
| For the More Curious: The Application Bundle | 365 |
| 19. Subclassing UITableViewCell | 369 |
| Creating BNRIItemCell | 369 |
| Configuring a UITableViewCell subclass's interface | 371 |
| Exposing the properties of BNRIItemCell | 372 |
| Using BNRIItemCell | 372 |
| Constraints for BNRIItemCell | 374 |
| Image Manipulation | 377 |
| Relaying Actions from UITableViewCells | 380 |
| Adding a block to the cell subclass | 382 |
| Presenting the image in a popover controller | 383 |
| Variable Capturing | 385 |
| Bronze Challenge: Color Coding | 386 |
| Gold Challenge: Zooming | 387 |
| For the More Curious: UICollectionView | 387 |
| 20. Dynamic Type | 389 |
| Using Preferred Fonts | 390 |
| Responding to User Changes | 392 |
| Updating Auto Layout | 393 |
| Content Hugging and Compression Resistance Priorities revisited | 393 |
| Determining the User's Preferred Text Size | 395 |
| Updating BNRIItemCell | 397 |
| Constraint outlets | 398 |
| Placeholder constraints | 399 |
| 21. Web Services and UIWebView | 403 |
| Web Services | 404 |

| | |
|--|-----|
| Starting the Nerdfeed application | 405 |
| NSURL, NSURLRequest, NSURLSession, and NSURLSessionTask | 406 |
| Formatting URLs and requests | 407 |
| Working with NSURLSession | 408 |
| JSON data | 409 |
| Parsing JSON data | 410 |
| The main thread | 412 |
| UIWebView | 413 |
| Credentials | 416 |
| Silver Challenge: More UIWebView | 417 |
| Gold Challenge: Upcoming Courses | 417 |
| For the More Curious: The Request Body | 417 |
| 22. UISplitViewController | 421 |
| Splitting Up Nerdfeed | 422 |
| Displaying the Master View Controller in Portrait Mode | 425 |
| Universalizing Nerdfeed | 428 |
| 23. Core Data | 431 |
| Object-Relational Mapping | 431 |
| Moving Homepwner to Core Data | 431 |
| The model file | 431 |
| NSManagedObject and subclasses | 437 |
| Updating BNRItemStore | 439 |
| Adding BNRAAssetTypes to Homepwner | 444 |
| More About SQL | 449 |
| Faults | 450 |
| Trade-offs of Persistence Mechanisms | 453 |
| Bronze Challenge: Assets on the iPad | 453 |
| Silver Challenge: New Asset Types | 453 |
| Gold Challenge: Showing Assets of a Type | 453 |
| 24. State Restoration | 455 |
| How State Restoration Works | 455 |
| Opting In to State Restoration | 456 |
| Restoration Identifiers and Classes | 457 |
| State Restoration Life Cycle | 459 |
| Restoring View Controllers | 461 |
| Encoding Relevant Data | 463 |
| Saving View States | 465 |
| Silver Challenge: Another Application | 467 |
| For the More Curious: Controlling Snapshots | 467 |
| 25. Localization | 469 |
| Internationalization Using NSNumberFormat | 470 |
| Localizing Resources | 473 |
| NSLocalizedString() and Strings Tables | 477 |
| Bronze Challenge: Another Localization | 480 |
| For the More Curious: NSBundle's Role in Internationalization | 480 |
| For the More Curious: Localizing XIB files without Base Internationalization | 481 |
| 26.NSUserDefaults | 483 |
| NSUserDefaults | 483 |

| | |
|---|-----|
| Register the factory settings | 484 |
| Read a preference | 485 |
| Change a preference | 485 |
| Settings Bundle | 487 |
| Editing the Root.plist | 488 |
| Localized Root.strings | 489 |
| 27. Controlling Animations | 491 |
| Basic Animations | 491 |
| Timing functions | 493 |
| Keyframe Animations | 494 |
| Animation Completion | 496 |
| Spring Animations | 497 |
| Silver Challenge: Improved Quiz | 498 |
| 28. UIStoryboard | 499 |
| Creating a Storyboard | 499 |
| UITableViewController s in Storyboards | 503 |
| Segues | 506 |
| Enabling Color Changes | 511 |
| Passing Data Around | 513 |
| More on Storyboards | 519 |
| For the More Curious: State Restoration | 520 |
| 29. Afterword | 523 |
| What to do Next | 523 |
| Shameless Plugs | 523 |
| Index | 525 |

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 HypnoNerd 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 **BNRReXViewController** 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.

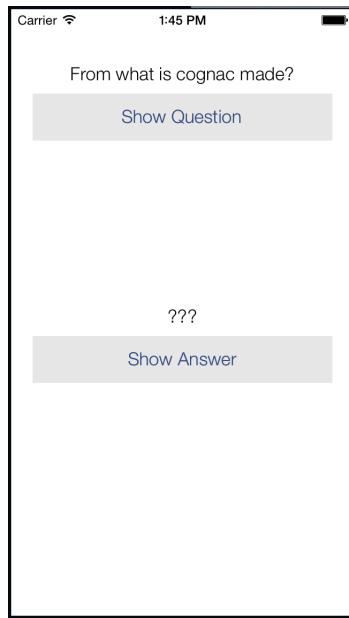
Excited yet? Good. Let's get started.

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.

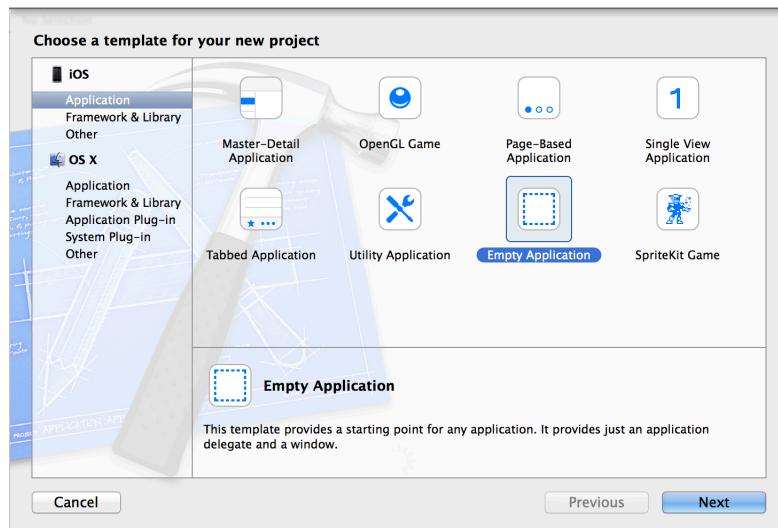
As you go through this first chapter, you will probably not understand everything that you are doing, and you may feel ridiculous just going through the motions. But going through the motions is enough for now. Mimicry is a powerful form of learning; it is how you learned to speak, and it is how you 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 Empty Application.

Figure 1.2 Creating a project



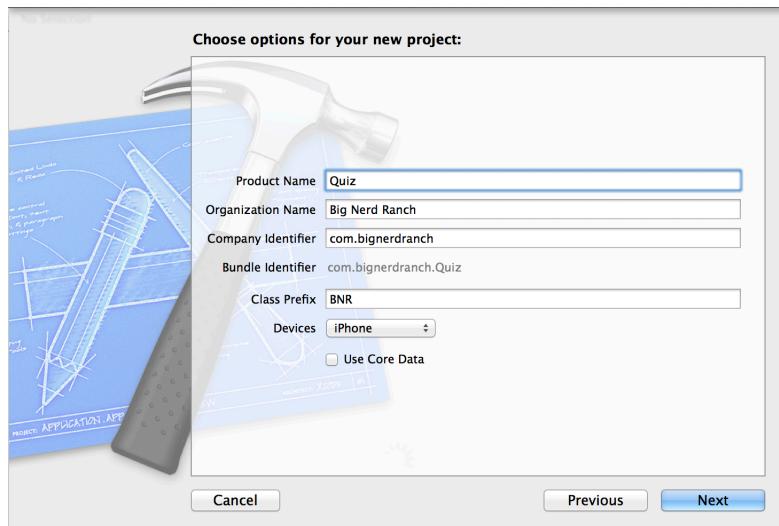
You are using the **Empty Application** template because it generates the least amount of boilerplate code. Too much boilerplate gets in the way of learning how things work.

This book was created for Xcode 5.0.2. The names of these templates may change with new Xcode releases. If you do not see an **Empty 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**.

In the Class Prefix field, enter **BNR** and, from the pop-up menu labeled **Devices**, choose **iPhone**. Make sure that the **Use Core Data** checkbox is unchecked.

Figure 1.3 Configuring a new project

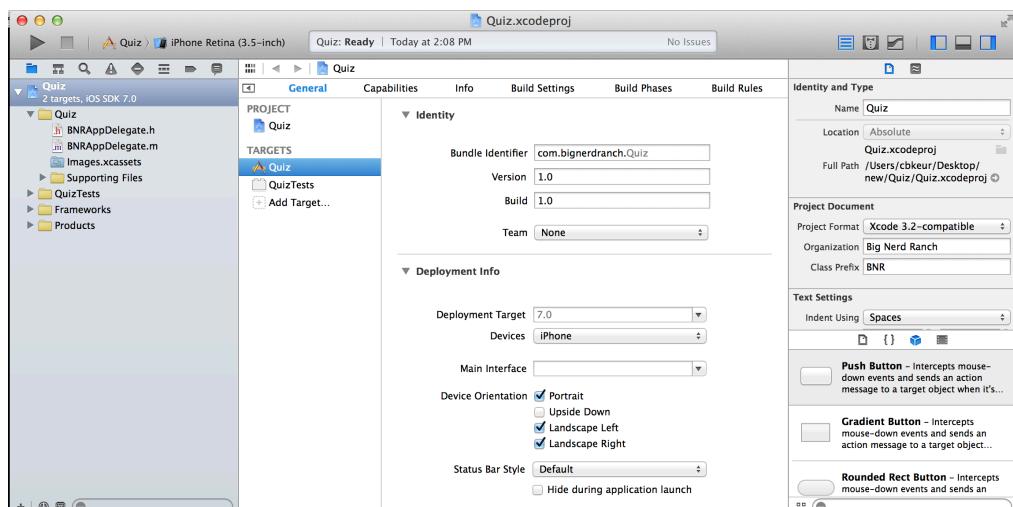


You are creating Quiz as an iPhone application, but it will run on an iPad. It will not look quite right on the iPad's screen, but that is okay for now. For the first part of this book, you will stick to iPhone applications and focus on learning the fundamentals of the iOS SDK, which are the same across 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. You can uncheck the box that creates a local git repository, but keeping it checked will not hurt anything. 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

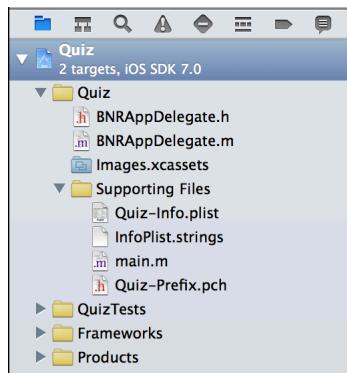


Take a look at the lefthand side of the workspace window. This area is called the *navigator area*, and it displays different *navigators* – tools that show you different parts of your project. You can choose which navigator to use by selecting one of the icons in the navigator selector, which is the 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 `BNRAppDelegate.h` and `BNRAppDelegate.m`. These are the files for a *class* named **BNRAppDelegate**. The Empty Application template created this class 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 **BNRAppDelegate** kind will be created. We refer to a **BNRAppDelegate** object as an *instance* of the **BNRAppDelegate** class.

You will learn much more about how classes and objects work in Chapter 2. 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.

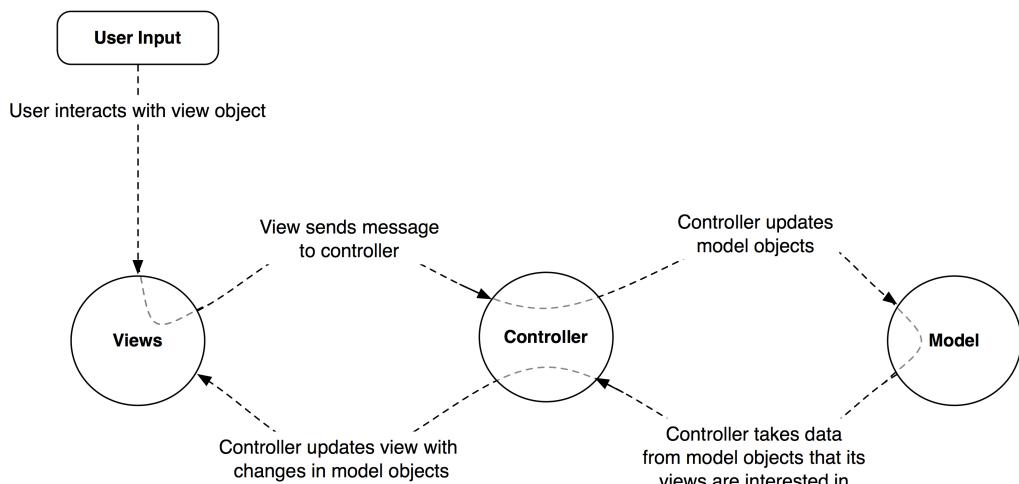
Usually, the model objects are modeling real things from the world of the user. For example, when you write an app for an insurance company, you will almost certainly end up with a custom model class called **InsurancePolicy**.

- *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.

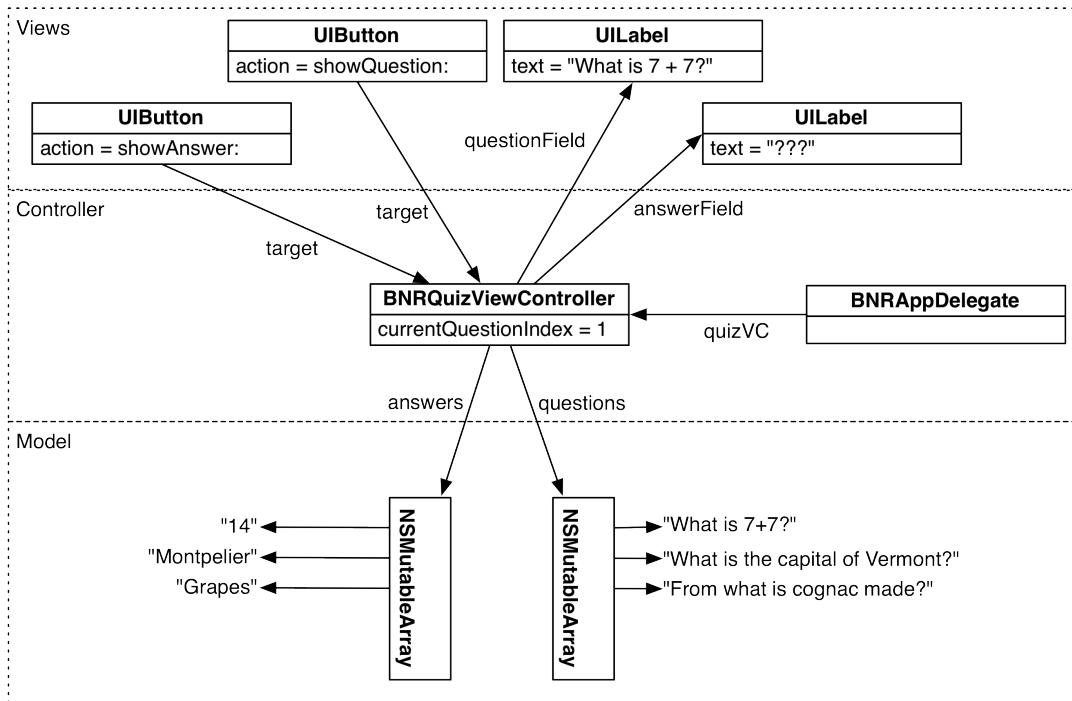
Designing Quiz

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**
- 2 controller objects: an instance of **BNRAppDelegate** and an instance of **BNRQuizViewController**
- 2 model objects: two instances of **NSMutableArray**

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 the **BNRQuizViewController**. 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.

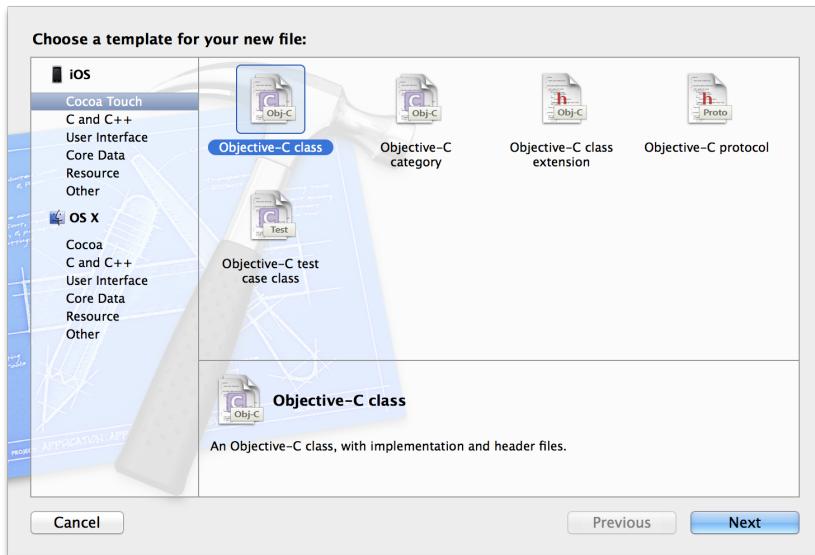
You are going to build Quiz in steps, starting with the controller object that sits in the middle of the app – **BNRQuizViewController**.

Creating a View Controller

The **BNRAppDelegate** class was created for you by the Empty Application template, but you will have to create the **BNRQuizViewController** class. We will talk more about classes in Chapter 2 and more about view controllers in Chapter 6. For now, just follow along.

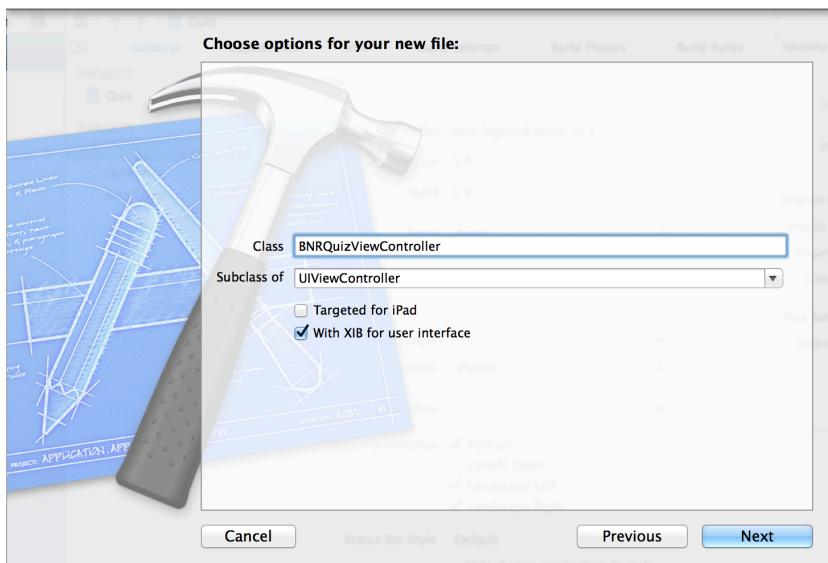
From the File menu, select New → File.... A sheet will slide down asking what type of file you would like to create. On the lefthand side under the iOS section, select Cocoa Touch. Then choose Objective-C Class and click Next.

Figure 1.8 Creating an Objective-C class



On the next sheet, name the class **BNRQuizViewController**. For the Subclass of field, click the drop-down menu arrow and select **UIViewController**. Select the With XIB for user interface checkbox.

Figure 1.9 Creating a view controller

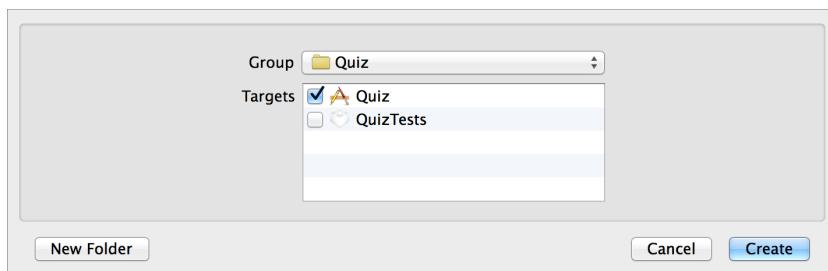


Click Next, and a panel will drop down that prompts you to create the files for this new class. When creating a new class for a project, you want to save the files that describe it inside the project's source

directory on the filesystem. By default, the current project directory is already selected for you. You can also choose the group in the project navigator that these files will be added to. Because these groups are simply for organizing and because this project is very small, just stick with the default.

Make sure the checkbox is selected for the Quiz target. This ensures that the `BNRQuizViewController` class will be compiled when the Quiz project is built. Click Create.

Figure 1.10 Quiz target is selected



Building an Interface

In the project navigator, find the class files for `BNRQuizViewController`. When you created this class, you checked the box for With XIB for user interface, so `BNRQuizViewController` came with a third class file: `BNRQuizViewController.xib`. Find and select `BNRQuizViewController.xib` in the project navigator to open it in the editor area.

When Xcode opens a XIB (pronounced “zib”) file, it opens it with Interface Builder, a visual tool where you can add and arrange objects to create a graphical user interface. In fact, XIB stands for XML Interface Builder.

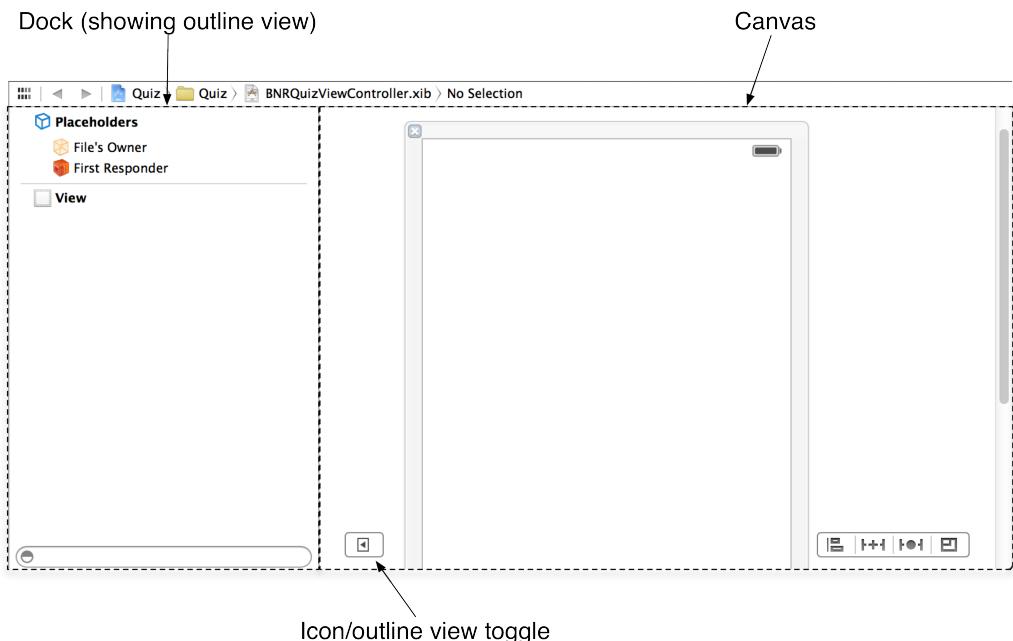
In many GUI builders on other platforms, you describe what you want an application to look like and then press a button to generate a bunch of code. Interface Builder is different. It is an object editor: you create and configure objects, like buttons and labels, and then save them into an archive. The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

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

The dock lists the objects in the XIB file either as icons (*icon view*) or in words (*outline view*). The icon view is useful when screen real estate is running low. However, for learning purposes, it is easier to see what is going on in the outline view.

If the dock is in icon view, click the disclosure button in the bottom lefthand corner of the canvas to reveal the outline view (Figure 1.11).

Figure 1.11 Editing a XIB file in Interface Builder



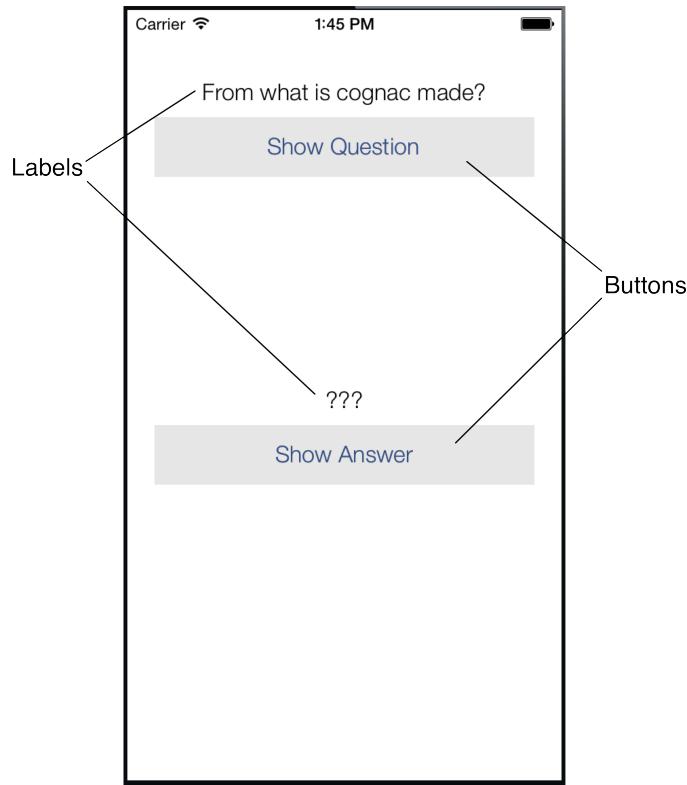
The outline view tells you that `BNRQuizViewController.xib` contains three objects: two placeholders and a View. Ignore the placeholders for now; you will learn about them later.

The View object is an instance of `UIImageView`. 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. You can also close the view by clicking the x in its top left corner. Again, this does not delete the view; it just removes it from the canvas. You can get it back by selecting it in the outline view.

Right now, your interface consists solely of this view object. You need to add four additional view objects for Quiz: two labels and two buttons.

Figure 1.12 Labels and buttons needed



Creating view objects

To add these view objects, you need to get to the object library 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 1.13). From the library selector, select the  tab to reveal the *object library*.

Figure 1.13 Xcode utility area



The object library contains the objects that you can add to a XIB file to compose your interface. Find the Label object. (It may be right at the top; if not, 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.

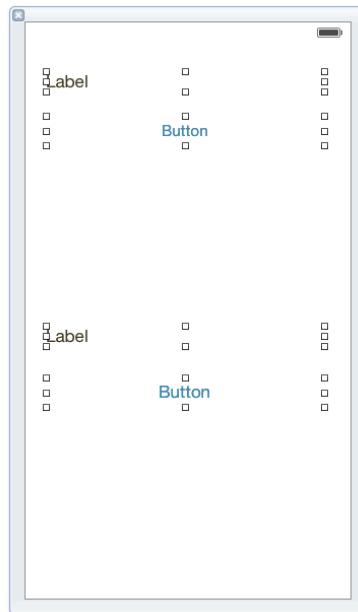
You have now created four view objects and added them to `BNRQuizViewController`'s user interface. Confirm this in the document outline.

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*, a tool that you will use shortly.

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 1.14 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 1.15.

Figure 1.15 Setting the text on the labels and buttons

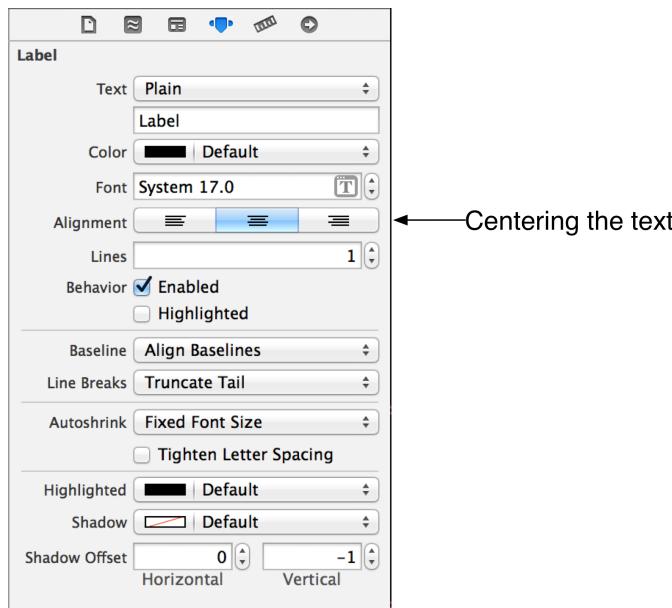


It would be nice if the text in the labels was centered. Setting the text alignment of a label must be done in the attributes inspector.

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 1.16.

Figure 1.16 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.

NIB files

At this point, you may be wondering how these objects are brought to life when the application is run.

When you build an application that uses a XIB file, the XIB file is compiled into a NIB file that is smaller and easier for the application to parse. Then the NIB file is copied into the application's

bundle. The bundle is a directory containing the application’s executable and any resources the executable uses.

At runtime, the application will read in, or load, the NIB file when its interface is needed. Quiz only has one XIB file and thus will have only one NIB file in its bundle. Quiz’s single NIB file will be loaded when the application first launches. A complex application, however, will have many NIB files that are loaded as they are needed. You will learn more about how NIB files are loaded in Chapter 6.

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 **BNRQuizViewController** 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 an object. (If you are not familiar with “pointers,” you will learn about them in Chapter 2.) 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 **BNRQuizViewController.m**. The editor area will change from Interface Builder to Xcode’s code editor.

In **BNRQuizViewController.m**, delete any code that the template added between the `@implementation` and `@end` directives so that the file looks like this:

```
#import "BNRQuizViewController.h"

@interface BNRQuizViewController : NSObject

@end

@implementation BNRQuizViewController
```

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

```
#import "BNRQuizViewController.h"

@interface BNRQuizViewController : NSObject

@property (nonatomic, weak) IBOutlet UILabel *questionLabel;
@property (nonatomic, weak) IBOutlet UILabel *answerLabel;

@end

@implementation
```

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 Chapter 3. For now, focus on the second half of the first line.

```
@property (nonatomic, weak) IBOutlet UILabel *questionLabel;
```

This code gives every instance of **BNRViewController** 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

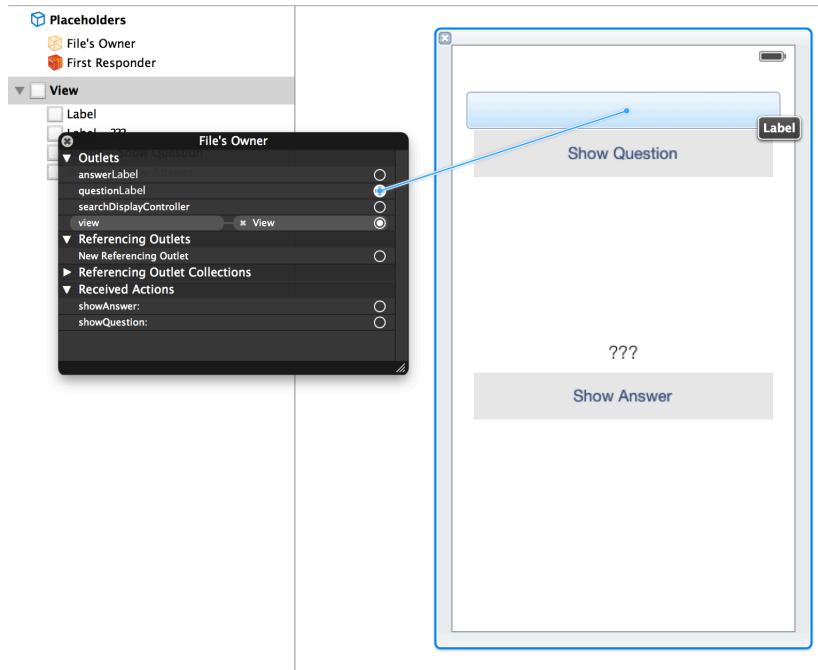
In the project navigator, select `BNRQuizViewController.xib` to reopen Interface Builder.

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

In the dock, find the Placeholders section and the File's Owner object. A placeholder stands in for another object in the running application. In your case, the File's Owner stands in for an instance of **BNRQuizViewController**, which is the object responsible for managing the interface defined in `BNRQuizViewController.xib`.

In the dock, right-click or Control-click on the File's Owner to bring up the connections panel (Figure 1.17). 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 1.17 Setting `questionLabel`

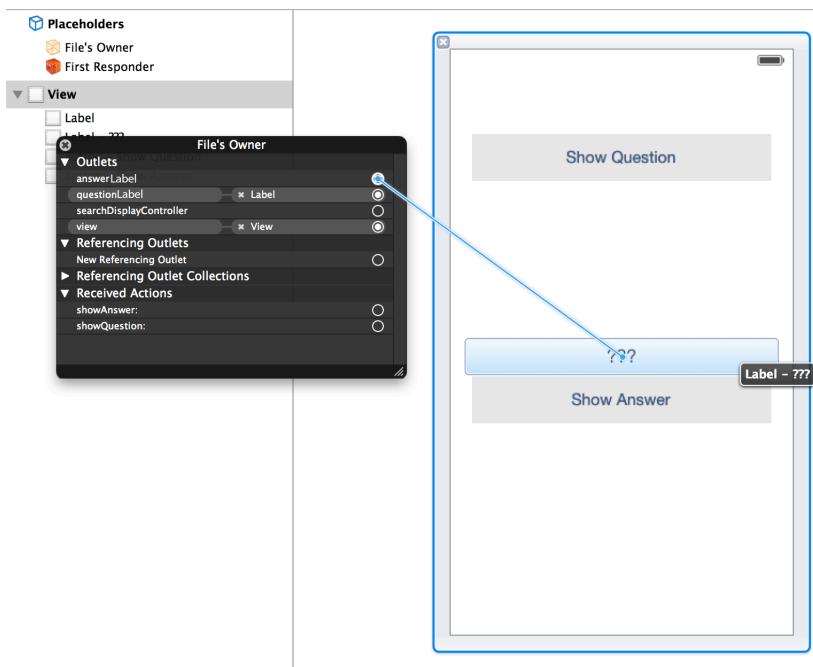


(If you do not see `questionLabel` in the connections panel, double-check your `BNRQuizViewController.m` file for typos.)

Now when the NIB file is loaded, the `BNRQuizViewController`'s `questionLabel` outlet will automatically point to the instance of `UILabel` at the top of the screen. This will allow the `BNRQuizViewController` 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 1.18).

Figure 1.18 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 `BNRQuizViewController`. Each button will have its own action. Let's start by defining the two action methods: `showQuestion:` and `showAnswer:`:

Defining action methods

Return to `BNRQuizViewController.m` and add the following code in between `@implementation` and `@end`.

```

@implementation

- (IBAction)showQuestion:(id)sender
{
}

- (IBAction)showAnswer:(id)sender
{
}

@end

```

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.

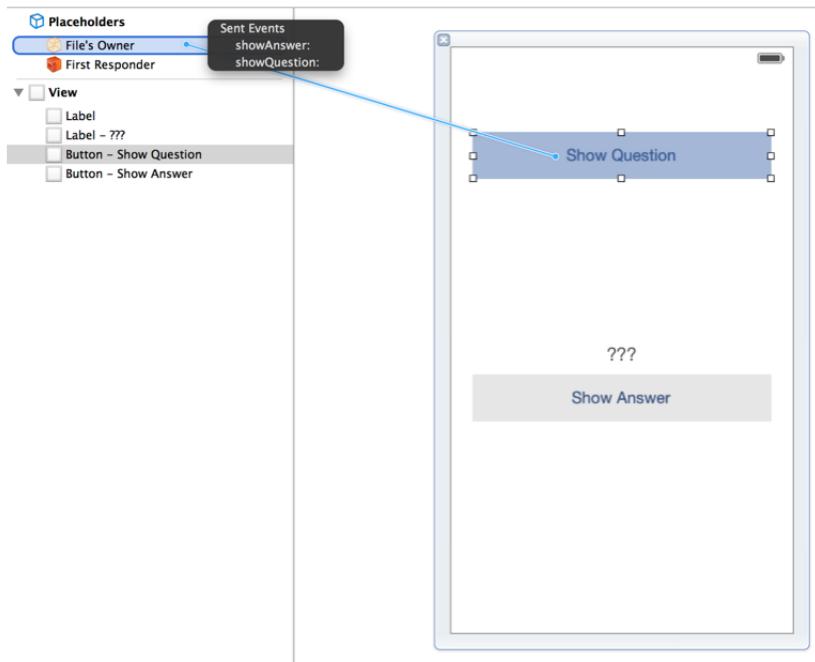
Setting targets and actions

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 `BNRQuizViewController` and its action to be `showQuestion:`.

Reopen `BNRQuizViewController.xib`. Select the Show Question button in the canvas and Control-drag (or right-click and drag) to the File's Owner. When the File's Owner is highlighted, release the mouse button and choose `showQuestion:` from the pop-up menu, as shown in Figure 1.19.

Figure 1.19 Setting Show Question target/action



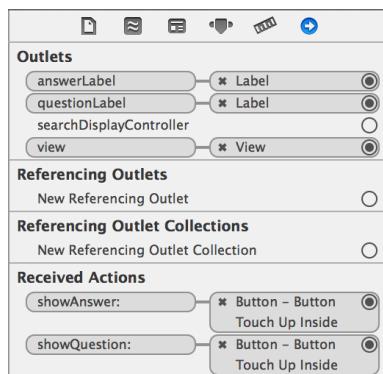
Now for the Show Answer button. Select the button and Control-drag from the button to the File's Owner. Then choose `showAnswer:` from the pop-up menu.

Summary of connections

There are now five connections between your `BNRQuizViewController` and the view objects. You have set the pointers `answerLabel` and `questionLabel` to point at the label objects – two connections. The `BNRQuizViewController` is the target for both buttons – two more. The project's template made one additional connection: the `view` pointer of `BNRQuizViewController` 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 File's Owner in the outline view. Then in the inspector, click the  tab to reveal the connections inspector. (Figure 1.20).

Figure 1.20 Checking connections in the inspector



Your XIB 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 `BNRQuizViewController.m`. Add the following code that declares an integer and pointers to two arrays.

```

@interface BNRQuizViewController ()
@property (nonatomic) int currentQuestionIndex;
@property (nonatomic, copy) NSArray *questions;
@property (nonatomic, copy) NSArray *answers;
@property (nonatomic, weak) IBOutlet UILabel *questionLabel;
@property (nonatomic, weak) IBOutlet UILabel *answerLabel;

@end

@implementation

@end

```

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

These arrays need to be ready to go at the same time that the interface appears to the user. To make sure that this happens, you are going to create the arrays right after an instance of **BNRQuizViewController** is created.

When an instance of **BNRQuizViewController** is created, it is sent the message **initWithNibName:bundle:**. In **BNRQuizViewController.m**, implement the **initWithNibName:bundle:** method.

```

...
@property (nonatomic, weak) IBOutlet UILabel *answerLabel;

@end

@implementation BNRQuizViewController

- (instancetype)initWithNibName:(NSString *)nibNameOrNil
                           bundle:(NSBundle *)nibBundleOrNil
{
    // Call the init method implemented by the superclass
    self = [super initWithNibName:nibNameOrNil bundle:nibBundleOrNil];

    if (self) {
        // Create two arrays filled with questions and answers
        // and make the pointers point to them

        self.questions = @[@"From what is cognac made?",
                           @"What is 7+7?",
                           @"What is the capital of Vermont?"];

        self.answers = @[@"Grapes",
                        @"14",
                        @"Montpelier"];
    }

    // Return the address of the new object
    return self;
}

@end

```

(Scary syntax? Feelings of dismay? Do not panic – you will learn more about the Objective-C language in the next two chapters.)

Using code-completion

As you work through this book, you will type a lot of code. Notice that as you were typing this code, Xcode was ready to fill in parts of it for you. For example, when you started typing `initWithNibName:bundle:`, it suggested this method before you could finish. You can press the Enter key to accept Xcode's suggestion or select another suggestion from the pop-up box that appears.

When you accept a code-completion suggestion for a method that takes arguments, Xcode puts *placeholders* in the areas for the arguments. (Note that this use of the term “placeholder” is completely distinct from the placeholder objects that you saw in the XIB file.)

Placeholders are not valid code, and you have to replace them to build the application. This can be confusing because placeholders often have the same names that you want your arguments to have. So the text of the code looks completely correct, but you get an error.

Figure 1.21 shows two placeholders you might have seen when typing in the previous code.

Figure 1.21 Code-completion placeholders and errors

Placeholders: select and press Return to
replace with arguments of the same names

The screenshot shows the Xcode interface with a project named "Quiz.xcodeproj" and a file "BNRQuizViewController.m". The code editor displays the implementation of the `BNRQuizViewController` class. On line 20, there is a placeholder for the `nibBundleOrNil` argument, represented by the text `nibNameOrNil` inside a rounded rectangle. On line 20, there is also a placeholder for the `nibBundleOrNil` argument, represented by the text `nibBundleOrNilOrNil` inside a rounded rectangle. The code is as follows:

```
16
17 @implementation BNRQuizViewController
18
19 - (id)initWithNibName:(NSString *)nibNameOrNilOrNil bundle:(NSBundle *)nibBundleOrNil
20 {
21     self = [super initWithNibName:NibNameOrNil bundle:nibBundleOrNil]
22
23     if (self) {
24         self.questions = @{@"From what is cognac made?","
25                           @"What is 7+7?","
26                           @"What is the capital of Vermont?"};
27
28         self.answers = @{@"Grapes",
29                          @"14",
30                          @"Montpelier"};
31     }
32
33     return self;
34 }
35
36 @end
```

See the `nibNameOrNilOrNil` and `nibBundleOrNil` in the first line of the implementation of `initWithNibName:bundle:`? Those are placeholders. You can tell because they are inside slightly-shaded, rounded rectangles. The fix is to select the placeholders, type the correct argument name, and press the Enter key. The rounded rectangles will go away, and your code will be correct and valid.

Because the placeholders have the correct text in this case, you can simply select a placeholder and press Enter to have Xcode replace it with an argument of the same name.

When using code-completion, watch out for names that look *almost* like what you want. Cocoa Touch uses naming conventions which often cause distinct methods, types, and variables to have very similar names. Thus, do not blindly accept the first suggestion Xcode gives you without verifying it. Always double-check.

Pulling it all Together

You have created, configured, and connected your view objects and their view controller. You have created the `model` objects. Two things are left to do for Quiz:

- finish implementing the action methods `showQuestion:` and `showAnswer:` in `BNRQuizViewController`
- add code to `BNRAppDelegate` that will create and display an instance of `BNRQuizViewController`

Implementing action methods

In `BNRQuizViewController.m`, finish the implementations of `showQuestion:` and `showAnswer:`.

```
...
// Return the address of the new object
return self;
}

- (IBAction)showQuestion:(id)sender
{
    // Step to the next question
    self.currentQuestionIndex++;

    // Am I past the last question?
    if (self.currentQuestionIndex == [self.questions count]) {

        // Go back to the first question
        self.currentQuestionIndex = 0;
    }

    // Get the string at that index in the questions array
    NSString *question = self.questions[self.currentQuestionIndex];

    // Display the string in the question label
    self.questionLabel.text = question;

    // Reset the answer label
    self.answerLabel.text = @"????";
}

- (IBAction)showAnswer:(id)sender
{
    // What is the answer to the current question?
    NSString *answer = self.answers[self.currentQuestionIndex];

    // Display it in the answer label
    self.answerLabel.text = answer;
}

@end
```

Getting the view controller on the screen

If you were to run Quiz right now, you would not see the interface that you created in `BNRQuizViewController.xib`. Instead, you would see a blank white screen. To get your interface on screen, you have to connect your view controller with the application's other controller – **BNRAppDelegate**.

Whenever you create an iOS application using an Xcode template, an *app delegate* is created for you. **The app delegate is the starting point of an application, and every iOS application has one.**

The app delegate manages a single top-level **UIWindow** for the application. To get the **BNRQuizViewController** on screen, you need to make it the *root view controller* of this window.

When an iOS application is launched, it is not immediately ready for the user. There is some setup that goes on behind the scenes. Right before the app is ready for the user, the app delegate receives the message **application:didFinishLaunchingWithOptions:**. This is your chance to prepare the application for action. In particular, you want to make sure that your interface is ready before the user has a chance to interact with it.

In the project navigator, find and select `BNRAppDelegate.m`. Add the following code to the **application:didFinishLaunchingWithOptions:** method to create an instance of **BNRQuizViewController** and to set it as the root view controller of the app delegate's window.

```
#import "BNRAppDelegate.h"
#import "BNRQuizViewController.h"

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch

    BNRQuizViewController *quizVC = [[BNRQuizViewController alloc] init];
    self.window.rootViewController = quizVC;

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

Now whenever the app is launched, an instance of **BNRQuizViewController** will be created. This instance will then receive the **initWithNibName:bundle:** message, which will trigger loading the NIB file compiled from `BNRQuizViewController.xib` and the creation of the model objects.

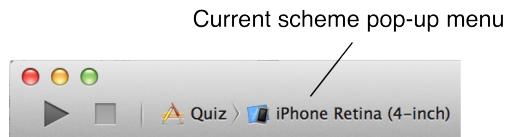
Your Quiz is complete. Time to try it out.

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 1.22).

Figure 1.22 iPhone Retina (4-inch) scheme selected



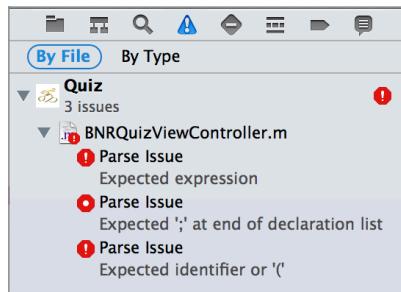
If it says something generic like iPhone Retina (4-inch), then the project is set to run on the simulator and you are good to go. If it says something like Christian's iPhone, then click it and choose iPhone Retina (4-inch) from the pop-up menu.

For this book, use the iPhone Retina (4-inch) scheme. The only difference between that and iPhone Retina (3.5-inch) is the height of the screen. If you select the 3.5-inch simulator, 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 15.

Next, click the iTunes-esque 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 1.23).

Figure 1.23 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 `BNRQuizViewController.xib`.

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:

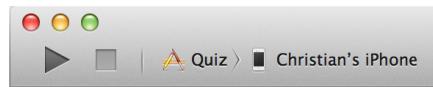
| | |
|-----------------------|---|
| Developer Certificate | This certificate file is added to your Mac's keychain using Keychain Access . It is used to digitally sign your code. |
| App ID | The application identifier is a string that uniquely identifies your application on the App Store. Application identifiers typically look like this: <code>com.bignerdranchAwesomeApp</code> , 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 <code>.mobileprovision</code> . |

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 1.24). If iOS Device is not an option, find the choice that reads something like Christian's iPhone.

Figure 1.24 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 / iPod touch (iOS 7) | 120x120 pixels (@2x) |
| iPhone / iPod touch (iOS 6 and earlier) | 57x57 pixels 114x114 pixels (@2x) |
| iPad (iOS 7 and earlier) | 72x72 pixels 144x144 pixels (@2x) |

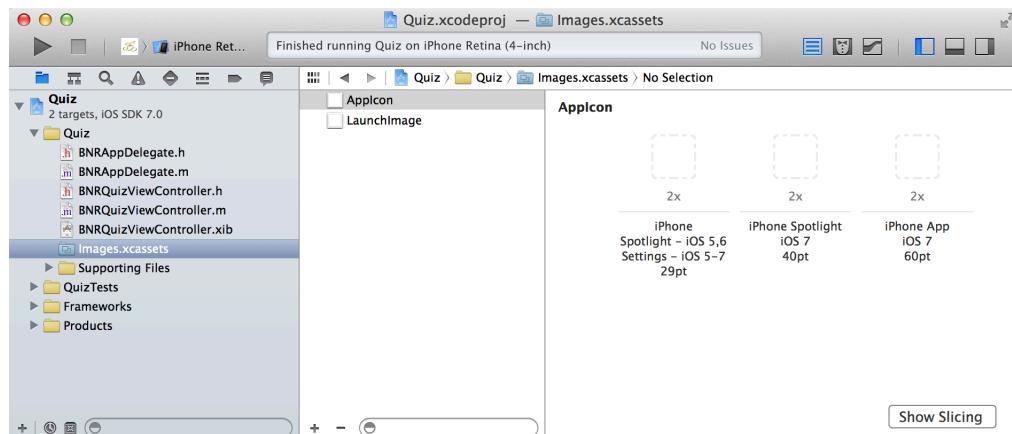
Every application you deploy to the App Store is required to have icons for every device class on which it can run. For example, if you are supporting just iPhone and iPod touch running iOS 7 or later, then you just need to supply one image (at the resolution listed above). On the other extreme, if you have a universal application which supports iOS 6 and above, you will need to have five different resolution app icons, the two necessary for iPad and the three others necessary for iPhone and iPod touch.

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 `BNRQuizViewController.h` and `BNRQuizViewController.m`) is used to create the application itself. Resources are things like images and sounds that are used by the application at runtime. XIB files, which are compiled into NIB files that are read in at runtime, 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.

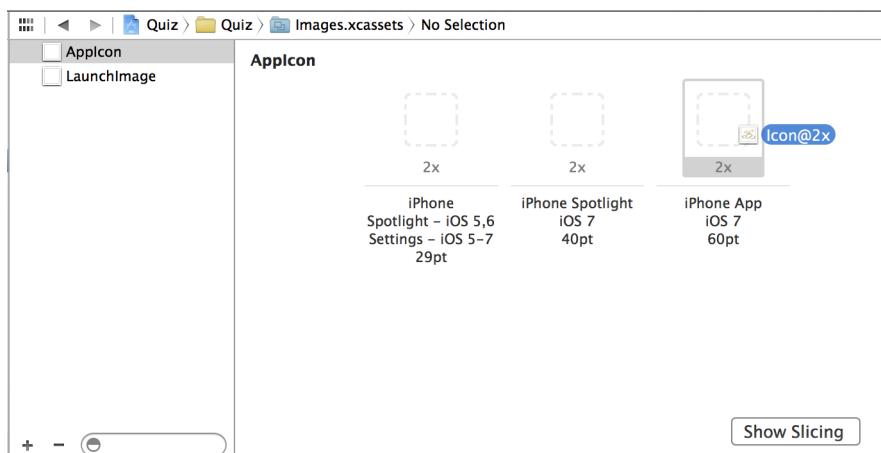
Figure 1.25 Showing the Asset Catalog



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 1.26 Adding the App Icon to the Asset Catalog



Build and run the application again. Exit the application and look for the Quiz application with the BNR logo.

(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 resource that you can manage from the Asset Catalog is the application 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.)

In the Resources directory where you found the app icons, there are two launch images: Default@2x.png and Default-568h@2x.png. Open the LaunchImage item in the Asset Catalog, and then drag these images in to the Asset Catalog like you did with the app icons.

Build and run the application. As the application launches, you will briefly see the launch image.

Why two launch images? A launch image must fit the screen of the device it is being launched on. Therefore you need one launch image for 3.5 inch Retina displays, and another launch image for 4 inch Retina displays. Note that if you are supporting iOS6 or earlier on iPhone and iPod touch, you will need to include a third non-Retina launch image. Table 1.2 shows the different size images you will need for each type of device.

Table 1.2 Launch image sizes by device

| Device | Launch image size |
|--|---|
| iPhone/iPod touch without Retina display | 320x480 pixels (Portrait only) |
| iPhone/iPod touch with Retina display (3.5 inch) | 640x960 pixels (Portrait only) |
| iPhone/iPod touch with Retina display (4 inch) | 640x1136 pixels (Portrait only) |
| iPad without Retina display | 768x1024 pixels (Portrait) 1024x768 pixels (Landscape) |
| iPad with Retina display | 1536x2048 pixels (Portrait) 2048x1536 pixels (Landscape) |

(Note that Table 1.2 lists the screen resolutions of the devices; the real status bar is overlaid on top of the launch image.)

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.

This page intentionally left blank

2

Objective-C

iOS applications are written in the Objective-C language using the Cocoa Touch frameworks. Objective-C is an extension of the C language, and the Cocoa Touch frameworks are collections of Objective-C classes.

In this chapter, you will learn the basics of Objective-C and create an application called `RandomItems`. Even if you are familiar with Objective-C, you should still go through this chapter to create the `BNRItem` class that you will use later in the book.

This book assumes you know some C and understand the basic ideas of object-oriented programming. If C or object-oriented programming makes you uneasy, we recommend starting with *Objective-C Programming: The Big Nerd Ranch Guide*.

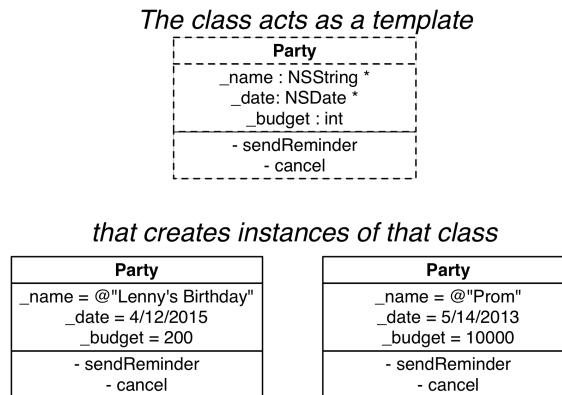
Objects

Let's say you need a way to represent a party. Your party has a few attributes that are unique to it, like a name, a date, and a list of invitees. You can also ask the party to do things, like send an email reminder to all the invitees, print name tags, or cancel the party altogether.

In C, you would define a structure to hold the data that describes a party. The structure would have data members – one for each of the party's attributes. Each data member would have a name and a type. To create an individual party, you would use the function `malloc` to allocate a chunk of memory large enough to hold the structure.

In Objective-C, instead of using a structure to represent a party, you use a *class*. A class is like a cookie-cutter that produces objects. The `Party` class creates objects, and these objects are instances of the `Party` class. Each instance of the `Party` class can hold the data for a single party (Figure 2.1).

Figure 2.1 A class and its instances



An instance of **Party**, like any object, is a chunk of data in memory, and it stores the values for its attributes in *instance variables*. (You also may see these referred to as “ivars” in some places.) In Objective-C, we typically put an underscore at the beginning of the instance variable name. So an instance of **Party** might have the instance variables `_name`, `_date`, and `_budget`.

A C structure is a chunk of memory, and an object is a chunk of memory. A C structure has data members, each with a name and a type. Similarly, an object has instance variables, each with a name and a type.

The important difference between a C structure and an Objective-C class is that a class has *methods*. A method is similar to a function: it has a name, a return type, and a list of parameters that it expects. A method also has access to an object’s instance variables. If you want an object to run the code in one of its methods, you send that object a *message*.

Using Instances

To use an instance of a class, you must have a variable that points to that object. A pointer variable stores the location of an object in memory, not the object itself. (It “points to” the object.) A variable that points to a **Party** object is declared like this:

```
Party *partyInstance;
```

Creating this pointer does not create a **Party** object – only a variable that can point to a **Party** object.

This variable is named `partyInstance`. Notice that this variable’s name does not start with an underscore; it is not an instance variable. It is meant to be a pointer to an instance of **Party**.

Creating objects

An object has a life span: it is created, sent messages, and then destroyed when it is no longer needed.

To create an object, you send an `alloc` message to a class. In response, the class creates an object in memory (on the heap, just like `malloc()` would) and gives you the address of the object, which you store in a variable:

```
Party *partyInstance = [Party alloc];
```

You create a pointer to an instance so that you can send messages to it. The first message you *always* send to a newly allocated instance is an initialization message. Although sending an **alloc** message to a class creates an instance, the instance is not ready for work until it has been initialized.

```
Party *partyInstance = [Party alloc];
[partyInstance init];
```

Because an object must be allocated *and* initialized before it can be used, you always combine these two messages in one line.

```
Party *partyInstance = [[Party alloc] init];
```

Combining two messages in a single line of code is called a *nested message send*. The innermost brackets are evaluated first, so the message **alloc** is sent to the class **Party** first. This returns a pointer to a new, uninitialized instance of **Party** that is then sent the message **init**. This returns a pointer to the initialized instance that you store in your pointer variable.

Sending messages

What do you do with an instance that has been initialized? You send it more messages.

First, let's take a closer look at message anatomy. A message is always contained in square brackets. Within a pair of square brackets, a message has three parts:

receiver a pointer to the object being asked to execute a method

selector the name of the method to be executed

arguments the values to be supplied as the parameters to the method

For example, a party might have a list of attendees that you can add to by sending the party the message **addAttendee:**.

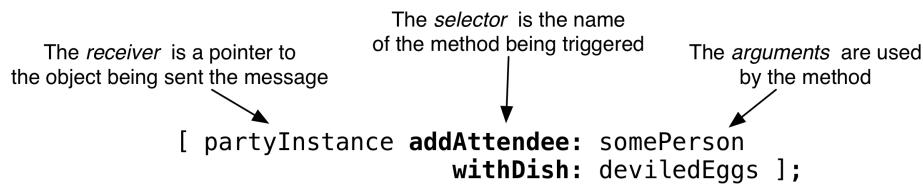
```
[partyInstance addAttendee:somePerson];
```

Sending the **addAttendee:** message to **partyInstance** (the receiver) triggers the **addAttendee:** method (named by the selector) and passes in **somePerson** (an argument).

The **addAttendee:** message has one argument. Objective-C methods can take a number of arguments or none at all. The message **init**, for instance, has no arguments.

An attendee to a party might need to RSVP and inform the host what dish the attendee will bring. Thus, the **Party** class may have another method named **addAttendee:withDisk:**. This message takes two arguments: the attendee and the dish. Each argument is paired with a label in the selector, and each label ends with a colon. The selector is all of the labels taken together (Figure 2.2).

Figure 2.2 Parts of a message send



The pairing of labels and arguments is an important feature of Objective-C. In other languages, this method might look something like this:

```
partyInstance.addAttendeeWithDish(somePerson, deviledEggs);
```

In those languages, it is not completely obvious what each of the arguments sent to this function are. In Objective-C, however, each argument is paired with the appropriate label.

```
[partyInstance addAttendee:somePerson withDish:deviledEggs];
```

It takes some getting used to, but eventually Objective-C programmers appreciate the clarity of arguments being interposed into the selector. The trick is to remember that for every pair of square brackets, there is only one message being sent. Even though **addAttendee:withDish:** has two labels, it is still only one message, and sending that message results in only one method being executed.

In Objective-C, the name of a method is what makes it unique. Therefore, a class cannot have two methods with the same name. Two method names can contain the same individual label, as long as the name of each method differs as a whole. For example, our **Party** class has two methods, **addAttendee:** and **addAttendee:withDish:**. These are two distinct methods, and they do not share any code.

Also, notice the distinction being made between a *message* and a *method*: a method is a chunk of code that can be executed, and a message is the act of asking a class or object to execute a method. The name of a message always matches the name of the method to be executed.

Destroying objects

To destroy an object, you set the variable that points to it to **nil**.

```
partyInstance = nil;
```

This line of code destroys the object pointed to by the **partyInstance** variable and sets the value of the **partyInstance** variable to **nil**. (It is actually a bit more complicated than that, and you will learn about memory management in the next chapter.)

The value **nil** is the zero pointer. (C programmers know it as **NULL**. Java programmers know it as **null**.) A pointer that has a value of **nil** is typically used to represent the absence of an object. For example, a party could have a venue. While the organizer of the party is still determining where to host the party, **venue** would point to **nil**. This allows you to do things like:

```
if (venue == nil) {
    [organizer remindToFindVenueForParty];
}
```

Objective-C programmers typically use the shorthand form of determining if a pointer is **nil**:

```
if (!venue) {
    [organizer remindToFindVenueForParty];
}
```

Since the **!** operator means “not,” this reads as “if there is not a venue” and will evaluate to true if **venue** is **nil**.

If you send a message to a variable that is **nil**, nothing happens. In other languages, sending a message to the zero pointer is illegal, so you see this sort of thing a lot:

```
// Is venue non-nil?
if (venue) {
    [venue sendConfirmation];
}
```

In Objective-C, this check is unnecessary because a message sent to `nil` is ignored. Therefore, you can simply send a message without a `nil`-check:

```
[venue sendConfirmation];
```

If the venue has not yet been chosen, you will not send a confirmation anywhere. (A corollary: if your program is not doing anything when you think it should be doing something, an unexpected `nil` pointer is often the culprit.)

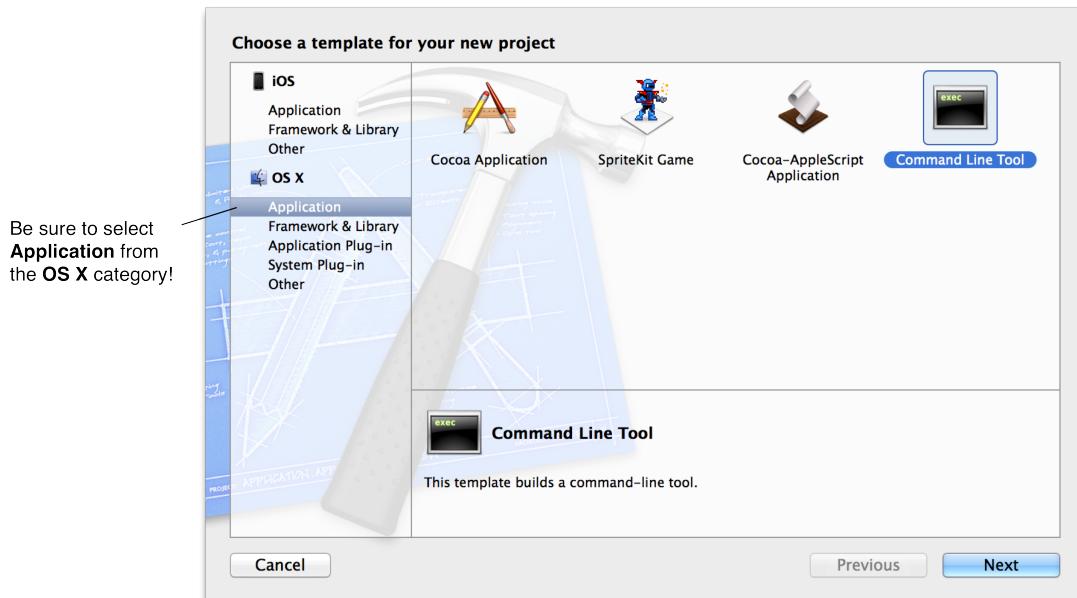
Enough theory. Time for some practice and a new project.

Beginning RandomItems

This new project is not an iOS application; it is a command-line program. Building a command line program will let you focus on Objective-C without the distraction of a user interface. Still, the concepts that you will learn here and in Chapter 3 are critical for iOS development. You will get back to building iOS applications in Chapter 4.

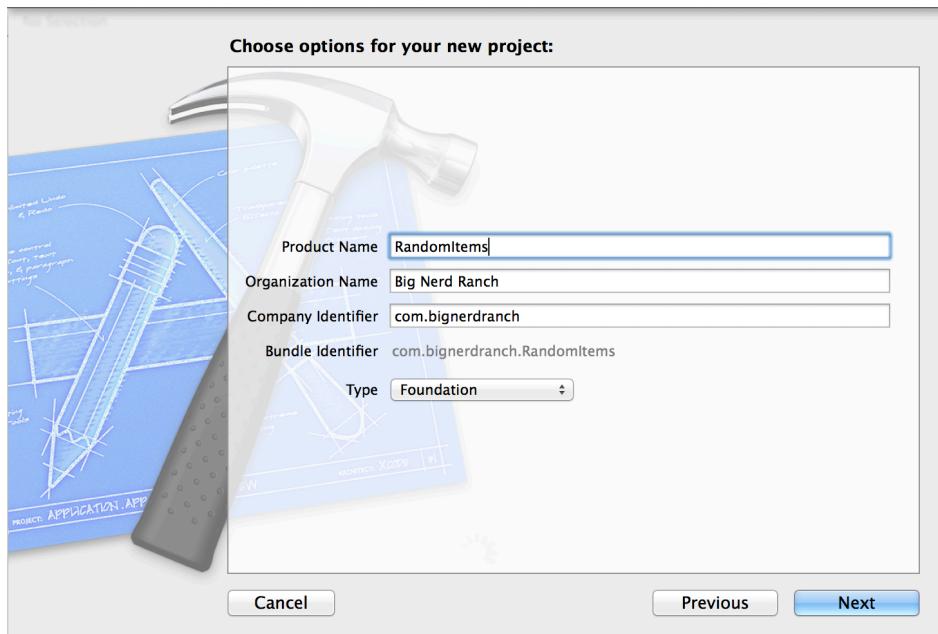
Open Xcode and select `File → New → Project...`. In the lefthand table in the OS X section, click Application and then select Command Line Tool from the upper panel, as shown in Figure 2.3. Then click Next.

Figure 2.3 Creating a command line tool



On the next sheet, name the product RandomItems and choose Foundation as its type (Figure 2.4).

Figure 2.4 Naming the project



Click Next and you will be prompted to save the project. Save it some place safe – you will be reusing parts of this code in future projects.

In the initial version of RandomItems, you will create an *array* of four strings. An array is an ordered list of pointers to other objects. Pointers in an array are accessed by an index. (Other languages might call it a list or a vector.) Arrays are zero-based; the first item in the array is always at index 0.

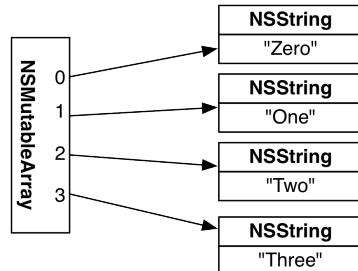
After the array is created, you will loop through the array and print each string. Its output will appear in Xcode's console:

Figure 2.5 Console output

```
2013-12-08 18:37:03.481 RandomItems[4208:303] Zero
2013-12-08 18:37:03.483 RandomItems[4208:303] One
2013-12-08 18:37:03.484 RandomItems[4208:303] Two
2013-12-08 18:37:03.484 RandomItems[4208:303] Three
Program ended with exit code: 0
```

Your program will need five objects: one instance of **NSMutableArray** and four instances of **NSString** (Figure 2.6).

Figure 2.6 **NSMutableArray** instance containing pointers to **NSString** instances

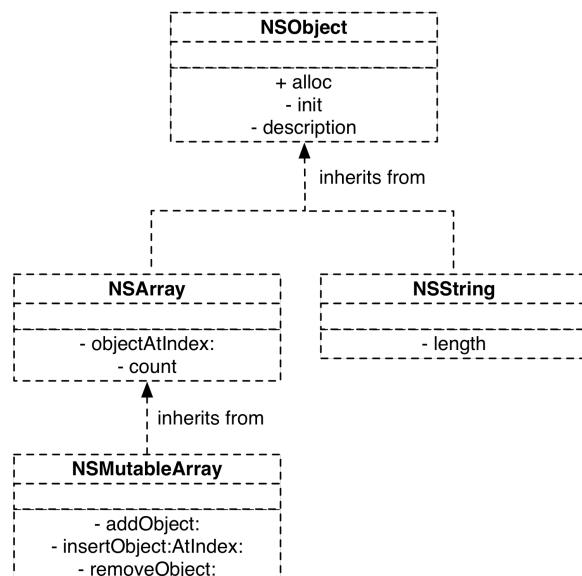


In Objective-C, an array does not contain the objects that belong to it; instead it holds a pointer to each object. When an object is added to an array, the address of that object in memory is stored inside the array.

Now let's consider the **NSMutableArray** and **NSString** classes. **NSMutableArray** is a *subclass* of **NSArray**. Classes exist in a hierarchy, and every class has exactly one superclass – except for the root class of the entire hierarchy: **NSObject**. A class inherits the behavior of its superclass.

Figure 2.7 shows the class hierarchy for **NSMutableArray** and **NSString** along with a few methods implemented by each class.

Figure 2.7 Class hierarchy



As the top superclass, **NSObject**'s role is to implement the basic behavior of every object in Cocoa Touch. Every class inherits the methods and instance variables defined in **NSObject**. Two methods that

NSObject implements `alloc` and `init` (Figure 2.7). Thus, these methods can be used to create an instance of any class.

A subclass adds methods and instance variables to extend the behavior of its superclass:

- **NSString** adds behavior for storing and handling strings, including the method `length` that returns the number of characters in a string.
- **NSArray** adds behavior for ordered lists, including accessing an object at a given index (`objectAtIndex:`) and getting the number of objects in an array (`count`).
- **NSMutableArray** extends **NSArray**'s abilities by adding the abilities to add and remove pointers.

Creating and populating an array

Let's use these classes in some real code. In the project navigator, find and select the file named `main.m`. When it opens, you will see that some code has been written for you – most notably, a `main` function that is the entry point of any C or Objective-C application.

In `main.m`, delete the line of code that `NSLogs` "Hello, World!" and replace it with code that creates an instance of **NSMutableArray**, adds a total of four objects to the mutable array, and then destroys the array.

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

int main (int argc, const char * argv[])
{
    @autoreleasepool {

        // insert code here...
        // NSLog(@"%@", @"Hello, World!");

        // Create a mutable array object, store its address in items variable
        NSMutableArray *items = [[NSMutableArray alloc] init];

        // Send the message addObject: to the NSMutableArray pointed to
        // by the variable items, passing a string each time
        [items addObject:@"One"];
        [items addObject:@"Two"];
        [items addObject:@"Three"];

        // Send another message, insertObjectAtIndex:, to that same array object
        [items insertObject:@"Zero" atIndex:0];

        // Destroy the mutable array object
        items = nil;
    }
    return 0;
}
```

The objects that you have added and inserted into the array are instances of **NSString**. You can create an instance of **NSString** by prefixing a character string with an @ symbol:

```
NSString *myString = @"Hello, World!";
```

Iterating over an array

Now that you have an `items` array with strings in it, you are going to iterate over the array, access each string, and output it to the console.

You could write this operation as a `for` loop:

```
for (int i = 0; i < [items count]; i++) {
    NSString *item = [items objectAtIndex:i];
    NSLog(@"%@", item);
}
```

Because arrays are zero-based, the counter starts at `0` and stops at one less than the result of sending the `items` array the message `count`. Within the loop, you then send the message `objectAtIndex:` to retrieve the object at the current index before printing it out.

The information returned from `count` is important because if you ask an array for an object at an index that is equal to or greater than the number of objects in the array, an exception will be thrown. (In some languages, exceptions are thrown and caught all the time. In Objective-C, exceptions are considered programmer errors and should be fixed in code instead of handled at runtime. We will talk more about exceptions at the end of this chapter.)

This code would work fine, but Objective-C provides a better option for iterating over an array called *fast enumeration*. Fast enumeration is shorter syntax than a traditional `for` loop and far less error-prone. In some cases, it will be faster.

In `main.m`, add the following code that uses fast enumeration to iterate over the `items` array.

```
int main (int argc, const char * argv[])
{
    @autoreleasepool {

        // Create a mutable array object, store its address in items variable
        NSMutableArray *items = [[NSMutableArray alloc] init];

        // Send the message addObject: to the NSMutableArray pointed to
        // by the variable items, passing a string each time
        [items addObject:@"One"];
        [items addObject:@"Two"];
        [items addObject:@"Three"];

        // Send another message, insertObjectAtIndex:, to that same array object
        [items insertObject:@"Zero" atIndex:0];

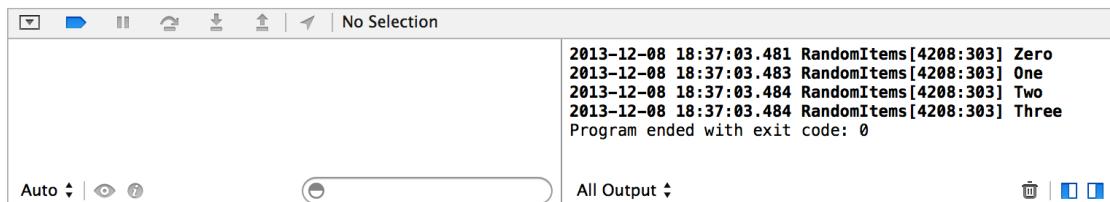
        // For every item in the items array ...
        for (NSString *item in items) {
            // Log the description of item
            NSLog(@"%@", item);
        }

        items = nil;
    }
    return 0;
}
```

Fast enumeration has a limitation: `you cannot use it if you need to add or remove objects within the loop`. Trying to do so will cause an exception to be thrown. If you need to add or remove objects, you must use a regular `for` loop with a counter variable.

Build and run the application (Command-R). A new pane will appear at the bottom of the Xcode window. This is the *debug area*. On the righthand side of this area is the *console* and your output.

Figure 2.8 Output in console



If you need to, you can resize the debug area and its panels by dragging their frames. (In fact, you can resize any area in the workspace window this way.)

You have completed the first phase of the `RandomItems` program. Before you continue with the next phase, let's take a closer look at the `NSLog` function and format strings.

Format strings

`NSLog()` takes a variable number of arguments and prints a string to the log. In Xcode, the log appears in the console. The first argument of `NSLog()` is required. It is an instance of `NSString` and is called the *format string*.

A format string contains text and a number of *tokens*. Each token (also called a format specification) is prefixed with a percent symbol (%). Each additional argument passed to the function replaces a token in the format string.

Tokens specify the type of the argument that they correspond to. Here is an example:

```
int a = 1;
float b = 2.5;
char c = 'A';
 NSLog(@"%@", @"Integer: %d Float: %f Char: %c", a, b, c);
```

The output would be

```
Integer: 1 Float: 2.5 Char: A
```

Objective-C format strings work the same way as in C. However, Objective-C adds one more token: %@. This token corresponds to an argument whose type is “a pointer to any object.”

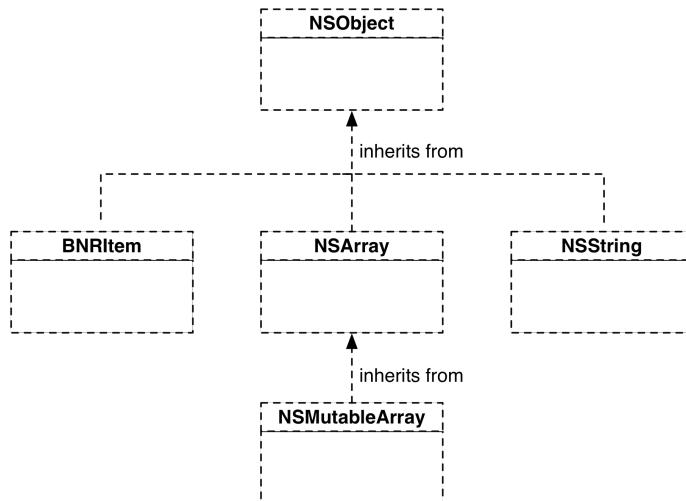
When %@ is encountered in the format string, instead of the token being replaced by the corresponding argument, that argument is sent the message `description`. The `description` method returns an instance of `NSString` that replaces the token.

Because the argument that corresponds to the %@ token is sent a message, that argument must be an object. Back in Figure 2.7, you can see that `description` is a method on the `NSObject` class. Thus, every class implements `description`, which is why you can use the %@ token with any object.

Subclassing an Objective-C Class

In this section, you are going to create a new class named `BNRItem`. `BNRItem` will be a subclass of `NSObject`.

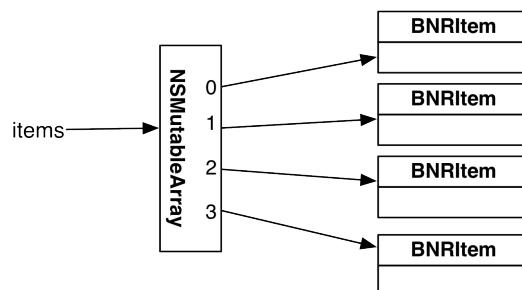
Figure 2.9 Class hierarchy including **BNRItem**



An instance of **BNRItem** will represent something that a person owns in the real world, like a laptop, a bicycle, or a backpack. In terms of Model-View-Controller, **BNRItem** is a model class. An instance of **BNRItem** stores information about a possession.

Once you have created the **BNRItem** class, you will populate the `items` array with instances of **BNRItem** instead of **NSString**.

Figure 2.10 A different class of items

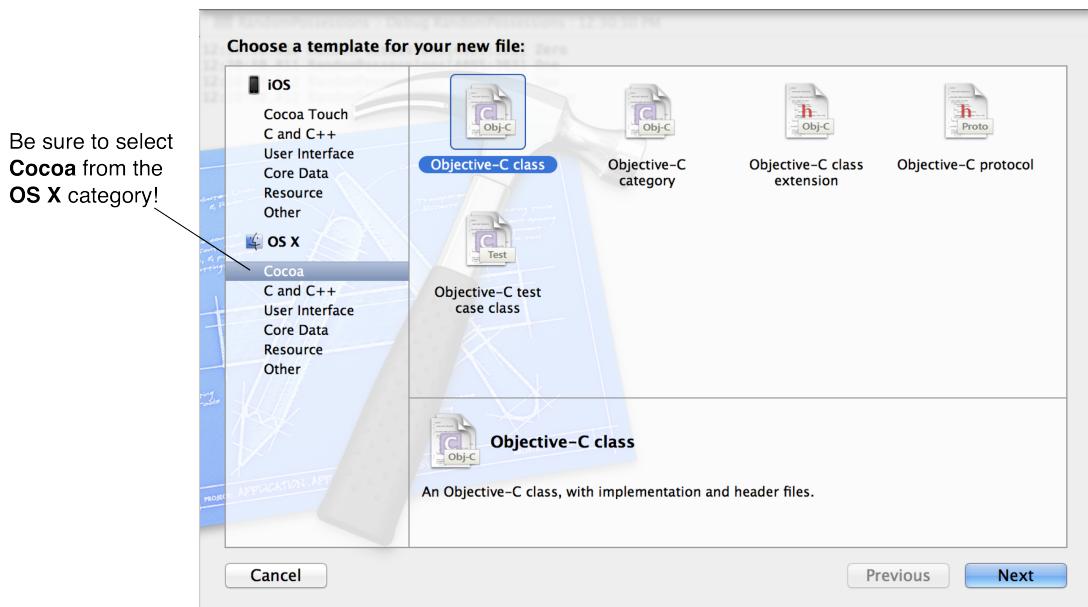


Later in the book, you will reuse **BNRItem** in a complex iOS application.

Creating an NSObject subclass

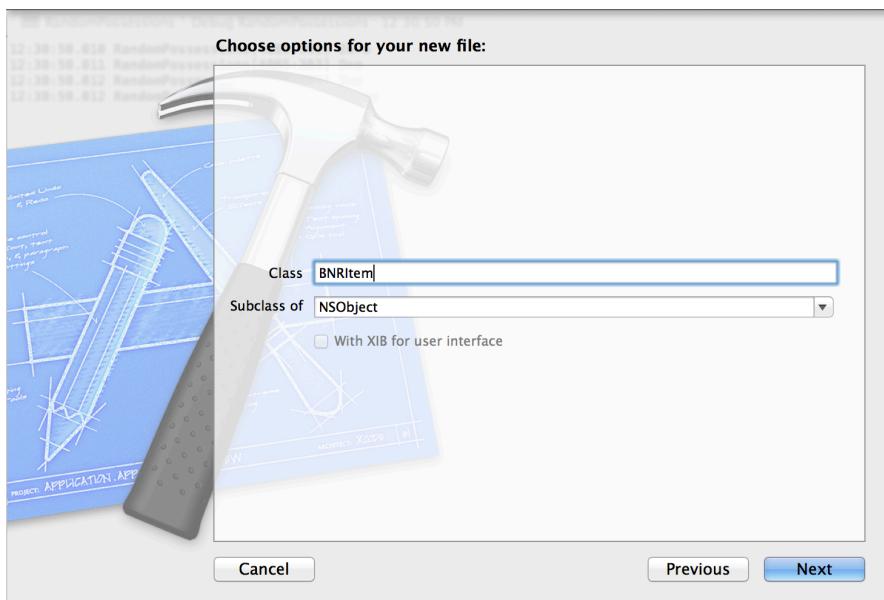
To create a new class in Xcode, choose **File → New → File...**. In the lefthand table of the panel that appears, select **Cocoa** from the **OS X** section. Then select **Objective-C class** from the upper panel and click **Next** (Figure 2.11).

Figure 2.11 Creating a class



On the next panel, name this new class **BNRItem** and enter **NSObject** as its superclass, as shown in Figure 2.12.

Figure 2.12 Choosing a superclass



Click **Next** and you will be asked where to save the files for this class. The default location is fine. Make sure the checkbox is selected for the **RandomItems** target. Click **Create**.

In the project navigator, find the class files for **BNRItem**: **BNRItem.h** and **BNRItem.m**:

- **BNRItem.h** is the *header file* (also called the *interface file*). This file declares the name of the new class, its superclass, the instance variables that each instance of this class has, and any methods this class implements.
- **BNRItem.m** is the implementation file, and it contains the code for the methods that the class implements.

Every Objective-C class has these two files. You can think of the header file as a user manual for an instance of a class and the implementation file as the engineering details that define how it really works.

Select **BNRItem.h** in the project navigator. The contents of the file look like this:

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

@interface BNRItem : NSObject

@end
```

To declare a class in Objective-C, you use the keyword `@interface` followed by the name of the new class. After a colon comes the name of the superclass. Objective-C only allows **single inheritance**, so every class can only have one superclass:

```
@interface ClassName : SuperclassName
```

The `@end` directive indicates that the class declaration has come to an end.

Notice the `@` prefixes. Objective-C retains the keywords of the C language. Additional keywords specific to Objective-C are distinguishable by the `@` prefix.

Instance variables

An “item,” in our world, is going to have a name, a serial number, a value, and a date of creation. These will be the instance variables of **BNRItem**.

Instance variables for a class are declared between curly braces immediately after the class declaration.

In **BNRItem.h**, add a set of curly braces and four instance variables to the **BNRItem** class:

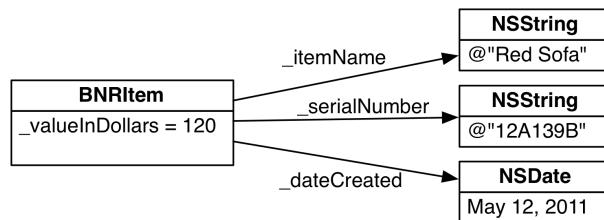
```
#import <Foundation/Foundation.h>

@interface BNRItem : NSObject
{
    NSString *_itemName;
    NSString *_serialNumber;
    int _valueInDollars;
    NSDate *_dateCreated;
}

@end
```

Now every instance of **BNRItem** will have one spot for a simple integer and three spots for pointers to objects, specifically two **NSString** instances and one **NSDate** instance. (Remember, the * denotes that the variable is a pointer.) Figure 2.13 shows an example of a **BNRItem** instance after its instance variables have been given values.

Figure 2.13 A **BNRItem** instance



Notice that Figure 2.13 shows a total of four objects: the instance of **BNRItem**, two instances of **NSString**, and an instance of **NSDate**. Each object exists independently and outside of the others. The **BNRItem** object's instance variables are the *pointers* to the other objects, not the objects themselves.

For example, every **BNRItem** instance has a pointer instance variable named `_itemName`. The `_itemName` of the **BNRItem** object shown in Figure 2.13 points to an **NSString** object whose contents are "Red Sofa". The "Red Sofa" string does not live inside the **BNRItem** object. The **BNRItem** object only knows where the "Red Sofa" string lives in memory and stores that address as `_itemName`. One way to think of this relationship is "the **BNRItem** object calls this string its `_itemName`."

The story is different for the instance variable `_valueInDollars`. This instance variable is *not* a pointer to another object; it is just an `int`. The `int` itself does live inside the **BNRItem** object.

The idea of pointers is not easy to understand at first. In the next chapter, you will learn more about objects, pointers, and instance variables, and throughout this book you will see object diagrams like Figure 2.13 to drive home the difference between an object and a pointer to an object.

Accessing instance variables

Now that instances of **BNRItem** have instance variables, you need a way to get and set their values. In object-oriented languages, we call methods that get and set instance variables *accessors*. Individually, we call them *getters* and *setters*. Without these methods, an object cannot access the instance variables of another object.

In `BNRItem.h`, declare accessor methods for the instance variables of the **BNRItem** class. You need getters and setters for `_valueInDollars`, `_itemName`, and `_serialNumber`. The `_dateCreated` instance variable will be read-only, so it only needs a getter method.

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

@interface BNRItem : NSObject
{
    NSString *_itemName;
    NSString *_serialNumber;
    int _valueInDollars;
    NSDate *_dateCreated;
}

- (void)setItemName:(NSString *)str;
- (NSString *)itemName;

- (void)setSerialNumber:(NSString *)str;
- (NSString *)serialNumber;

- (void)setValueInDollars:(int)v;
- (int)valueInDollars;

- (NSDate *)dateCreated;
@end
```

In Objective-C, the name of a setter method is **set** plus the capitalized name of the instance variable it is changing – in this case, **setItemName:**. In other languages, the name of the getter method would likely be **getItemName**. However, in Objective-C, the name of the getter method is just the name of the instance variable. Some of the cooler parts of the Cocoa Touch library make the assumption that your classes follow this convention; therefore, stylish Cocoa Touch programmers always do so.

(For those of you with some experience in Objective-C, we will talk about properties in the next chapter.)

Next, open **BNRItem**'s implementation file, **BNRItem.m**.

At the top of any implementation file, the header file of that class is always imported. The implementation of a class needs to know how it has been declared. (Importing a file is the same as including a file in C except that it ensures that the file will only be included once.)

After the import statement is the implementation block that begins with the **@implementation** keyword followed by the name of the class that is being implemented. All of the method definitions in the implementation file are inside this implementation block. Methods are defined until you close out the block with the **@end** keyword.

In **BNRItem.m**, delete anything that the template may have added between **@implementation** and **@end**. Then define the accessor methods for the instance variables that you declared in **BNRItem.h**.

Chapter 2 Objective-C

```
#import "BNRItem.h"

@implementation BNRItem

- (void)setItemName:(NSString *)str
{
    _itemName = str;
}
- (NSString *)itemName
{
    return _itemName;
}

- (void)setSerialNumber:(NSString *)str
{
    _serialNumber = str;
}

- (NSString *)serialNumber
{
    return _serialNumber;
}

- (void)setValueInDollars:(int)v
{
    _valueInDollars = v;
}

- (int)valueInDollars
{
    return _valueInDollars;
}

- (NSDate *)dateCreated
{
    return _dateCreated;
}

@end
```

Notice that each setter method sets the instance variable to whatever is passed in as an argument, and each getter method returns the value of the instance variable.

At this point, check for and fix any errors in your code that Xcode is warning you about. Some possible culprits are typos and missing semicolons.

Let's test out your new class and its accessor methods. In `main.m`, first import the header file for the `BNRItem` class.

```
#import <Foundation/Foundation.h>
#import "BNRItem.h"

int main (int argc, const char * argv[])
{
    ...
}
```

Why do you import the class header `BNRItem.h` but not, say, `NSMutableArray.h`? The `NSMutableArray` class comes from the Foundation framework, so it is included when you import

`Foundation/Foundation.h`. On the other hand, the `BNRItem` class exists in its own file, so you have to explicitly import it into `main.m`. If you do not, the compiler will not know that it exists and will complain loudly.

Next, create an instance of `BNRItem` and log its instance variables to the console.

```
int main (int argc, const char * argv[])
{
    @autoreleasepool {

        NSMutableArray *items = [[NSMutableArray alloc] init];
        [items addObject:@"One"];
        [items addObject:@"Two"];
        [items addObject:@"Three"];
        [items insertObject:@"Zero" atIndex:0];

        // For every item in the array pointed to by items...
        for (NSString *item in items) {
            // print a description
            NSLog(@"%@", item);
        }

        BNRItem *item = [[BNRItem alloc] init];
        NSLog(@"%@", item.itemName, item.dateCreated,
               item.serialNumber, item.valueInDollars);

        items = nil;
    }

    return 0;
}
```

Build and run the application. At the end of your output, you will see three `(null)` strings and a `0`. These are the values of the instance variables of your freshly-minted instance of `BNRItem`.

Figure 2.14 Instance variables' values in console

```
2013-12-08 18:43:11.237 RandomItems[4239:303] Zero
2013-12-08 18:43:11.239 RandomItems[4239:303] One
2013-12-08 18:43:11.239 RandomItems[4239:303] Two
2013-12-08 18:43:11.240 RandomItems[4239:303] Three
2013-12-08 18:43:11.240 RandomItems[4239:303] (null) (null) (null) 0
Program ended with exit code: 0
```

When an object is created, all of its instance variables are “zeroed-out.” A pointer to an objects points to `nil`; a primitive like `int` has the value of `0`.

To give the `BNRItem` object's instance variables more interesting values, you need to create new objects and pass them as arguments to the setter methods.

Chapter 2 Objective-C

In main.m, type in the following code:

```
// Notice we are omitting some of the surrounding code  
...  
BNRItem *item = [[BNRItem alloc] init];  
  
// This creates an NSString, "Red Sofa" and gives it to the BNRItem  
[item setName:@"Red Sofa"];  
  
// This creates an NSString, "A1B2C" and gives it to the BNRItem  
[item setSerialNumber:@"A1B2C"];  
  
// This sends the value 100 to be used as the valueInDollars of this BNRItem  
[item setValueInDollars:100];  
  
NSLog(@"%@", item.name, [item dateCreated],  
       [item serialNumber], [item valueInDollars]);  
  
...
```

Build and run the application. You will see the values of the three instance variables. You will still see (null) for dateCreated. Later in the chapter, you will take care of giving this instance variable a value when an object is created.

Figure 2.15 More interesting values

```
2013-12-08 18:44:56.551 RandomItems[4254:303] Zero  
2013-12-08 18:44:56.553 RandomItems[4254:303] One  
2013-12-08 18:44:56.553 RandomItems[4254:303] Two  
2013-12-08 18:44:56.554 RandomItems[4254:303] Three  
2013-12-08 18:44:56.554 RandomItems[4254:303] Red Sofa (null) A1B2C 100  
Program ended with exit code: 0
```

Using dot syntax

To get and set an instance variable, you can send explicit accessor messages:

```
BNRItem *item = [[BNRItem alloc] init];  
  
// set valueInDollars by sending an explicit message  
[item setValueInDollars:5];  
  
// get valueInDollars by sending an explicit message  
int value = [item valueInDollars];
```

Or you can use *dot syntax*, also called *dot notation*. Here is the same code using dot syntax:

```
BNRItem *item = [[BNRItem alloc] init];  
  
// set valueInDollars using dot syntax  
item.valueInDollars = 5;  
  
// get valueInDollars using dot syntax  
int value = item.valueInDollars;
```

The receiver (`item`) is followed by a `.` followed by the name of the instance variable without the leading underscore (`valueInDollars`).

Notice that the syntax is the same for both setting and getting the instance variable (`item.valueInDollars`); the difference is in which side of the assignment operator it is on.

There is no difference at runtime between accessor messages and dot syntax; the compiled code is the same and either syntax will invoke the `valueInDollars` and `setValueInDollars:` methods that you just implemented.

These days, stylish Objective-C programmers tend to use dot syntax for invoking accessors. It makes code easier to read, especially when there would traditionally be nested message calls. It is also consistent with Apple's code. It is what we will do in this book.

In `main.m`, update your code to use dot syntax to set the instance variables and to get them as part of the format string.

```
...
BNRItem *item = [[BNRItem alloc] init];

// This creates an NSString, "Red Sofa" and gives it to the BNRItem
item.setitemName:@"Red Sofa";
item.itemName=@"Red Sofa";

// This creates an NSString, "A1B2C" and gives it to the BNRItem
item.setSerialNumber:@"A1B2C";
item.serialNumber=@"A1B2C";

// This sends the value 100 to be used as the valueInDollars of this BNRItem
item.setValueInDollars:100;
item.valueInDollars = 100;

 NSLog(@"%@", [item itemName], [item dateCreated],
           [item serialNumber], [item valueInDollars]);
NSLog(@"%@", item.itemName, item.dateCreated,
        item.serialNumber, item.valueInDollars);

...

```

Class vs. instance methods

Methods come in two types: *instance methods* and *class methods*. A class method typically either creates a new instance of the class or retrieves some global property of the class. An instance method operates on a particular instance of the class. For instance, the accessors that you just implemented are all instance methods. You use them to set or get the instance variables of a particular object.

To invoke an instance method, you send the message to an instance of the class. To invoke a class method, you send the message to the class itself.

For example, when you created an instance of `BNRItem`, you sent `alloc` (a class method) to the `BNRItem` class and then `init` (an instance method) to the instance of `BNRItem` returned from `alloc`.

The `description` method is an instance method. In the next section, you are going to implement `description` in `BNRItem` to return an `NSString` object that describes an instance of `BNRItem`. Later in the chapter, you will implement a class method to create an instance of `BNRItem` using random values.

Overriding methods

A subclass can also override methods of its superclass. For example, sending the **description** message to an instance of **NSObject** returns the object's class and its address in memory as an instance of **NSString** that looks like this:

```
<BNRQuizViewController: 0x4b222a0>
```

A subclass of **NSObject** can override this method to return an **NSString** object that better describes an instance of that subclass. For example, the **NSString** class overrides **description** to return the string itself. The **NSArray** class overrides **description** to return the description of every object in the array.

Because **BNRItem** is a subclass of **NSObject** (the class that originally declares the **description** method), when you re-implement **description** in **BNRItem**, you are *overriding* it.

When overriding a method, all you need to do is define it in the implementation file; you do not need to declare it in the header file because it has already been declared by the superclass.

In **BNRItem.m**, override the **description** method. The code for a method implementation can go anywhere between `@implementation` and `@end`, as long as it is not inside the curly braces of an existing method.

```
- (NSString *)description
{
    NSString *descriptionString =
        [[NSString alloc] initWithFormat:@"%@ (%@): Worth $%d, recorded on %@", self.itemName, self.serialNumber, self.valueInDollars, self.dateCreated];

    return descriptionString;
}
```

Note what you are not doing here: you are not passing the instance variables by name (e.g., `_itemName`). Instead you are invoking accessors (via dot syntax). It is good practice to use accessor methods to access instance variables even inside a class. It is possible that an accessor method may change something about the instance variable that you are trying to access, and you want to make sure it gets the chance to do what it needs to.

Now whenever you send the message **description** to an instance of **BNRItem**, it will return an instance of **NSString** that better describes the instance.

In **main.m**, replace the statement that prints out the instance variables individually with a statement that relies on **BNRItem**'s implementation of **description**.

```
...
item.valueInDollars = 100;

NSLog(@"%@", item.itemName, item.dateCreated,
        item.serialNumber, item.valueInDollars);

// The %@ token is replaced with the result of sending
// the description message to the corresponding argument
NSLog(@"%@", item);

items = nil;
```

Build and run the application and check your results in the console.

Figure 2.16 An instance of **BNRItem** described

```
2013-12-08 18:49:05.934 RandomItems[4277:303] Zero
2013-12-08 18:49:05.935 RandomItems[4277:303] One
2013-12-08 18:49:05.936 RandomItems[4277:303] Two
2013-12-08 18:49:05.936 RandomItems[4277:303] Three
2013-12-08 18:49:05.937 RandomItems[4277:303] Red Sofa (A1B2C): Worth $100, recorded on (null)
Program ended with exit code: 0
```

What if you want to create an entirely new instance method, one that you are not overriding from the superclass? You declare the new method in the header file and define it in the implementation file. Let's see how that works by creating two new instance methods to initialize an instance of **BNRItem**.

Initializers

Right now, the **BNRItem** class has only one way to initialize an instance – the **init** method, which it inherits from the **NSObject** class. In this section, you are going to write two additional initialization methods, or *initializers*, for **BNRItem**.

In **BNRItem.h**, declare two initializers.

```
NSDate *_dateCreated;
}

- (instancetype)initWithItemName:(NSString *)name
    valueInDollars:(int)value
    serialNumber:(NSString *)sNumber;

- (instancetype)initWithItemName:(NSString *)name;

- (void)setItemName:(NSString *)str;
```

(Wondering about **instancetype**? Hold on – we will get there shortly.)

Each initializer begins with the word **init**. Naming initializers this way does not make these methods different from other instance methods; it is only a naming convention. However, the Objective-C community is all about naming conventions, which you should strictly adhere to. (Disregarding naming conventions in Objective-C results in problems that are worse than you might imagine.)

An initializer takes arguments that the object can use to initialize itself. Often, a class has multiple initializers because instances can have different initialization needs. For instance, the first initializer that you declared takes three arguments that it uses to configure the item's name, value, and serial number. So you need all of this information to initialize an instance with this method. What if you only know the item's name? Then you can use the second initializer.

The designated initializer

For each class, regardless of how many initialization methods there are, one method is chosen as the **designated initializer**. The designated initializer makes sure that every instance variable of an object is valid. (“Valid” in this context means “when you send messages to this object after initializing it, you can predict the outcome and nothing bad will happen.”)

Typically, the designated initializer has parameters for the most important and frequently used instance variables of an object. The **BNRItem** class has four instance variables, but only three are writeable. Therefore, **BNRItem**'s designated initializer should accept three arguments and provide a value within its implementation for `_dateCreated`.

In `BNRItem.h`, add a comment naming the designated initializer:

```
    NSDate *_dateCreated;  
}  
  
// Designated initializer for BNRItem  
- (instancetype)initWithItemName:(NSString *)name  
    valueInDollars:(int)value  
    serialNumber:(NSString *)sNumber;  
  
- (instancetype)initWithItemName:(NSString *)name;  
  
- (void)setItemName:(NSString *)str;
```

instancetype

The return type for both initializers is `instancetype`. This keyword can only be used for return types, and it matches the return type to the receiver. `init` methods are always declared to return `instancetype`.

Why not make the return type `BNRItem *`? That would cause a problem if the **BNRItem** class was ever subclassed. The subclass would inherit all of the methods from **BNRItem**, including this initializer and its return type. If an instance of the subclass was sent this initializer message, what would be returned? Not a pointer to a **BNRItem** instance, but a pointer to an instance of the subclass. You might think, “No problem. I will override the initializer in the subclass to change the return type.” But in Objective-C, you cannot have two methods with the same selector and different return types (or arguments). By specifying that an initialization method returns “an instance of the receiving object,” you never have to worry what happens in this situation.

id

Before the `instancetype` keyword was introduced in Objective-C, initializers returned `id` (pronounced “eye-dee”). This type is defined as “a pointer to any object.” (`id` is a lot like `void *` in C.). As of this writing, Xcode class templates still use `id` as the return type of initializers added in boilerplate code. We imagine that this will change soon.

Unlike `instancetype`, `id` can be used as more than just a return type. You can declare variables or method parameters of type `id` when you are unsure what type of object the variable will end up pointing to.

```
id objectOfUnknownType;
```

You can use `id` when using fast enumeration to iterate over an array of multiple or unknown types of objects:

```
for (id item in items) {  
    NSLog(@"%@", item);  
}
```

Note that because `id` is defined as “a pointer to any object,” you do not include an `*` when declaring a variable or method parameter of this type.

Implementing the designated initializer

In `BNRItem.m`, implement the designated initializer within the implementation block.

```
@implementation BNRItem

- (instancetype)initWithItemName:(NSString *)name
                           valueInDollars:(int)value
                            serialNumber:(NSString *)sNumber
{
    // Call the superclass's designated initializer
    self = [super init];

    // Did the superclass's designated initializer succeed?
    if (self) {
        // Give the instance variables initial values
        _itemName = name;
        _serialNumber = sNumber;
        _valueInDollars = value;
        // Set _dateCreated to the current date and time
        _dateCreated = [[NSDate alloc] init];
    }

    // Return the address of the newly initialized object
    return self;
}
```

There is a lot to talk about in this code. First, notice that you set the `_dateCreated` instance variable to point a new instance of `NSDate`, which represents the current date and time.

Next, consider the first line of code in this implementation. In the designated initializer, the first thing you always do is call the superclass’s designated initializer using `super`. The last thing you do is return a pointer to the successfully initialized object using `self`. So to understand what is going on in an initializer, you need to know about `self` and `super`.

self

Inside a method, `self` is an implicit local variable. There is no need to declare it, and it is automatically set to point to the object that was sent the message. (Most object-oriented languages have this concept, but some call it `this` instead of `self`.) Typically, `self` is used so that an object can send a message to itself:

```
- (void)chickenDance
{
    [self pretendHandsAreBeaks];
    [self flapWings];
    [self shakeTailFeathers];
}
```

In the last line of an `init` method, you always return the newly initialized object so that the caller can assign it to a variable:

```
return self;
```

super

When you are overriding a method, you often want to keep what the method of the superclass is doing and have your subclass add something new on top of that. To make this easier, there is a compiler directive in Objective-C called `super`:

```
- (void)someMethod
{
    [super someMethod];
    [self doMoreStuff];
}
```

How does `super` work? Usually when you send a message to an object, the search for a method of that name starts in the object's class. If there is no such method, the search continues in the superclass of the object. The search will continue up the inheritance hierarchy until a suitable method is found. (If it gets to the top of the hierarchy and no method is found, an exception is thrown.)

When you send a message to `super`, you are sending a message to `self`, but the search for the method skips the object's class and starts at the superclass. In the case of `BNRItem`'s designated initializer, you send the `init` message to `super`. This calls `NSObject`'s implementation of `init`.

Confirming initialization success

Now let's look at the next line where you confirm what the superclass's initializer returned. If an initializer message fails, it will return `nil`. Therefore, it is a good idea to save the return value of the superclass's initializer into the `self` variable and confirm that it is not `nil` before doing any further initialization.

Instance variables in initializers

Now we get to the core of this method where the instance variables are given values. Earlier we told you not to access instance variables directly and to use accessor methods. Now we are asking you to break that rule when writing initializers.

While an initializer is being executed, the object is being born, and you cannot be sure that its instance variables have all been set to usable values. When writing a method, you typically assume that all of an object's instance variables have been set to usable values. Thus, invoking a method (like an accessor) at a time when this may not be the case is unsafe. At Big Nerd Ranch, we typically set the instance variables directly in initializers, instead of calling accessor methods.

Some very good Objective-C programmers do use accessors in initializers. They argue that if the accessor does something complicated, you want that code in exactly one place; replicating it in your initializer is bad. We are not religiously devoted to either approach, but in this book we will set instance variables directly in initializers.

Other initializers and the initializer chain

Let's implement the second initializer for the `BNRItem` class. In this initializer's definition, you are not going to replicate the code in the designated initializer. Instead, this initializer will simply call the designated initializer, passing the information it is given for the `_itemName` and default values for the other arguments.

In `BNRItem.m`, implement `initWithItemName:`.

```
- (instancetype)initWithItemName:(NSString *)name
{
    return [self initWithItemName:name
                      valueInDollars:0
                           serialNumber:@""];
}
```

The **BNRItem** class already has a third initializer – **init**, which it inherits from **NSObject**. If **init** is used to initialize an instance of **BNRItem**, none of the stuff that you put in the designated initializer will happen. Therefore, you must override **init** in **BNRItem** to link to **BNRItem**'s designated initializer.

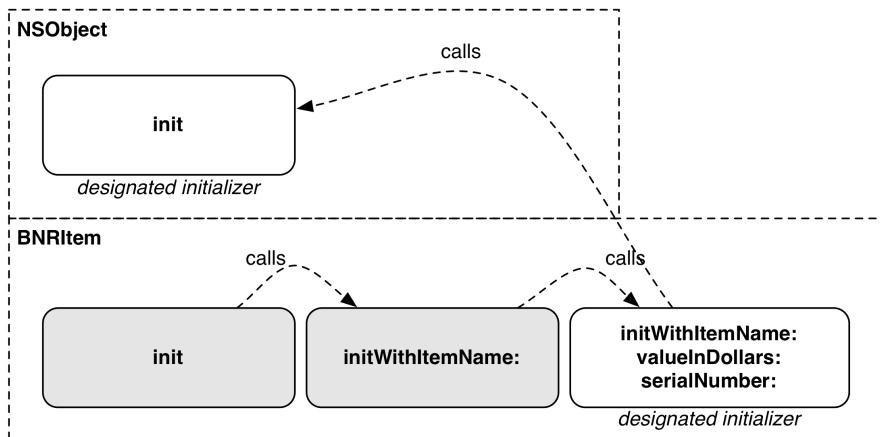
In **BNRItem.m**, override **init** to call **initWithItemName:**, passing a default value for the item's name.

```
- (instancetype)init
{
    return [self initWithItemName:@"Item"];
}
```

Now when **init** is sent to an instance of **BNRItem**, the method will call **initWithItemName:** with a default value for **_itemName**, which will call the designated initializer, **initWithItemName:valueInDollars:serialNumber:** with default values for **_valueInDollars** and **_serialNumber**.

The relationships between **BNRItem**'s initializers are shown in Figure 2.17; the designated initializers are white, and the additional initializers are gray.

Figure 2.17 A chain of initializers



Using initializers in a chain reduces the possibility of error and makes maintaining code easier. The programmer who created the class makes it clear which initializer is the designated initializer. You only write the core of the initializer once in the designated initializer, and other initialization methods simply call the designated initializer (directly or indirectly) with default values.

Let's form some simple rules for initializers from these ideas.

- A class inherits all initializers from its superclass and can add as many as it wants for its own purposes.
- Each class picks one initializer as its *designated initializer*.
- The designated initializer calls the superclass's designated initializer (directly or indirectly) before doing anything else.
- Any other initializers call the class's designated initializer (directly or indirectly).
- If a class declares a designated initializer that is different from its superclass, the superclass's designated initializer must be overridden to call the new designated initializer (directly or indirectly).

Using initializers

Now that you have a designated initializer for **BNRItem**, you can use it instead of setting instance variables individually.

In **main.m**, remove the creation of the single **BNRItem** instance and all three setter messages. Then add code that creates an instance and sets its instance variables using the designated initializer.

```
...
// For every item in the items array ...
for (NSString *item in items) {
    // ... print a description of the current item
    NSLog(@"%@", item);
}

BNRItem *item = [[BNRItem alloc] init];
item.itemName = @"Red Sofa";
item.serialNumber = @"A1B2C";
item.valueInDollars = 100;

BNRItem *item = [[BNRItem alloc] initWithItemName:@"Red Sofa"
                                         valueInDollars:100
                                         serialNumber:@"A1B2C"];

NSLog(@"%@", item);
...
```

Build and run the application. Notice that the console now prints a single **BNRItem** instance that was instantiated with the values passed to the **BNRItem** class's designated initializer.

Let's confirm that your other two initializers work as expected. In **main.m**, create two additional instances of **BNRItem** using **initWithItemName:** and **init**.

```

...
BNRItem *item = [[BNRItem alloc] initWithItemName:@"Red Sofa"
                                         valueInDollars:100
                                         serialNumber:@"A1B2C"];
NSLog(@"%@", item);

BNRItem *itemWithNibName = [[BNRItem alloc] initWithItemName:@"Blue Sofa"];
NSLog(@"%@", itemWithNibName);

BNRItem *itemWithNoName = [[BNRItem alloc] init];
NSLog(@"%@", itemWithNoName);

items = nil;
}
return 0;
}

```

Build and run the application and check the console to confirm that **BNRItem**'s initialization chain is working.

Figure 2.18 Three initializers at work

```

2013-12-08 18:55:18.620 RandomItems[4302:303] Zero
2013-12-08 18:55:18.622 RandomItems[4302:303] One
2013-12-08 18:55:18.623 RandomItems[4302:303] Two
2013-12-08 18:55:18.623 RandomItems[4302:303] Three
2013-12-08 18:55:18.628 RandomItems[4302:303] Red Sofa (A1B2C): Worth $100, recorded on 2013-12-08 23:55:18 +0000
2013-12-08 18:55:18.629 RandomItems[4302:303] Blue Sofa (): Worth $0, recorded on 2013-12-08 23:55:18 +0000
2013-12-08 18:55:18.629 RandomItems[4302:303] Item (): Worth $0, recorded on 2013-12-08 23:55:18 +0000
Program ended with exit code: 0

```

There is only one thing left to do to complete the **BNRItem** class. You are going to write a method that creates an instance and initializes it with random values. This method will be a class method.

Class methods

Class methods typically either create new instances of the class or retrieve some global property of the class. Class methods do not operate on an instance or have any access to instance variables.

Syntactically, class methods differ from instance methods by the first character in their declaration. An instance method uses the - character just before the return type, and a class method uses the + character.

In **BNRItem.h**, declare a class method that will create a random item.

```

@interface BNRItem : NSObject
{
    NSString *_itemName;
    NSString *_serialNumber;
    int _valueInDollars;
    NSDate *_dateCreated;
}

+ (instancetype)randomItem;

- (instancetype)initWithItemName:(NSString *)name
                           valueInDollars:(int)value
                           serialNumber:(NSString *)sNumber;

```

Chapter 2 Objective-C

Notice the order of the declarations in the header file. Instance variables come first, followed by class methods, followed by initializers, followed by any other instance methods. This convention makes header files easier to read.

In `BNRItem.m`, implement `randomItem` to create, configure, and return a `BNRItem` instance. (Make sure this method is between the `@implementation` and `@end`.)

```
+ (instancetype)randomItem
{
    // Create an immutable array of three adjectives
    NSArray *randomAdjectiveList = @[@"Fluffy", @"Rusty", @"Shiny"];

    // Create an immutable array of three nouns
    NSArray *randomNounList = @[@"Bear", @"Spork", @"Mac"];

    // Get the index of a random adjective/noun from the lists
    // Note: The % operator, called the modulo operator, gives
    // you the remainder. So adjectiveIndex is a random number
    // from 0 to 2 inclusive.
    NSInteger adjectiveIndex = arc4random() % [randomAdjectiveList count];
    NSInteger nounIndex = arc4random() % [randomNounList count];

    // Note that NSInteger is not an object, but a type definition
    // for "long"

    NSString *randomName = [NSString stringWithFormat:@"%@ %@",  

                           [randomAdjectiveList objectAtIndex:adjectiveIndex],  

                           [randomNounList objectAtIndex:nounIndex]];

    int randomValue = arc4random() % 100;

    NSString *randomSerialNumber = [NSString stringWithFormat:@"%@%c%c%c%c%c",
                                    '0' + arc4random() % 10,  

                                    'A' + arc4random() % 26,  

                                    '0' + arc4random() % 10,  

                                    'A' + arc4random() % 26,  

                                    '0' + arc4random() % 10];

    BNRItem *newItem = [[self alloc] initWithitemName:randomName  

                                valueInDollars:randomValue  

                                serialNumber:randomSerialNumber];
    return newItem;
}
```

First, at the beginning of this method, notice the syntax for creating the two arrays `randomAdjectiveList` and `randomNounList` – an @ symbol followed by square brackets. Within the brackets is a comma-delimited list of objects that will populate the array. (In this case, the objects are instances of `NSString`.) This syntax is shorthand for creating instances of `NSArray`. Note that it always creates an immutable array. You can only use this shorthand if you do not need the resulting array to be mutable.

After creating the arrays, `randomItem` creates a string from a random adjective and noun, a random integer value, and another string from random numbers and letters.

Finally, the method creates an instance of `BNRItem` and sends it the designated initializer message with these randomly-created objects and `int` as parameters.

In this method, you also used `stringWithFormat:`, which is a class method of `NSString`. This message is sent directly to the `NSString` class, and the method returns an `NSString` instance with the passed-in parameters. In Objective-C, class methods that return an object of their type (like `stringWithFormat:` and `randomItem`) are called *convenience methods*.

Notice the use of `self` in `randomItem`. Because `randomItem` is a class method, `self` refers to the `BNRItem` class itself instead of an instance. Class methods should use `self` in convenience methods instead of their class name so that a subclass can be sent the same message. In this case, if you create a subclass of `BNRItem` called `BNRToxicWasteItem`, you could do this:

```
BNRToxicWasteItem *item = [BNRToxicWasteItem randomItem];
```

Testing your subclass

For the final version of `RandomItems` in this chapter, you are going to fill the `items` array with 10 randomly-created instances of `BNRItem`. Then you will loop through the array and log each item (Figure 2.19).

Figure 2.19 Random items

```
2013-10-29 18:17:42.880 RandomItems[64653:303] Rusty Spork (8Q2U8): Worth $73, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.880 RandomItems[64653:303] Shiny Spork (5Y2V3): Worth $40, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.881 RandomItems[64653:303] Rusty Spork (2F9Z7): Worth $40, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.881 RandomItems[64653:303] Rusty Bear (8G5V6): Worth $99, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.881 RandomItems[64653:303] Shiny Spork (3P9B1): Worth $10, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.882 RandomItems[64653:303] Rusty Mac (6R5C1): Worth $93, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.882 RandomItems[64653:303] Fluffy Spork (3E400): Worth $1, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.882 RandomItems[64653:303] Fluffy Mac (3A6T4): Worth $30, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.883 RandomItems[64653:303] Shiny Spork (8S3I1): Worth $77, recorded on 2013-10-29 22:17:42 +0000
2013-10-29 18:17:42.883 RandomItems[64653:303] Rusty Spork (4F6F9): Worth $65, recorded on 2013-10-29 22:17:42 +0000
Program ended with exit code: 0
```

In `main.m`, delete all of the code except for the creation and destruction of the `items` array. Then add 10 random `BNRItem` instances to the array and log them.

```
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSMutableArray *items = [[NSMutableArray alloc] init];

        [items addObject:@"One"];
        [items addObject:@"Two"];
        [items addObject:@"Three"];
        [items insertObject:@"Zero" atIndex:0];

        /* For every item in the items array... */
        for (NSString *item in items) {
            // ... print a description of the current item
            NSLog(@"%@", item);
        }

        BNRItem *item = [[BNRItem alloc] initWithItemName:@"Red Sofa"
                                                 valueInDollars:100
                                                 serialNumber:@"A1B2C"];

        NSLog(@"%@", item);

        BNRItem *itemWithNoName = [[BNRItem alloc] initWithItemName:@""];
        NSLog(@"%@", itemWithNoName);

        for (int i = 0; i < 10; i++) {
            BNRItem *item = [BNRItem randomItem];
            [items addObject:item];
        }

        for (BNRItem *item in items) {
            NSLog(@"%@", item);
        }

        items = nil;
    }
    return 0;
}
```

Notice that you do not use fast enumeration in the first loop because you are adding to the array within the loop.

Build and run your application and then check the output in the console.

More on NSArray and NSMutableArray

You will frequently use arrays when developing iOS applications, so let's go over some more array-related details.

An Objective-C array can contain objects of different types. For example, although your `items` array currently only contains instances of `BNRItem`, you could add an instance of `NSDate` or any other Objective-C object. This is different from most strongly typed languages where an array can only hold objects of a single type.

Objective-C arrays can only hold references to Objective-C objects. Primitives and C structures cannot be added to an Objective-C array. If you need to add primitives or C structures, you can “wrap” them in Objective-C objects written for this purpose, including **NSNumber**, **NSValue**, and **NSData**.

Note that you cannot add **nil** to an array. If you need to add “holes” to an array, you must use **NSNull**. **NSNull** is a class whose only instance is meant to stand in for **nil** and is used specifically for this task.

```
[items addObject:[NSNull null]];
```

When accessing members of an array, you have used the **objectAtIndex:** message with the index of the object you want returned. This, like many other elements of Objective-C, is very verbose. Thus, there exists a shorthand syntax for accessing members of an array:

```
NSString *foo = items[0];
```

This line of code is equivalent to sending **objectAtIndex:** to **items**.

```
NSString *foo = [items objectAtIndex:0];
```

In **BNRItem.m**, update **randomItem** to use this syntax when creating the random name.

```
+ (instancetype)randomItem
{
    ...
    NSString *randomName = [NSString stringWithFormat:@"%@ %@",
    randomAdjectiveList objectAtIndex:adjectiveIndex],
    randomNounList objectAtIndex:nounIndex];

    NSString *randomName = [NSString stringWithFormat:@"%@ %@",
```

~~randomAdjectiveList[adjectiveIndex],~~
~~randomNounList[nounIndex]];~~

 int randomValue = arc4random() % 100;
 ...
 return newItem;
}

Build and run to confirm that the program works the same as before.

The nested brackets that you end up with can make things confusing because they are used in two distinct ways: one use sends a message and the other use accesses items in an array. Sometimes, it can be clearer to stick with sending the typed-out message. Other times, it is nice to avoid typing the finger-numbing **objectAtIndex:**.

Whichever syntax you use, it is important to understand that there is no difference in your application: the compiler turns the shorthand syntax into code that sends the **objectAtIndex:** message.

In an **NSMutableArray**, you can use a similar shorthand syntax to add and replace objects.

```
NSMutableArray *items = [[NSMutableArray alloc] init];
items[0] = @"A"; // Add @"A"
items[1] = @"B"; // Add @"B"
items[0] = @"C"; // Replace @"A" with @"C"
```

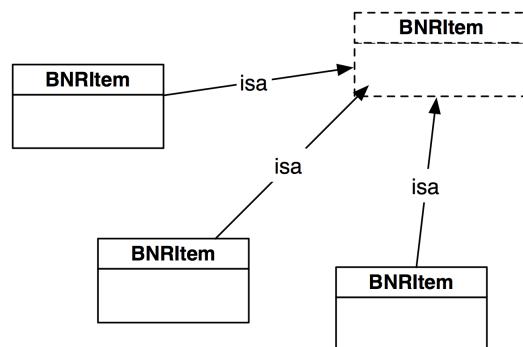
These lines are equivalent to sending `insertObject:atIndex:` and `replaceObjectAtIndex:withObject:` messages to `items`.

Exceptions and Unrecognized Selectors

At runtime, when a message is sent to an object, that object goes to the class that created it and says, “I was sent this message. Run the code for the matching method.” This is different than in most compiled languages, where the method to be executed is determined at compile time.

How does an object know which class created it? It uses its `isa` pointer. Every object has an instance variable called `isa`. When an object is created, the class sets the `isa` instance variable of the returned object to point back at that class (Figure 2.20). It is called the `isa` pointer because an object “is a” instance of that class. Although you probably will never explicitly use the `isa` pointer, its existence gives Objective-C its power.

Figure 2.20 The `isa` pointer



An object only responds to a message if its class (pointed to by its `isa` pointer) implements the associated method. Because this happens at runtime, Xcode cannot always figure out at compile time (when the application is built) whether an object will respond to a message. Xcode will give you an error if it thinks you are sending a message to an object that will not respond, but if it is not sure, it will let the application build.

If, for some reason (and there are many possibilities), you end up sending a message to an object that does not respond, your application will throw an *exception*. Exceptions are also known as *run-time errors* because they occur once your application is running, as opposed to *compile-time errors* that show up when your application is being built, or compiled.

To practice dealing with exceptions, you are going to cause one in `RandomItems`.

In `main.m`, get the last item in the array using the `lastObject` method of `NSArray`. Then send this item a message that it will not understand:

```
#import <Foundation/Foundation.h>
#import "BNRItem.h"

int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSMutableArray *items = [[NSMutableArray alloc] init];

        for (int i = 0; i < 10; i++) {
            BNRItem *item = [BNRItem randomItem];
            [items addObject:item];
        }

        id lastObj = [items lastObject];

        // lastObj is an instance of BNRItem and will not understand the count message
        [lastObj count];

        for (BNRItem *item in items) {
            NSLog(@"%@", item);
        }

        items = nil;
    }
    return 0;
}
```

Build and run the application. Your application will compile, start running, and then halt. Check your console and find the line that looks like this:

```
2014-01-19 12:23:47.990 RandomItems[10288:707] ***
Terminating app due to uncaught exception 'NSInvalidArgumentException', reason:
'-[BNRItem count]: unrecognized selector sent to instance 0x100117280'
```

This is what an exception looks like. What exactly is it saying? First, it tells you the date, time, and name of the application. You can ignore that information and focus on what comes after the “***.” That line tells you that an exception occurred and the reason.

The reason is the most important piece of information an exception gives you. Here the reason tells you that an *unrecognized selector* was sent to an instance. You know that selector means message. You sent a message to an object, and the object does not implement that method.

The type of the receiver and the name of the message are also in this output, which makes it easier to debug. An instance of **BNRItem** was sent the message **count**. The - at the beginning tells you the receiver was an instance of **BNRItem**. A + would mean the class itself was the receiver.

There are two important lessons to take away from this. First, always check the console if your application halts or crashes; errors that occur at runtime (exceptions) are just as important as those that occur during compiling. Second, remember that *unrecognized selector* means the message you are sending is not implemented by the receiver. You will make this mistake more than once, and you will want to be able to diagnose it quickly.

Some languages use try and catch blocks to handle exceptions. While Objective-C has this ability, we do not use it very often in application code. Typically, an exception is a programmer error and should be fixed in the code instead of handled at runtime.

In `main.m`, remove the exception-causing code.

```
for (int i = 0; i < 10; i++) {
    BNRItem *item = [BNRItem randomItem];
    [items addObject:p];
}

id lastObj = [items lastObject];
[lastObj count];

for (BNRItem *item in items) {
    NSLog(@"%@", item);
}
```

Challenges

Most chapters in this book will finish with at least one challenge that encourages you to take your work in the chapter one step further and prove to yourself what you have learned. We suggest that you tackle as many of these challenges as you can to cement your knowledge and move from *learning* iOS development from us to *doing* iOS development on your own.

Challenges come in three levels of difficulty:

- Bronze challenges typically ask you to do something very similar to what you did in the chapter. These challenges reinforce what you learned in the chapter and force you to type in similar code without having it laid out in front of you. Practice makes perfect.
- Silver challenges require you to do more digging and more thinking. You will need to use methods, classes, and properties that you have not seen before, but the tasks are still similar to what you did in the chapter.
- Gold challenges are difficult and can take hours to complete. They require you to understand the concepts from the chapter and then do some quality thinking and problem-solving on your own. Tackling these challenges will prepare you for the real-world work of iOS development.

Before beginning any challenge, *always make a copy of your project directory in Finder and attack the challenge in that copy*. Many chapters build on previous chapters, and working on challenges in a copy of the project assures you will be able to progress through the book.

Bronze Challenge: Bug Finding

Create a bug in your program by asking for the eleventh item in the array. Run it and note the exception that gets thrown.

Silver Challenge: Another Initializer

Create another initializer method for the `BNRItem` class. This initializer is *not* the designated initializer of `BNRItem`. It takes an instance of `NSString` that identifies the `itemName` of the item and an instance of `NSString` that identifies the `serialNumber`.

Gold Challenge: Another Class

Create a subclass of **BNRItem** named **BNRContainer**. An instance of **BNRContainer** should have an array of **subitems** that contains instances of **BNRItem**. Printing the description of a **BNRContainer** object should show you the name of the container, its value in dollars (a sum of all items in the container plus the value of the container itself), and a list of every instance of **BNRItem** it contains. A properly-written **BNRContainer** class can contain instances of **BNRContainer**. It can also report back its full value and every contained item properly.

Are You More Curious?

In addition to Challenges, many chapters will conclude with one or more “For the More Curious” sections. These sections offer deeper explanations of or additional information about the topics presented in the chapter. The knowledge in these sections is not absolutely essential to get you where you are going, but we hope you will find it interesting and useful.

For the More Curious: Class Names

In simple applications like `RandomItems`, you only need a few classes. As applications grow larger and more complex, the number of classes grows. At some point, you will run into a situation where you have two classes that could easily be named the same thing. This is bad news. If two classes have the same name, it is impossible for your program to figure out which one it should use. This is known as a *namespace collision*.

Other languages solve this problem by declaring classes inside a *namespace*. You can think of a namespace as a group to which classes belong. To use a class in these languages, you have to specify both the class name and the namespace.

Objective-C has no notion of namespaces. Instead, class names are prefixed with two or three letters to keep them distinct. For example, in this exercise, the class was named **BNRItem** instead of **Item**.

Stylish Objective-C programmers always prefix their classes. The prefix is typically related to the name of the application you are developing or the library that it belongs to. For example, if I were writing an application named “MovieViewer,” I would prefix all classes with **MOV**. Classes that you will use across multiple projects typically bear a prefix that is related to your name (**CBK**), your company’s name (**BNR**), or a portable library (a library for dealing with maps might use **MAP**).

Notice that Apple’s classes have prefixes, too. Apple’s classes are organized into frameworks, and each framework has its own prefix. For instance, the **UILabel** class belongs to the UIKit framework. The classes **NSArray** and **NSString** belong to the Foundation framework. (The **NS** stands for NeXTSTEP, the platform for which these classes were originally designed.)

For your classes, you should use three-letter prefixes. Two-letter prefixes are reserved by Apple for use in framework classes. Although nothing is stopping you from creating a class with a two-letter prefix, you should use three-letter prefixes to eliminate the possibility of namespace collisions with Apple’s present and future classes.

For the More Curious: #import and @import

When Objective-C was new, the system did not ship with many classes. Eventually, however, there were enough classes that it became necessary to organize them into frameworks. In your source code, you would typically `#import` the master header file for a framework:

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

And that file would `#import` all the headers in that framework, like this:

```
#import <Foundation/NSArray.h>
#import <Foundation/NSAutoreleasePool.h>
#import <Foundation/NSBundle.h>
#import <Foundation/NSByteOrder.h>
#import <Foundation/NSDateCalendar.h>
#import <Foundation/NSCharacterSet.h>
...
...
```

And then you would explicitly link that framework into your program at compile time.

This was easy to implement; it was using the existing C preprocessor to copy all these headers into the file that was about to be compiled.

This approach worked pretty well for about a decade. Then, as more classes were added to the frameworks and more frameworks went into each project, we noticed that the compiler was spending most of its time parsing and processing those same standard headers again and again. So, the *precompiled header file* was added to every project. The first time you compiled your project, the headers listed in the that file would be compiled once and the result would be cached away. Having this pre-digested clump of headers made compiling all the other files much, much faster. The project you just created has the file `RandomItems-Prefix.pch` and it forces the build system to precompile the headers for the Foundation framework:

```
#ifdef __OBJC__
    #import <Foundation/Foundation.h>
#endif
```

You still had to explicitly link that framework into your program at compile time.

That worked pretty well for another decade, but recently Apple realized that developers were not maintaining their `.pch` files effectively. So, they made the compiler smarter and the `@import` directive was introduced:

```
@import Foundation;
```

This tells the compiler, “Hey, I’m using the Foundation module. You figure out how to make that work.” The compiler is given a lot of freedom to optimize the preprocessing and caching of header files. (This also eliminates the need to explicitly link the module into the program – when the compiler sees the `@import`, it makes a note to link in the appropriate module.)

As we write this, only Apple can create modules that can be used with `@import`. To use classes and frameworks that you create, you will still need to use `#import`.

We are writing this book on Xcode 5.0, and `#import` still appears in the template projects and files, but we are certain that in the near future `@import` will be ubiquitous.

3

Managing Memory with ARC

In this chapter, you will learn how memory is managed in iOS and the concepts that underlie *automatic reference counting*, or ARC. Let's start with some basics of application memory.

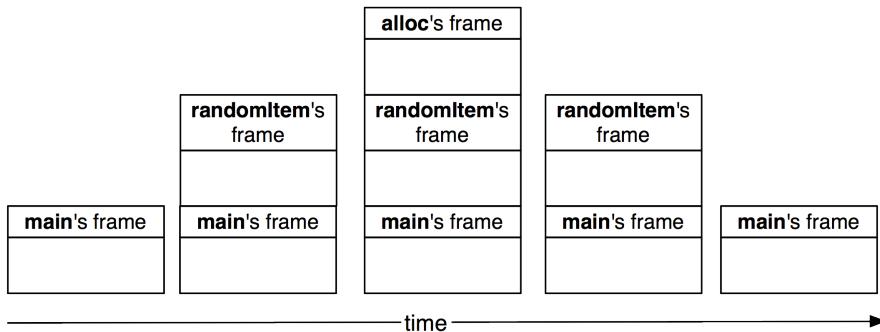
The Stack

When a method (or function) is executed, a chunk of memory is allocated from a part of memory called the *stack*. This chunk of memory is called a *frame*, and the frame stores the values for variables declared inside the method. A variable declared inside a method is called a *local variable*.

When an application launches and runs `main()`, the frame for `main()` is put on the stack. When `main()` calls another method (or function), the frame for that method is put on top of the stack. Of course, that method could call another method, and so on, until you have a towering stack of frames. As each method or function finishes, its frame is “popped off” the top of the stack and destroyed. If the method is called again, a new frame will be allocated and put on the stack.

For example, in the `RandomItems` application, the `main` function runs `BNRItem`'s `randomItem` method, which in turn runs `alloc`. The stack would look like Figure 3.1. Notice that `main()`'s frame stays alive while the other methods are executing because it has not yet finished executing.

Figure 3.1 Stack growing and shrinking



The `randomItem` method runs inside a loop in `main()`. With every iteration of the loop, the stack grows and shrinks as frames are put on and popped off the stack.

The Heap

There is another part of memory called the *heap* that is separate from the stack. The reason for the names “heap” and “stack” has to do with how you visualize them. The stack can be visualized as an

orderly stack of frames. The heap, on the other hand, is where all Objective-C objects live. It is a giant heaping mess of objects. You use pointers to keep track of where those objects are stored in the heap.

When you send the `alloc` message to a class, a chunk of memory is allocated from the heap. This chunk is your object, and it includes space for the object's instance variables. An instance of `BNRItem` has five instance variables: four pointers (`isa`, `_itemName`, `_serialNumber`, and `_dateCreated`) and an `int` (`_valueInDollars`). Thus, the chunk of memory that is allocated includes space for one `int` and four pointers. These pointers store addresses of other objects in the heap.

An iOS application creates objects at launch and will typically continue to create objects for as long as the application is running. If heap memory were infinite, the application could create all the objects that it wanted to and have them exist for the entire run of the application.

But an application gets only so much heap memory, and memory on an iOS device is especially limited. Thus, this resource must be managed: It is important to destroy objects that are no longer needed to free up heap memory so that it can be reused to create new objects. On the other hand, it is critical not to destroy objects that are still needed.

ARC and memory management

The good news is that you do not need to keep track of which objects should live and die. Your application's memory management is handled for you by ARC, which stands for Automatic Reference Counting. All of the applications in this book will use ARC. Before ARC was available, applications used *manual reference counting*. There is more information about manual reference counting at the end of the chapter.

ARC can be relied on to manage your application's memory automatically for the most part. However, it is important to understand the concepts behind it to know how to step in when you need to. Let's start with the idea of object ownership.

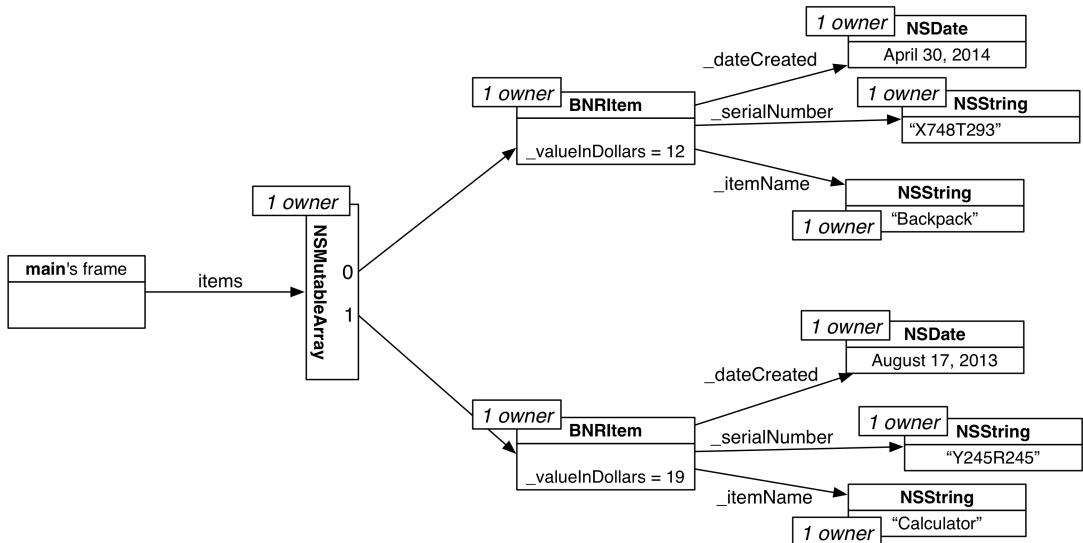
Pointer Variables and Object Ownership

Pointer variables imply *ownership* of the objects that they point to.

- When a method (or function) has a local variable that points to an object, that variable is said to *own* the object being pointed to.
- When an object has an instance variable that points to another object, the object with the pointer is said to *own* the object being pointed to.

Think back to your `RandomItems` application. In this application, an instance of `NSMutableArray` is created in `main()` and then ten `BNRItem` instances are added to it. Figure 3.2 shows some of the objects in `RandomItems` and the pointers that reference them.

Figure 3.2 RandomItems object diagram (with only two items)



Within **main()**, the local variable **items** points to an instance of **NSMutableArray**, so **main()** owns that **NSMutableArray** instance.

The array, in turn, owns the **BNRItem** instances. A collection object, like an instance of **NSMutableArray**, holds pointers to objects instead of actually containing them, and these pointers imply ownership: an array always owns the objects that are “in” the array.

Finally, each **BNRItem** instance owns the objects pointed to by its instance variables.

The idea of object ownership is useful for determining whether an object will be destroyed so that its memory can be reused.

- **An object with no owners will be destroyed.** An ownerless object cannot be sent messages and is isolated and useless to the application. Keeping it around wastes precious memory. This is called a *memory leak*.
- **An object with one or more owners will not be destroyed.** If an object is destroyed but another object or method still has a pointer to it (or, more accurately, a pointer that stores the address where the object *used* to live), then you have a dangerous situation: sending a message via this pointer may crash your application. Destroying an object that is still needed is called *premature deallocation*. It is also known as a *dangling pointer* or a *dangling reference*.

How objects lose owners

Here are the ways that an object can lose an owner:

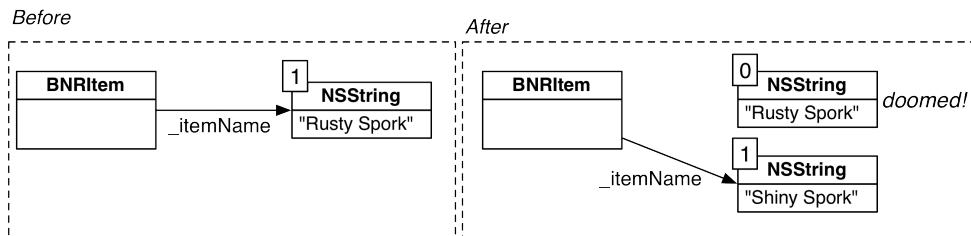
- A variable that points to the object is changed to point to another object.
- A variable that points to the object is set to **nil**.

- The owner of the object is itself destroyed.
- An object in a collection, like an array, is removed from that collection.

Let's take a look at each of these situations.

Changing a pointer

Imagine an instance of **BNRItem**. Its `_itemName` instance variable points to an **NSString** instance `@“Rusty Spork”`. If you polished the rust off that spork, it would become a shiny spork, and you would want to change the `_itemName` to point at a different **NSString**.



When the value of `_itemName` changes from the address of the “Rusty Spork” string to the address of the “Shiny Spork” string, the “Rusty Spork” string loses an owner. If it has no other owners, then it will be destroyed.

Setting a pointer to nil

Setting a pointer to `nil` represents the absence of an object. For example, say you have a **BNRItem** instance that represents a television. Then, someone scratches off the television’s serial number. You would set its `_serialNumber` instance variable to `nil`. The **NSString** instance that `_serialNumber` previously pointed to loses an owner.

The owner is destroyed

When an object is destroyed, the objects that it owns lose an owner. In this way, one object being deallocated can cause a cascade of object deallocations.

Through its local variables, a method or a function can own objects. When the method or function is done executing and its frame is popped off the stack, the objects it owns will lose an owner.

Removing an object from a collection

There is one more important way an object can lose an owner. An object in a collection object is owned by the collection object. When you remove an object from a mutable collection object, like an instance of **NSMutableArray**, the removed object loses an owner.

```
[items removeObject:item]; // Object pointed to by item loses an owner
```

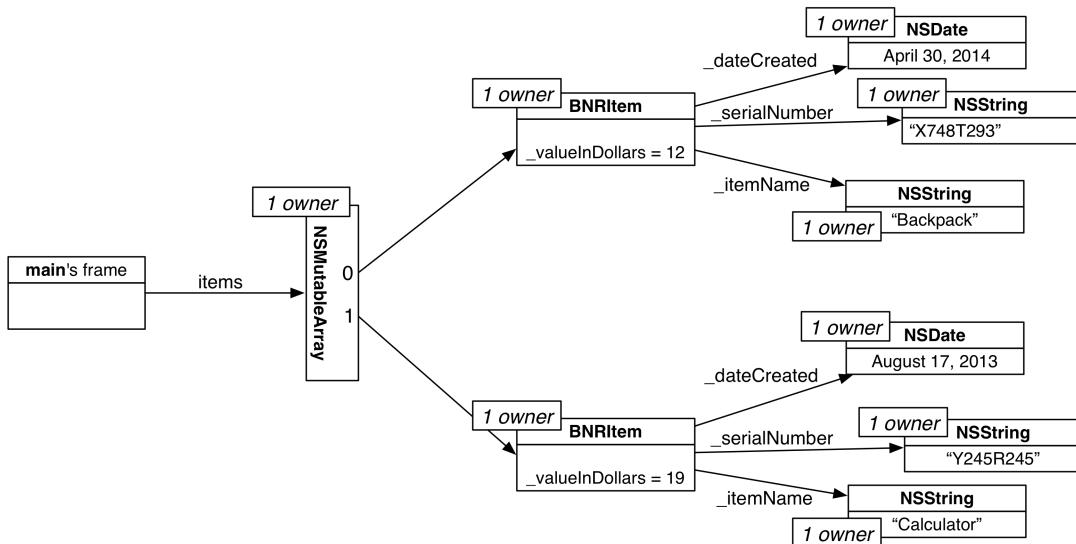
Keep in mind that losing an owner by any of these means does not necessarily result in the object being destroyed; if there is still another pointer to the object somewhere, then the object will continue to exist. When an object loses its last owner, the result is certain and appropriate death.

Ownership chains

Because objects own other objects, which can own other objects, the destruction of a single object can set off a chain reaction of loss of ownership, object destruction, and freeing up of memory.

There is an example of this in `RandomItems`. Take another look at the object diagram for this application.

Figure 3.3 Objects and pointers in `RandomItems`



In `main.m`, after you finish printing out the array, you set the `items` variable to `nil`. Setting `items` to `nil` causes the array to lose its only owner, so the array is destroyed.

But the destruction does not stop there. When the array is destroyed, all of the pointers to the `BNRItem` instances are destroyed. Once these variables are gone, no one owns any of the items, so they are destroyed.

Finally, destroying a `BNRItem` destroys its instance variables, which leaves the objects pointed to by those variables unowned. So they, too, get destroyed.

Let's add some code so that you can see this destruction as it happens. `NSObject` implements a `dealloc` method, which is sent to an object just before it is destroyed. You can override `dealloc` in `BNRItem` to print something to the console when an item is destroyed.

In the `RandomItems` project, open `BNRItem.m` and override `dealloc`.

```

- (void)dealloc
{
    NSLog(@"Destroyed: %@", self);
}
  
```

In `main.m`, add the following line of code.

```
NSLog(@"Setting items to nil...");  
items = nil;
```

Build and run the application. After the `items` print out, you will see the message announcing that `items` is being set to `nil`. Then, you will see the destruction of each `BNRItem` logged to the console.

At the end, there are no more objects taking up memory, and only the `main` function remains. All this automatic clean-up and memory recycling occurs as the result of setting `items` to `nil`. That is the power of ARC.

Strong and Weak References

We have said that anytime a pointer variable points to an object, that object has an owner and will stay alive. This is known as a *strong reference*.

A variable can optionally *not* take ownership of an object that it points to. A variable that does not take ownership of an object is known as a *weak reference*.

A weak reference is useful for preventing a problem called a *strong reference cycle* (also known as a *retain cycle*.) A strong reference cycle occurs when two or more objects have strong references to each other. This is bad news. When two objects own each other, they can never be destroyed by ARC. Even if every other object in the application releases ownership of these objects, these objects (and any objects that they own) will continue to exist inside their bubble of mutual ownership.

Thus, a strong reference cycle is a memory leak that ARC needs your help to fix. You fix it by making one of the references weak.

Let's introduce a strong reference cycle in `RandomItems` to see how this works. First, you are going to give an instance of `BNRItem` the ability to hold another `BNRItem` (to represent something like a backpack or a purse). In addition, an item will know which other item holds it.

In `BNRItem.h`, declare two instance variables and their accessors.

```

@interface BNRItem : NSObject
{
    NSString *_itemName;
    NSString *_serialNumber;
    int _valueInDollars;
    NSDate *_dateCreated;

    BNRItem *_containedItem;
    BNRItem *_container;
}
+ (instancetype)randomItem;
- (instancetype)initWithItemName:(NSString *)name
                           valueInDollars:(int)value
                            serialNumber:(NSString *)sNumber;
- (instancetype)initWithItemName:(NSString *)name;
- (void)setContainedItem:(BNRItem *)item;
- (BNRItem *)containedItem;
- (void)setContainer:(BNRItem *)item;
- (BNRItem *)container;

```

In `BNRItem.m`, implement the accessors.

```

- (void)setContainedItem:(BNRItem *)item
{
    _containedItem = item;

    // When given an item to contain, the contained
    // item will be given a pointer to its container
    item.container = self;
}

- (BNRItem *)containedItem
{
    return _containedItem;
}

- (void)setContainer:(BNRItem *)item
{
    _container = item;
}

- (BNRItem *)container
{
    return _container;
}

```

In `main.m`, remove the code that populates the array with random items. Then create two new items, add them to the array, and make them point at each other.

```

int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSMutableArray *items = [[NSMutableArray alloc] init];

        for (int i = 0; i < 10; i++) {
            BNRItem *item = [BNRItem randomItem];
            [items addObject:item];
        }

        BNRItem *backpack = [[BNRItem alloc] initWithItemName:@"Backpack"];
        [items addObject:backpack];

        BNRItem *calculator = [[BNRItem alloc] initWithItemName:@"Calculator"];
        [items addObject:calculator];

        backpack.containedItem = calculator;

        backpack = nil;
        calculator = nil;

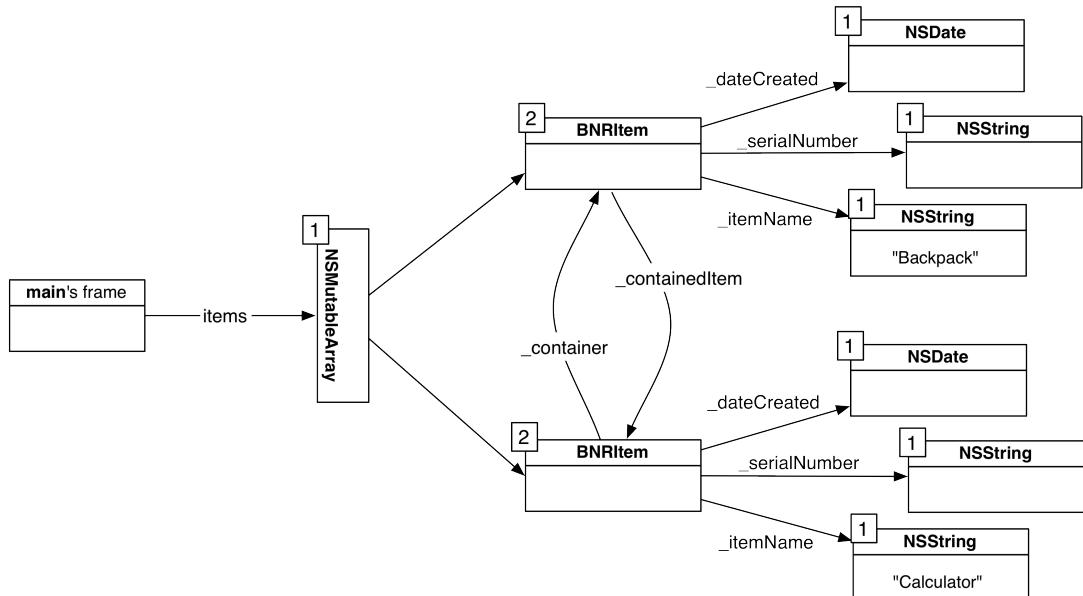
        for (BNRItem *item in items)
            NSLog(@"%@", item);

        NSLog(@"Setting items to nil...");
        items = nil;
    }
    return 0;
}

```

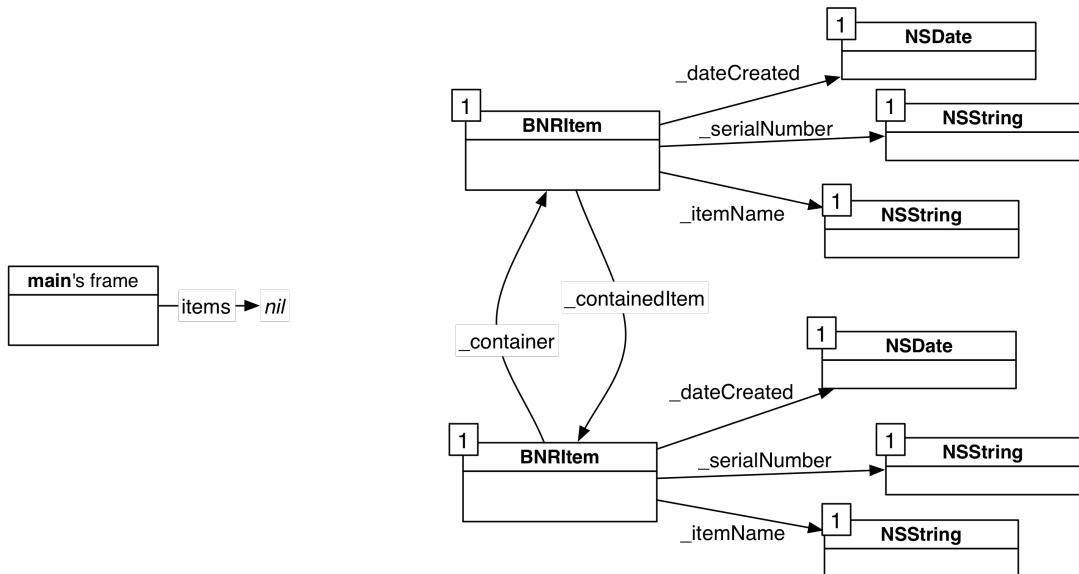
Here is what the application looks like now:

Figure 3.4 RandomItems with strong reference cycle



Build and run the application. This time, you will not see any messages reporting the destruction of the `BNRItem` objects. This is a strong reference cycle: the backpack and the calculator have strong references to one another, so there is no way to destroy these objects. Figure 3.5 shows the objects in the application that are still taking up memory once `items` has been set to `nil`.

Figure 3.5 Memory leak!



The two items cannot be accessed by any other part of the application (in this case, `main()`), yet they still exist, doing nothing useful. Moreover, because they cannot be destroyed, neither can the objects that their instance variables point to.

To fix this problem, one of the pointers between the items needs to be a weak reference. To decide which one should be weak, think of the objects in the cycle as being in a parent-child relationship. In this relationship, the parent can own its child, but a child should never own its parent. In our strong reference cycle, the backpack is the parent, and the calculator is the child. Thus, the backpack can keep its strong reference to the calculator (the `_containedItem` instance variable), but the calculator's reference to the backpack (the `_container` instance variable) should be weak.

To declare a variable as a weak reference, you use the `__weak` attribute. In `BNRItem.h`, change the `container` instance variable to be a weak reference.

```
__weak BNRItem * _container;
```

Build and run the application again. This time, the objects are destroyed properly.

Most strong reference cycles can be broken down into a parent-child relationship. A parent typically keeps a strong reference to its child, so if a child needs a pointer to its parent, that pointer must be a weak reference to avoid a strong reference cycle.

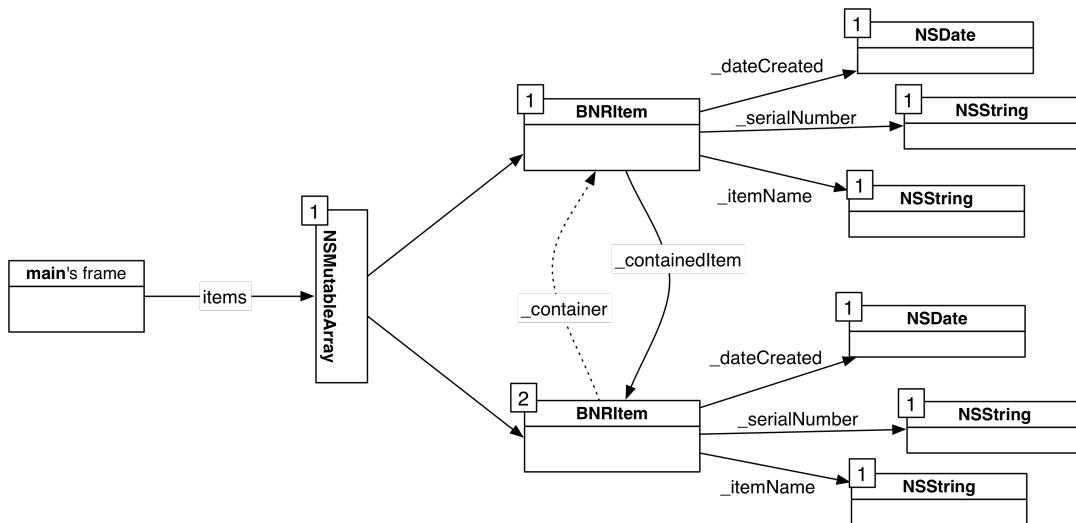
A child holding a strong reference to its *parent's* parent also causes a strong reference cycle. So the same rule applies in this situation: if a child needs a pointer to its parent's parent (or its parent's parent's parent, etc.), then that pointer must be a weak reference.

Apple's development tools includes a **Leaks** tool to help you find strong reference cycles. You will see how to use this tool in Chapter 14.

A weak reference knows when the object that it points to is destroyed and responds by setting itself to `nil`. Thus, if the backpack is destroyed, the calculator's `_container` instance variable will be automatically set to `nil`. This is convenient. If `_container` was not set to `nil`, then destroying the object would leave you with a dangling pointer, which could crash your application.

Here is the current diagram of `RandomItems`. Notice that the arrow representing the `container` pointer variable is now a dotted line. A dotted line denotes a weak reference. Strong references are always solid lines.

Figure 3.6 RandomItems with strong reference cycle avoided



Properties

Each time you have declared an instance variable in **BNRItem**, you have declared and implemented a pair of accessor methods. Now you are going to learn to use *properties*, a convenient alternative to writing out accessors methods that saves a lot of typing and makes your class files easier to read.

Declaring properties

A property declaration has the following form:

```
@property NSString *itemName;
```

By default, declaring a property will get you three things: an instance variable and two accessors for the instance variable. Take a look at Table 3.1, which shows a class not using properties on the left and the equivalent class with properties on the right.

Table 3.1 With and without properties

| | Without properties | With properties |
|------------|---|---|
| BNRThing.h | <pre>@interface BNRThing : NSObject { NSString *_name; } - (void)setName:(NSString *)n; - (NSString *)name; @end</pre> | <pre>@interface BNRThing : NSObject @property NSString *name; @end</pre> |
| BNRThing.m | <pre>@implementation BNRThing - (void)setName:(NSString *)n { _name = n; } - (NSString *)name { return _name; } @end</pre> | <pre>@implementation BNRThing @end</pre> |

These two classes in Table 3.1 are exactly the same: each has one instance variable for the name of the instance and a setter and getter for the name. On the left, you type out these declarations and instance variables yourself. On the right, you simply declare a property.

You are going to replace your instance variables and accessors in **BNRItem** with properties.

In `BNRItem.h`, delete the instance variable area and the accessor method declarations. Then, add the property declarations that replace them.

```
@interface BNRItem : NSObject
{
    NSString *_itemName;
    NSString *_serialNumber;
    int _valueInDollars;
    NSDate *_dateCreated;

    BNRItem *_containedItem;
    __weak BNRItem *_container;
}

@property BNRItem *containedItem;
@property BNRItem *container;

@property NSString *itemName;
@property NSString *serialNumber;
@property int valueInDollars;
@property NSDate *dateCreated;

+ (instancetype)randomItem;

- (instancetype)initWithItemName:(NSString *)name
                           valueInDollars:(int)value
                           serialNumber:(NSString *)sNumber;

- (instancetype)initWithItemName:(NSString *)name
                           itemName:(NSString *)str+
                           (NSString *)itemName;

- (void)setSerialNumber:(NSString *)str+
                           (NSString *)serialNumber;

- (void)setValueInDollars:(int)v+
                           (int)valueInDollars;

- (NSDate *)dateCreated;

- (void)setContainedItem:(BNRItem *)item;
                           (BNRItem *)containedItem;

- (void)setContainer:(BNRItem *)item;
                           (BNRItem *)container;

@end
```

Now, `BNRItem.h` is much easier to read:

```

@interface BNRItem : NSObject

+ (instancetype)randomItem;

- (instancetype)initWithItemName:(NSString *)name
                           valueInDollars:(int)value
                            serialNumber:(NSString *)sNumber;

- (instancetype)initWithItemName:(NSString *)name;

@property BNRItem *containedItem;
@property BNRItem *container;

@property NSString *itemName;
@property NSString *serialNumber;
@property int valueInDollars;
@property NSDate *dateCreated;

@end

```

Notice that the names of the properties are the names of the instance variables minus the underscore. The instance variable generated by a property, however, does have an underscore in its name.

Let's look at an example. When you declared the property named `itemName`, you got an instance variable named `_itemName`, a getter method named `itemName`, and a setter method named `setItemName:`. (Note that these declarations will not appear in your file; they are declared by the compiler behind the scenes.) Thus, the rest of the code in your application can work as before.

Declaring these properties also takes care of the implementations of the accessors. In `BNRItem.m`, delete the accessor implementations.

```

-(void)setItemName:(NSString *)str
{
    _itemName = str;
}
-(NSString *)itemName
{
    return _itemName;
}
-(void)setSerialNumber:(NSString *)str
{
    _serialNumber = str;
}
-(NSString *)serialNumber
{
    return _serialNumber;
}
-(void)setValueInDollars:(int)p
{
    _valueInDollars = p;
}
-(int)valueInDollars
{
    return _valueInDollars;
}

```

```
- (NSDate *)dateCreated
+ {
    return _dateCreated;
}
+ (void)setContainedItem:(BNRItem *)item
+ {
    _containedItem = item;
    // When given an item to contain, the contained
    // item will be given a pointer to its container
    item.container = self;
}
- (BNRItem *)containedItem
+ {
    return _containedItem;
}
+ (void)setContainer:(BNRItem *)item
+ {
    _container = item;
}
- (BNRItem *)container
+ {
    return _container;
}
```

You may be wondering about the implementation of `setContainedItem`: that you just deleted. This setter did more than just set the `_containedItem` instance variable. It also set the `_container` instance variable of the passed-in item. To replicate this functionality, you will shortly write a custom setter for the `containedItem` property. But first, let's discuss some property basics.

Property attributes

A property has a number of attributes that allow you to modify the behavior of the accessor methods and the instance variable it creates. The attributes are declared in parentheses after the `@property` directive. Here is an example:

```
@property (nonatomic, readwrite, strong) NSString *itemName;
```

Each attribute has a set of possible values, one of which is the default and does not have to be explicitly declared.

Multi-threading attribute

The `multi-threading attribute` of a property has two values: `nonatomic` or `atomic`. (Multi-threading is outside the scope of this book, but you still need to know the values for this attribute.) Most iOS programmers typically use `nonatomic`. We do at Big Nerd Ranch, and so does Apple. In this book, you will use `nonatomic` for all properties.

Unfortunately, the default value for this attribute is `atomic`, so you have to specify that you want your properties to be `nonatomic`.

In `BNRItem.h`, change all of your properties to be `nonatomic`.

```

@interface BNRItem : NSObject

+ (instancetype)randomItem;

- (instancetype)initWithItemName:(NSString *)name
    valueInDollars:(int)value
    serialNumber:(NSString *)sNumber;
- (instancetype)initWithItemName:(NSString *)name;

@property (nonatomic) BNRItem *containedItem;
@property (nonatomic) BNRItem *container;

@property (nonatomic) NSString *itemName;
@property (nonatomic) NSString *serialNumber;
@property (nonatomic) int valueInDollars;
@property (nonatomic) NSDate *dateCreated;
@end

```

Read/write attribute

The **read/write attribute**'s value, **readonly** or **readwrite**, tells the compiler whether to implement a setter method for the property. A **readwrite** property implements both a setter and getter. A **readonly** property just implements a getter. The default option for this attribute is **readwrite**. This is what you want for all of **BNRItem**'s properties except **dateCreated**, which should be **readonly**.

In **BNRItem.h**, declare **dateCreated** as a **readonly** property so that no setter method is generated for this instance variable.

```
@property (nonatomic, readonly) NSDate *dateCreated;
```

Memory management attribute

The **memory management attribute**'s values are **strong**, **weak**, **copy**, and **unsafe_unretained**. This attribute describes the type of reference that the object with the instance variable has to the object that the variable is pointing to.

For properties that do not point to objects (like the **int valueInDollars**), there is no need for memory management, and **the only option is unsafe_unretained**. This is direct assignment. You may also see the **value assign** in some places, which was the term used before ARC.

(The “unsafe” part of **unsafe_unretained** is misleading when dealing with non-object properties. It comes from contrasting unsafe unretained references with weak references. Unlike a weak reference, an unsafe unretained reference is not automatically set to **nil** when the object that it points to is destroyed. This is unsafe because you could end up with dangling pointers. However, the issue of dangling pointers is irrelevant when dealing with non-object properties.)

As the only option, **unsafe_unretained** is also the default value for non-object properties, so you can leave the **valueInDollars** property as is.

For properties that manage a pointer to an Objective-C object, all four options are possible. **The default is strong**. However, Objective-C programmers tend to explicitly declare this attribute. (One reason is that the default value has changed in the last few years, and that could happen again.)

In **BNRItem.m**, set the memory management attribute as **strong** for the **containedItem** and **dateCreated** properties and **weak** for the **container** property.

```
@property (nonatomic, strong) BNRItem *containedItem;
@property (nonatomic, weak) BNRItem *container;

@property (nonatomic) NSString *itemName;
@property (nonatomic) NSString *serialNumber;
@property (nonatomic) int valueInDollars;
@property (nonatomic, readonly, strong) NSDate *dateCreated;
```

Setting the container property to weak prevents the strong reference cycle that you caused and fixed earlier.

What about the itemName and serialNumber properties? These point to instances of **NSString**. When a property points to an instance of a class that has a mutable subclass (like **NSString/NSMutableString** or **NSArray/NSMutableArray**), you should set its memory management attribute to **copy**.

In BNRItem.m, set the memory management attribute for itemName and serialNumber as copy.

```
@property (nonatomic, strong) BNRItem *containedItem;
@property (nonatomic, weak) BNRItem *container;

@property (nonatomic, copy) NSString *itemName;
@property (nonatomic, copy) NSString *serialNumber;
@property (nonatomic) int valueInDollars;
@property (nonatomic, readonly, strong) NSDate *dateCreated;
```

Here is what the generated setter for itemName will look like:

```
- (void)setitemName:(NSString *)itemName
{
    _itemName = [itemName copy];
}
```

Instead of setting _itemName to point to the incoming object, this setter sends the message **copy** to that object. The **copy** method returns an immutable **NSString** object that has the same values as the original string, and _itemName is set to point at the new string.

Why is it safer to do this for **NSString**? It is safer to make a copy of the object rather than risk pointing to a possibly mutable object that could have other owners who might change the object without your knowledge.

For instance, imagine if an item was initialized so that its itemName pointed to an **NSMutableString**.

```
NSMutableString *mutableString = [[NSMutableString alloc] init];
BNRItem *item = [[BNRItem alloc] initWithitemName:mutableString
                                         valueInDollars:5
                                         serialNumber:@"4F2W7"]];
```

This code is valid because an instance of **NSMutableString** is also an instance of its superclass, **NSString**. The problem is that the string pointed to by mutableString can be changed without the knowledge of the item that also points to it.

In your application, you are not going to change this string unless you mean to. However, when you write classes for others to use, you cannot control how they will be used, and you have to program defensively.

In this case, the defense is to declare `itemName` with the `copy` attribute.

In terms of ownership, `copy` gives you a strong reference to the object pointed to. The original string is not modified in any way: it does not gain or lose an owner, and none of its data changes.

While it is wise to make a copy of a mutable object, it is wasteful to make a copy of an immutable object. An immutable object cannot be changed, so the kind of problem described above cannot occur. To prevent needless copying, immutable classes implement `copy` to quietly return a pointer to the original and immutable object.

Custom accessors with properties

By default, the accessors that a property implements are very simple and look like this:

```
- (void)setContainedItem:(BNRItem *)item
{
    _containedItem = item;
}
- (BNRItem *)containedItem
{
    return _containedItem;
}
```

For most properties, this is exactly what you want. However, for the `containedItem` property, the default setter method is not sufficient. The implementation of `setContainedItem:` needs an extra step: it should also set the `container` property of the item being contained.

You can replace the default setter by implementing the setter yourself in the implementation file.

In `BNRItem.m`, add back an implementation for `setContainedItem:`.

```
- (void)setContainedItem:(BNRItem *)containedItem
{
    _containedItem = containedItem;
    self.containedItem.container = self;
}
```

When the compiler sees that you have implemented `setContainedItem:`, it will not create a default setter for `containedItem`. It will still create the getter method, `containedItem`.

Note that if you implement both a custom setter and a custom getter (or just a custom getter on a read-only property), then the compiler will not create an instance variable for your property. If you need one, you must declare it yourself.

Note the moral: sometimes the default accessors do not do what you need, and you will need to implement them yourself.

Now you can build and run the application. The leaner `BNRItem` works in the exact same way.

For the More Curious: Property Synthesis

When explaining properties in this chapter, we noted that a property automatically generates the implementation for the accessor methods and it declares and creates an instance variable. While this is true, we omitted the fact that this behavior is only the default and you have other options.

Declaring a property in a class interface only declares the accessor methods in a class interface. In order for a property to automatically generate an instance variable and the implementations for its methods, it must be *synthesized*, either implicitly or explicitly. Properties are implicitly synthesized by default. A property is explicitly synthesized by using the `@synthesize` directive in an implementation file:

```
@implementation Person  
  
// Generates the code for -setAge: and -age,  
// and creates the instance variable _age  
@synthesize age = _age;  
  
// Other methods go here  
  
@end
```

This is how properties are automatically synthesized. The first attribute (`age`) says “create methods named `age` and `setAge:`,” and the second attribute (`_age`) says “the instance variable that backs these methods should be `_age`.”

You can optionally leave off the variable name, which creates a backing variable with the same name as the accessors.

```
@synthesize age;  
// Is the same as:  
@synthesize age = age;
```

There are cases where you do not want an instance variable to back a property and therefore do not want a property to automatically generate the accessor method implementations. Consider a **Person** class with three properties, `spouse`, `lastName`, and `lastNameOfSpouse`:

```
@interface Person : NSObject  
@property (nonatomic, strong) Person *spouse;  
@property (nonatomic, copy) NSString *lastName;  
@property (nonatomic, copy) NSString *lastNameOfSpouse;  
@end
```

In this somewhat contrived example, it makes sense for both the `spouse` and `lastName` properties to be backed by an instance variable. After all, this is information that each **Person** needs to hang onto. However, it does not make sense to hold onto the last name of the spouse as an instance variable. A **Person** can just ask their spouse for their `lastName`, so storing this information in both **Person** instances is redundant and therefore prone to error. Instead, the **Person** class would implement the getter and setter for the `lastNameOfSpouse` property like so:

```
@implementation Person  
- (void)setLastNameOfSpouse:(NSString *)lastNameOfSpouse  
{  
    self.spouse.lastName = lastNameOfSpouse;  
}  
  
- (NSString *)lastNameOfSpouse  
{  
    return self.spouse.lastName;  
}  
@end
```

In this case, because you have implemented both accessors, the compiler will not automatically synthesize an instance variable for `lastNameOfSpouse`. Which is exactly what you would hope for.

For the More Curious: Autorelease Pool and ARC History

Before automatic reference counting (ARC) was added to Objective-C, we had *manual reference counting*. With manual reference counting, ownership changes only happened when you sent an explicit message to an object.

```
[anObject release]; // anObject loses an owner
[anObject retain]; // anObject gains an owner
```

This was a bummer: Forgetting to send `release` to an object before setting a pointer to point at something else would create a memory leak. Sending `release` to an object if you had not previously sent `retain` to the object was a premature deallocation. A lot of time was spent debugging these problems, which could become very complex in large projects.

During the dark days of manual reference counting, Apple was contributing to an open source project known as the Clang static analyzer and integrating it into Xcode. You will see more about the static analyzer in Chapter 14, but the basic gist is that it could analyze code and tell you if you were doing something silly. Two of the silly things it could detect were memory leaks and premature deallocations. Smart programmers would run their code through the static analyzer to detect these problems and then write the necessary code to fix them.

Eventually, the static analyzer got so good that Apple thought, “Why not just let the static analyzer insert all of the retain and release messages?” Thus, ARC was born. People rejoiced in the streets, and most memory management problems became a thing of the past.

Another thing programmers had to understand in the days of manual reference counting was the *autorelease pool*. When an object was sent the message `autorelease`, the autorelease pool would take ownership of an object temporarily so that it could be returned from the method that created it without burdening the creator or the receiver with ownership responsibilities. This was crucial for convenience methods that created a new instance of some object and returned it:

```
+ (BNRItem *)someItem
{
    BNRItem *item = [[[BNRItem alloc] init] autorelease];
    return item;
}
```

Because you had to send the `release` message to an object to relinquish ownership, the caller of this method had to understand its ownership responsibilities. But it was easy to get confused.

```
BNRItem *item = [BNRItem someItem]; // I guess I own this now?
NSString *string = [item itemName]; // Well, if I own that, do I own this?
```

Thus, objects created by methods other than `alloc` and `copy` would be sent `autorelease` before being returned, and the receiver of the object would take ownership as needed or just let it be destroyed when the autorelease pool was drained.

With ARC, this is done automatically (and sometimes optimized out completely). An autorelease pool is created by the `@autoreleasepool` directive followed by curly braces. Inside those curly braces, any newly instantiated object returned from a method that does not have `alloc` or `copy` in its name is placed in that autorelease pool. When the curly brace closes, any object in the pool loses an owner.

```
@autoreleasepool {
    // Get a BNRItem back from a method that created it,
    // method does not say alloc/copy
    BNRItem *item = [BNRItem someItem];
} // Pool is drained, item loses an owner and is destroyed
```

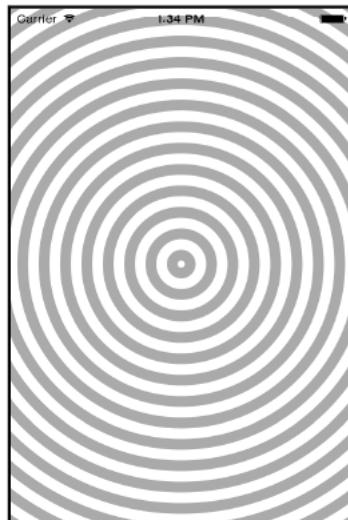
iOS applications automatically create an autorelease pool for you, and you really do not have to concern yourself with it. But isn't it nice to know what that `@autoreleasepool` is for?

4

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.

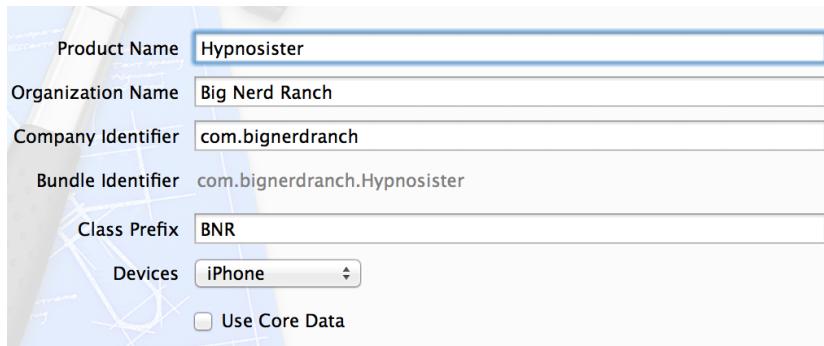
Figure 4.1 Hypnosister



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

Enter **Hypnosister** for the product name and **BNR** for the class prefix, as shown in Figure 4.2. Make sure **iPhone** is selected from the **Devices** drop down and make sure the **Use Core Data** box is unchecked.

Figure 4.2 Configuring Hypnosister



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.

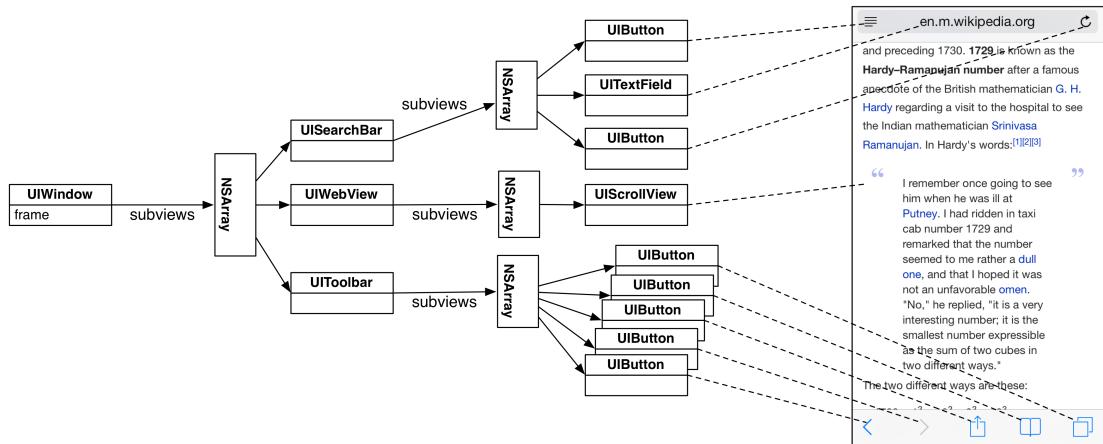
In Chapter 1, you created four views for the Quiz app: two instances of **UIButton** and two instances of **UILabel**. You created and configured these views in Interface Builder, but you can also create views programmatically. In Hypnosister, you will create views programmatically.

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 4.3 An example view hierarchy and the interface that it creates

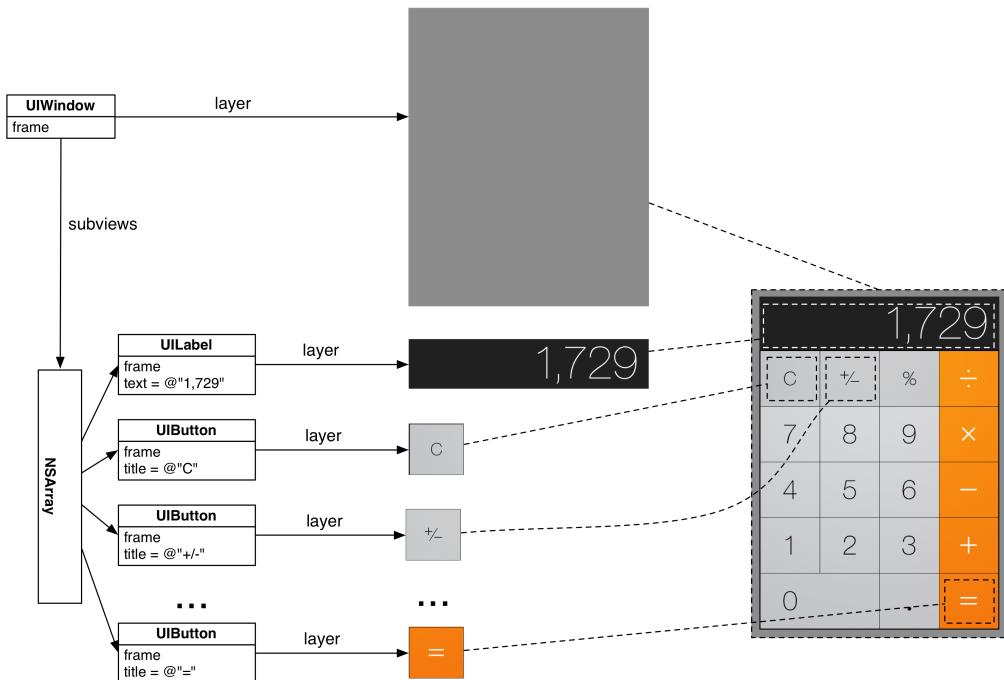


Once the view hierarchy has been created, it will be drawn to the screen. This process can be broken into two steps:

- 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 4.4 shows another example view hierarchy and the two drawing steps.

Figure 4.4 Views render themselves and then are composited together



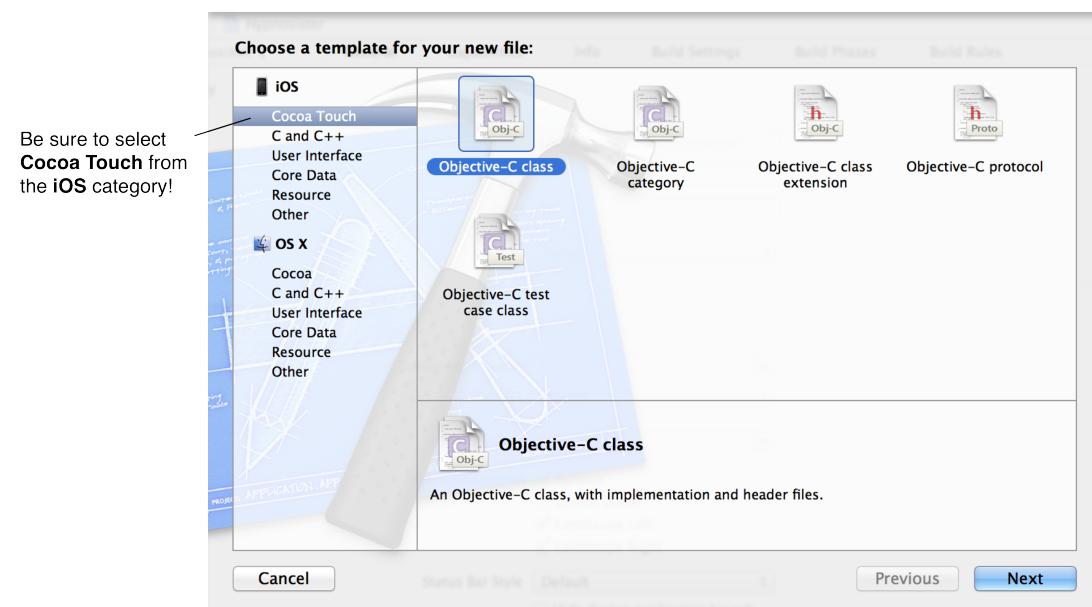
Classes like **UIButton** and **UILabel** already know how to render themselves to their layers. For instance, in Quiz, you created instances of **UILabel** and told them what text to display, but you did not have to tell them how to draw text. Apple's developers took care of that.

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**

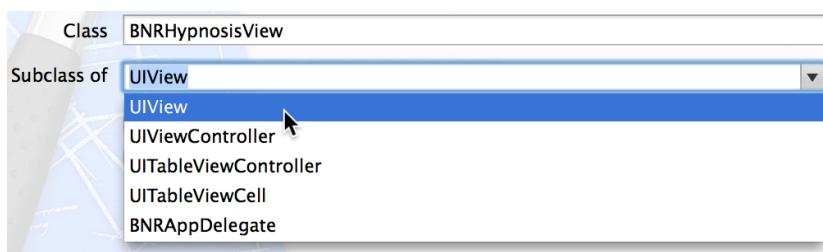
To create a **UIView** subclass, select File → New → File... (or press Command-N). From the iOS section, select Cocoa Touch and then choose Objective-C class (Figure 4.5).

Figure 4.5 Creating a new class



Click Next. On the next pane, name the class **BNRHypnosisView** and select **UIView** as the superclass.

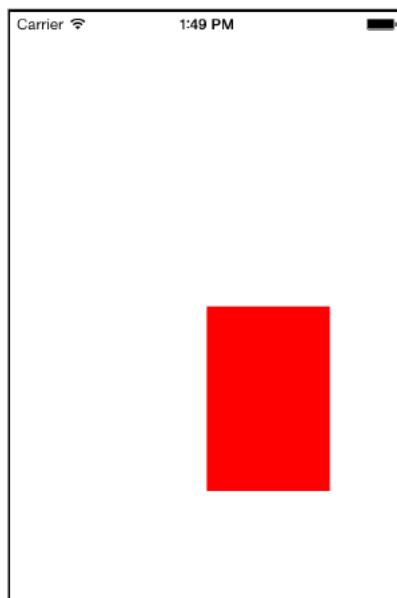
Figure 4.6 Choosing **UIView** as the superclass



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

Before writing the concentric circle drawing code for **BNRHypnosisView**, 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 **BNRHypnosisView** view will not draw concentric circles. Instead, it will draw a rectangle with a red background.

Figure 4.7 Initial version of **BNRHypnosisView**



Views and frames

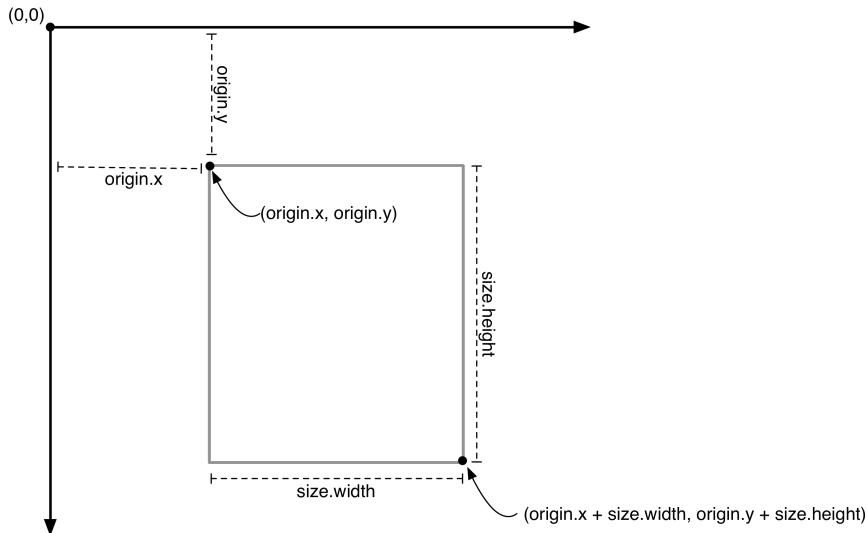
Open **BNRHypnosisView.m**. The **UIView** subclass template has provided two method stubs for you. The first is **initWithFrame:**, the designated initializer for **UIView**. This method takes one argument, a **CGRect**, that will become the view's **frame**, a property on **UIView**.

```
@property (nonatomic) CGRect frame;
```

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 C structure of type **CGPoint** and contains two float members: **x** and **y**. The **size** is a C structure of type **CGSize** and has two float members: **width** and **height** (Figure 4.8).

Figure 4.8 CGRect



Open `BNRAppDelegate.m`. At the top of this file, import the header file for `BNRHypnosisView`.

```
#import "BNRAppDelegate.h"
#import "BNRHypnosisView.h"
```

```
@implementation BNRAppDelegate
```

In `BNRAppDelegate.m`, 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 `BNRHypnosisView`. Next, create an instance of `BNRHypnosisView` and set its `backgroundColor` property to red. Finally, add the `BNRHypnosisView` as a subview of the window to make it part of the view hierarchy.

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    CGRect firstFrame = CGRectMake(160, 240, 100, 150);

    BNRHypnosisView *firstView = [[BNRHypnosisView alloc] initWithFrame:firstFrame];
    firstView.backgroundColor = [UIColor redColor];

    [self.window addSubview:firstView];

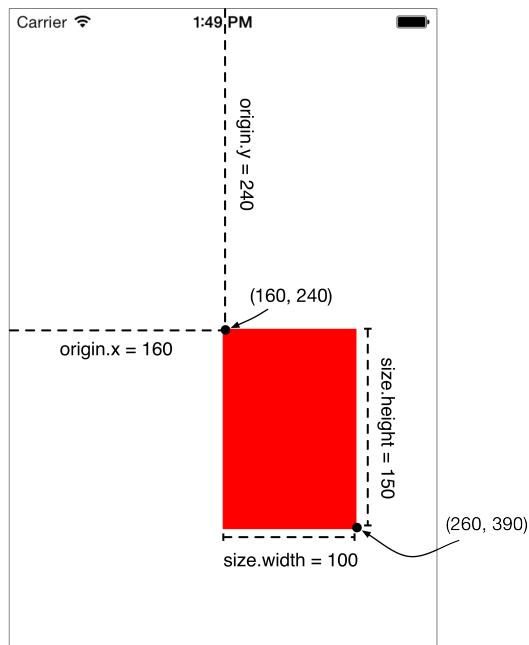
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

A structure is not an Objective-C object, so you cannot send messages to a `CGRect`. To create one, you used `CGRectMake()` and pass in the values for the `origin.x`, `origin.y`, `size.width` and `size.height`. A `CGRect` is small compared to most objects, so instead of passing a pointer to it, you just pass the entire structure. Thus, `initWithFrame:` expects a `CGRect`, not a `CGRect *`.

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

Build and run the application. The red rectangle is the instance of `BNRHypnosisView`. Because the `BNRHypnosisView`'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 4.9 Hypnosister with one `BNRHypnosisView`

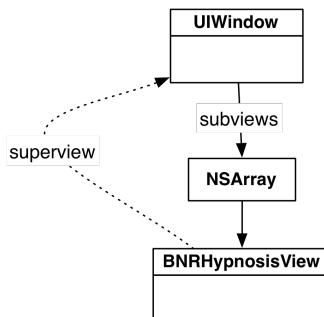


Note these values are in points, not pixels. If the values were in pixels, then they would not be consistent across displays of different resolutions (i.e., Retina vs. non-Retina). A single point is a relative unit of a measure; it will be a different number of pixels depending on how many pixels there 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 6.

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

Figure 4.10 **UIWindow** has one subview – a **BNRHypnosisView**

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 **BNRHypnosisView**'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 **BNRAppDelegate.m**, create another instance of **BNRHypnosisView** with a different frame and background color.

```

...
[self.window addSubview:firstView];

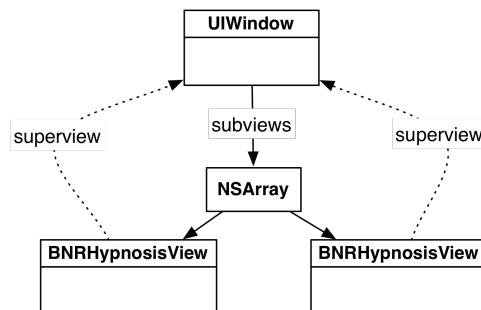
CGRect secondFrame = CGRectMake(20, 30, 50, 50);

BNRHypnosisView *secondView = [[BNRHypnosisView alloc] initWithFrame:secondFrame];
secondView.backgroundColor = [UIColor blueColor];

[self.window addSubview:secondView];

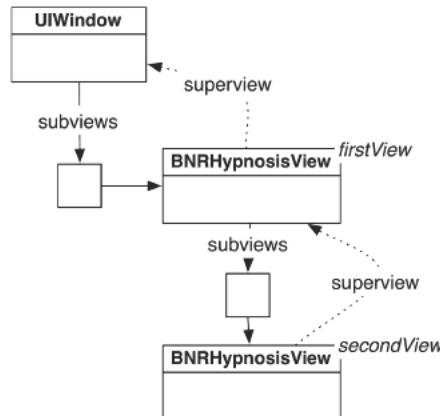
self.window.backgroundColor = [UIColor whiteColor];
...
  
```

Build and run again. In addition to the red rectangle, you will see a blue square near the top lefthand corner of the window. Figure 4.11 shows the updated view hierarchy.

Figure 4.11 **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 **BNRHypnosisView** as a subview of the first instance of **BNRHypnosisView** instead of the window:

Figure 4.12 One **BNRHypnosisView** as a subview of the other



In **BNRAppDelegate.m**, make this change.

```

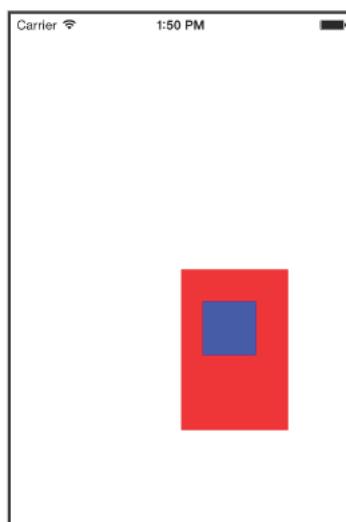
...
BNRHypnosisView *secondView = [[BNRHypnosisView alloc] initWithFrame:secondFrame];
secondView.backgroundColor = [UIColor blueColor];

[self.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 4.13 Hypnosister with new hierarchy



(If the blue instance of `BNRHypnosisView` 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 `BNRHypnosisView` before continuing.

```
...
[self.window addSubview:firstView];
CGRect secondFrame = CGRectMake(20, 30, 50, 50);
BNRHypnosisView *secondView = [[BNRHypnosisView alloc] initWithFrame:secondFrame];
secondView.backgroundColor = [UIColor blueColor];
[view addSubview:secondView];
self.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 backgroundColors. In this section, you will write the custom drawing code for `BNRHypnosisView` 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.

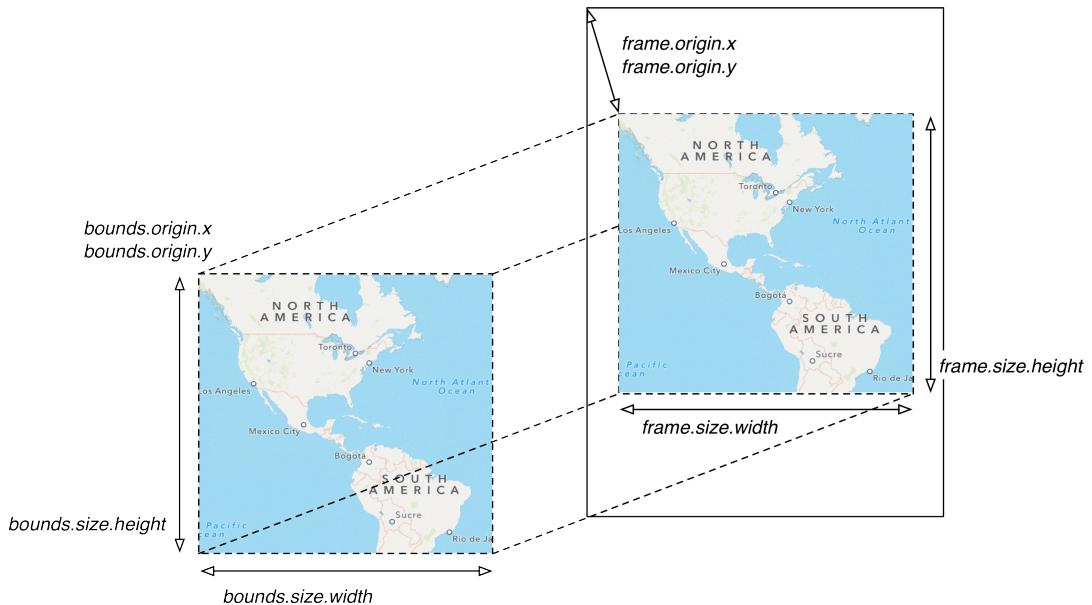
The first thing that you typically do when overriding `drawRect:` is get the bounds rectangle of the view. The `bounds` property, inherited from `UIView`, is the rectangle that defines the area where the view will draw itself.

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 4.14).

Figure 4.14 bounds vs. frame



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

In `BNRAppDelegate.m`, update `firstView`'s frame to match the `bounds` of the window.

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch
    CGRect firstFrame = CGRectMake(160, 240, 100, 150);
    CGRect firstFrame = self.window.bounds;

    BNRHypnosisView *firstView = [[BNRHypnosisView alloc] initWithFrame:firstFrame];
    [self.window addSubview:firstView];

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

Build and run the application, and you will be greeted with a full-sized view with a red 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 `BNRHypnosisView.m`, add code to `drawRect:` that finds the center point of bounds.

```
- (void)drawRect:(CGRect)rect
{
    CGRect bounds = self.bounds;

    // Figure out the center of the bounds rectangle
    CGPoint center;
    center.x = bounds.origin.x + bounds.size.width / 2.0;
    center.y = bounds.origin.y + bounds.size.height / 2.0;
}
```

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.)

```
- (void)drawRect:(CGRect)rect
{
    CGRect bounds = self.bounds;

    // Figure out the center of the bounds rectangle
    CGPoint center;
    center.x = bounds.origin.x + bounds.size.width / 2.0;
    center.y = bounds.origin.y + bounds.size.height / 2.0;

    // The circle will be the largest that will fit in the view
    float 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`.

```
- (void)drawRect:(CGRect)rect
{
    ...

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

    UIBezierPath *path = [[UIBezierPath alloc] init];

}
```

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.

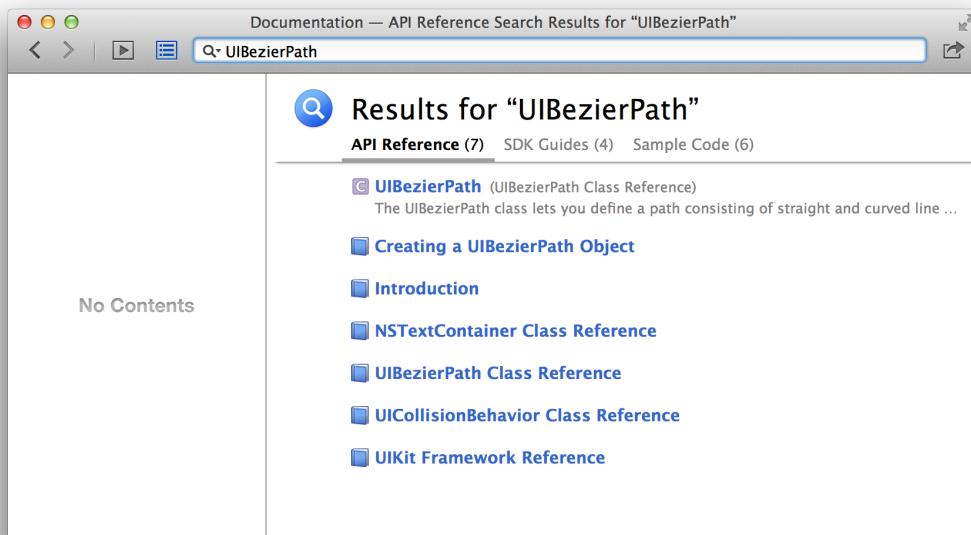
Using the developer documentation

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**.

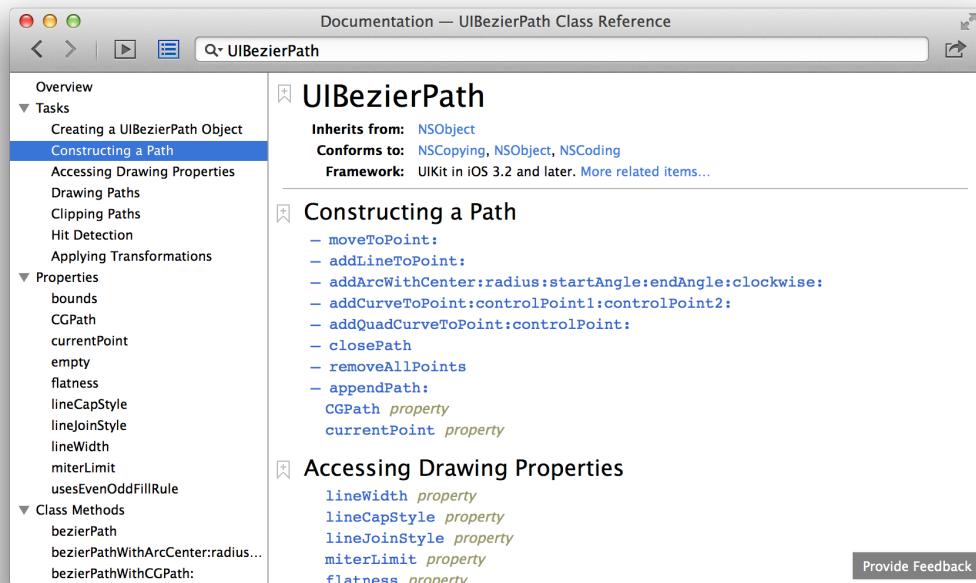
Figure 4.15 Documentation results



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 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 4.16 Methods for constructing a path



A likely candidate for a circular path is

`addArcWithCenter: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 `BNRHypnosisView.m`, send a message to the `UIBezierPath` that defines its path.

```
- (void)drawRect:(CGRect)rect
{
    CGRect bounds = self.bounds;

    // Figure out the center of the bounds rectangle
    CGPoint center;
    center.x = bounds.origin.x + bounds.size.width / 2.0;
    center.y = bounds.origin.y + bounds.size.height / 2.0;

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

    UIBezierPath *path = [[UIBezierPath alloc] init];

    // 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:0.0
                   endAngle:M_PI * 2.0
                  clockwise:YES];
}
```

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 **BNRHypnosisView.m**, send a message to the **UIBezierPath** that tells it to draw.

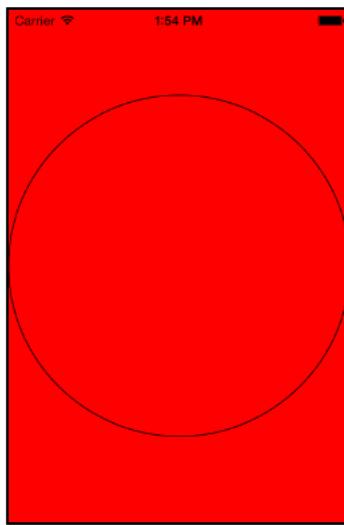
```
- (void)drawRect:(CGRect)rect
{
    ...
    UIBezierPath *path = [[UIBezierPath alloc] init];

    // 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:0.0
                   endAngle:M_PI * 2.0
                  clockwise:YES];

    // 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 4.17 **BNRHypnosisView** 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 light gray.

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 `BNRHypnosisView.m`, make the width of the line 10 points.

```
- (void)drawRect:(CGRect)rect
{
    ...
    // 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:0.0
        endAngle:M_PI * 2.0
        clockwise:YES];

    // 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 **lightGrayColor**.

Now you have the information you need. In **BNRHypnosisView.m**, add code to create a light gray **UIColor** instance and send it the **setStroke** message so that when the path is drawn, it will be drawn in light gray.

```
- (void)drawRect:(CGRect)rect
{
    ...
    // Configure line width to 10 points
    path.lineWidth = 10;

    // Configure the drawing color to light gray
    [[UIColor lightGrayColor] setStroke];

    // Draw the line!
    [path stroke];
}
```

Build and run the application, and you will see a wider, light gray 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 **BNRAppDelegate.m**, remove the code that sets the background color of the view.

```
BNRHypnosisView *firstView = [[BNRHypnosisView alloc] initWithFrame:firstFrame];
firstView.backgroundColor = [UIColor redColor];

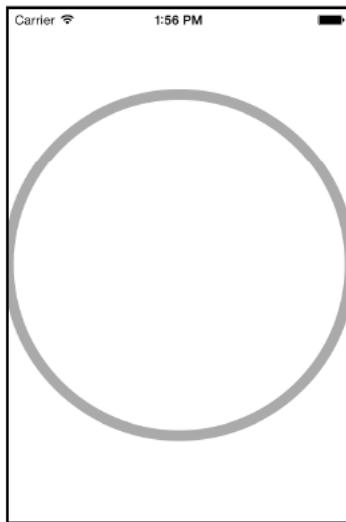
[self.window addSubview:view];
```

Then, in **BNRHypnosisView.m**, add code to **initWithFrame:** to set the background color of every **BNRHypnosisView** to clear.

```
- (instancetype)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        // All BNRHypnosisViews start with a clear background color
        self.backgroundColor = [UIColor clearColor];
    }
    return self;
}
```

Build and run the application. Figure 4.18 shows the clear background and the resulting circle.

Figure 4.18 **BNRHypnosisView** with clear background



Drawing concentric circles

There are two approaches you can take to draw multiple concentric circles inside the **BNRHypnosisView**. 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. It is slightly more efficient to use one instance, so you are going to do that.

To fill the screen with concentric circles, you need to determine the radius of the outermost circle. You will start drawing a circle with this radius and then draw circles with a decreasing radius for as long as the radius remains positive.

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 light gray in the corners.

In `BNRHypnosisView.m`, replace the code that draws one circle with code that draws concentric circles.

```
- (void)drawRect:(CGRect)rect
{
    CGRect bounds = self.bounds;

    // Figure out the center of the bounds rectangle
    CGPoint center;
    center.x = bounds.origin.x + bounds.size.width / 2.0;
    center.y = bounds.origin.y + bounds.size.height / 2.0;

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

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

    UIBezierPath *path = [[UIBezierPath alloc] init];

    // 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:0.0
        endAngle:M_PI * 2.0
        clockwise:YES];

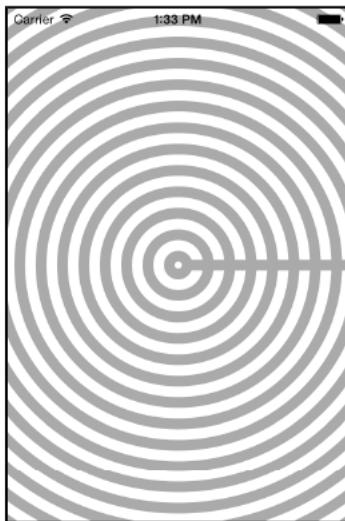
    for (float currentRadius = maxRadius; currentRadius > 0; currentRadius -= 20) {

        [path addArcWithCenter:center
            radius:currentRadius // Note this is currentRadius!
            startAngle:0.0
            endAngle:M_PI * 2.0
            clockwise:YES];
    }

    // Configure line width to 10 points
    path.lineWidth = 10.0;

    // Draw the line!
    [path stroke];
}
```

Build and run the application. It is not quite what you were expecting; it looks more like crop circles than concentric circles (Figure 4.19).

Figure 4.19 **BNRHypnosisView** drawing crop circles

The problem is that your single **UIBezierPath** object is connecting the sub-paths (the individual circles) to form the complete path. Think of a **UIBezierPath** object as a pencil on a piece of paper – when you go to draw another circle, the pencil stays on the piece of paper. You need to lift the pencil off the piece of paper before drawing a new circle.

In the for loop in **BNRHypnosisView**'s **drawRect:**, pick up the pencil and move it to the correct spot before drawing each circle.

```
- (void)drawRect:(CGRect)rect
{
    CGRect bounds = self.bounds;

    // Figure out the center of the bounds rectangle
    CGPoint center;
    center.x = bounds.origin.x + bounds.size.width / 2.0;
    center.y = bounds.origin.y + bounds.size.height / 2.0;

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

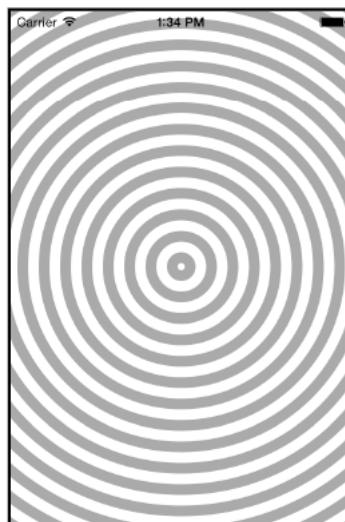
    UIBezierPath *path = [[UIBezierPath alloc] init];
    for (float currentRadius = maxRadius; currentRadius > 0; currentRadius -= 20) {

        [path moveToPoint:CGPointMake(center.x + currentRadius, center.y)];
        [path addArcWithCenter:center
                           radius:currentRadius // note this is currentRadius!
                         startAngle:0.0
                           endAngle:M_PI * 2.0
                          clockwise:YES];
    }
}
```

```
// Configure line width to 10 points  
path.lineWidth = 10.0;  
  
// Draw the line!  
[path stroke];  
}
```

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

Figure 4.20 **BNRHypnosisView** 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.

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 4.21.

Figure 4.21 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:

```
UIImage *logoImage = [UIImage imageNamed:@"logo.png"];
```

In your `drawRect:` method, compositing it onto your view is just one more:

```
[logoImage drawInRect:someRect];
```

For the More Curious: Core Graphics

In general, the `drawRect:` method uses `UIImage`, `UIBezierPath`, and `NSString` instances to draw images, shapes, and text, respectively. Each of these classes implements at least one method that, when executed in `drawRect:`, draws pixels to the layer of the view that was sent `drawRect:`.

These classes make iOS drawing look simple and convenient. However, there is a lot going on underneath the hood.

Drawing images in iOS – whether it is an image you will save as a JPEG or PDF or a layer that represents a `UIView` – is the responsibility of the *Core Graphics* framework. The classes that you used to perform drawing in this chapter, like `UIBezierPath`, wrap Core Graphics code into their methods to ease drawing for the programmer. To truly understand how these classes work and how images are created, you should understand how *Core Graphics* does its job.

Core Graphics is a 2D drawing API written in C. As such, there are no Objective-C objects or methods, but instead C structures and C functions that mimic object-oriented behavior. The most important Core Graphics “object” is the **graphics context**, which really holds two things: **the state of drawing**, like the current color of the pen and its line thickness, and **the memory that is being drawn upon**. A graphics context is represented by “instances” of CGContextRef.

Right before **drawRect:** is sent to an instance of **UIView**, the system creates a CGContextRef for that view’s layer. The layer has the same bounds as the view and some default values for its drawing state. As drawing operations are sent to the context, the pixels in the layer are changed. After **drawRect:** completes, the system grabs the layer and composites it to the screen.

The drawing classes you used in this chapter all know how to call Core Graphics functions that change the drawing state and issue drawing operations on the appropriate CGContextRef. For example, sending **setStroke** to an instance of **UIColor** will call functions that change the drawing state of the current context. So, these two chunks of code are equivalent:

```
[[UIColor colorWithRed:1.0 green:0.0 blue:1.0 alpha:1.0] setStroke];
```

```
UIBezierPath *path = [UIBezierPath bezierPath];
[path moveToPoint:a];
[path addLineToPoint:b];

[path stroke];
```

Is equivalent to these lines:

```
CGContextSetRGBStrokeColor(currentContext, 1, 0, 0, 1);

CGMutablePathRef path = CGPathCreateMutable();
CGPathMoveToPoint(path, NULL, a.x, a.y);
CGPathAddLineToPoint(path, NULL, b.x, b.y);
CGContextAddPath(currentContext, path);

CGContextStrokePath(currentContext);
CGPathRelease(path);
```

The Core Graphics functions that operate on the context, like **CGContextSetRGBStrokeColor**, take a pointer to context that they will modify as their first argument. You can grab a pointer to the current context in **drawRect:** by calling the function **UIGraphicsGetCurrentContext**. The current context is an application-wide pointer that is set to point to the context created for a view right before that view is sent **drawRect:**.

```
- (void)drawRect:(CGRect)rect
{
    CGContextRef currentContext = UIGraphicsGetCurrentContext();

    CGContextSetRGBStrokeColor(currentContext, 1, 0, 0, 1);

    CGMutablePathRef path = CGPathCreateMutable();
    CGPathMoveToPoint(path, NULL, a.x, a.y);
    CGPathAddLineToPoint(path, NULL, b.x, b.y);
    CGContextAddPath(currentContext, path);

    CGContextStrokePath(currentContext);
    CGPathRelease(path);

    CGContextSetStrokeColorWithColor(currentContext, color);
}
```

Anything you can do with **UIBezierPath** and **UIColor** can be done directly in Core Graphics. In fact, there are C structures that have the same behavior as these classes (**CGMutablePathRef** and **CGColorRef**). However, it is usually easier to work with the Objective-C counterparts.

Also, there are some things you just cannot do yet without dropping down to Core Graphics, like drawing gradients. However, because you remembered that types from frameworks all have the same prefix, you can search the documentation for types beginning in CG to find out what is available to you.

You might be wondering why many of the Core Graphics types have a Ref after them. Every Core Graphics type is a structure, but some mimic the behavior of objects by being allocated on the heap. Therefore, when you create one of these Core Graphics “objects”, you are returned a pointer to their address in memory.

Each Core Graphics structure that is allocated in this way has a type definition that incorporates the asterisk (*) into the type itself. For example, there exists a structure **CGColor** (that you never use) and a type definition **CGColorRef** that means **CGColor *** (that you always use). This convention makes it easy for a programmer to glance at code and determine whether or not the variable is a C structure masquerading as an object or an Objective-C object that you can send messages to.

Another point of confusion for programmers in Core Graphics is that some types do not have a Ref or an asterisk, like **CGRect** and **CGPoint**. These types are small data structures that can live on the stack, so there is no need to pass a pointer to them.

However, some Core Graphics types are much more involved than simply holding onto a few floats – they actually have pointers to other Core Graphics objects. These “objects” will take strong ownership of the objects they point to, but ARC will not track this ownership. Instead, you must manually release ownership of these types of objects when you are done with them. The rule is: **if you create a Core Graphics object with a function that has the word Create or Copy in it, you must call the matching Release function and pass a pointer to the object as the first argument.**

One final note about Core Graphics: It exists on the Mac, too. You can write code, as done in the open source core-plot framework, that will work on both iOS and OS X.

Gold Challenge: Shadows and Gradients

At this time, adding drop shadows and drawing with gradients can only be done using Core Graphics.

To create a drop shadow, you install a shadow on the graphics context. After that, anything opaque that you draw will have a drop shadow. The shadow has an offset (which is expressed with an **CGSize**), and a blur in points. Here is the declaration of the method used to install the shadow on the graphics context:

```
void CGContextSetShadow (
    CGContextRef context,
    CGSize offset,
    CGFloat blur);
```

(There is a version that takes a color, but you almost always want a dark shadow.)

There is no unset shadow function. Thus, you will need to save the graphics state before setting the shadow and then restore it after setting the shadow. It looks something like this:

```

CGContextSaveGState(currentContext);
CGContextSetShadow(currentContext, CGSizeMake(4,7), 3);

// Draw stuff here, it will appear with a shadow

CGContextRestoreGState(currentContext);

// Draw stuff here, it will appear with no shadow

```

The first part of the challenge is to put a drop shadow on the image you composited onto the view in the previous challenge.

Figure 4.22 Drop Shadow



Gradients allow you to do shading that moves smoothly through a list of colors. The `CGGradientRef` has a list of colors and you ask it to draw the list either linear or radial. It looks like this:

```

CGFloat locations[2] = { 0.0, 1.0 };
CGFloat components[8] = { 1.0, 0.0, 0.0, 1.0, // Start color is red
                        1.0, 1.0, 0.0, 1.0 }; // End color is yellow

CGColorSpaceRef colorspace = CGColorSpaceCreateDeviceRGB();
CGGradientRef gradient = CGGradientCreateWithColorComponents(colorspace, components,
                                                               locations, 2);

CGPoint startPoint = ...;
CGPoint endPoint = ...;
CGContextDrawLinearGradient(currentContext, gradient, startPoint, endPoint, 0);
CGGradientRelease(gradient);
CGColorSpaceRelease(colorspace);

```

The last argument to `CGContextDrawLinearGradient()` determines what happens before the start point and after the end point. If you want the first color to cover the space before the start point, you supply `kCGGradientDrawsBeforeStartLocation`. If you want the last color to cover the space after the end point, you supply `kCGGradientDrawsAfterEndLocation`. To use both, bitwise or them together:

```
CGContextDrawLinearGradient(currentContext, gradient, startPoint, endPoint,  
    KCGGradientDrawsBeforeStartLocation | KCGGradientDrawsAfterEndLocation);
```

The tricky thing about gradients is that they cover everything in the view. Before drawing a gradient, you typically install a clipping path on the graphics context that defines what you want painted in the gradient. Then, you draw the gradient. Once again, there is no function for clearing the clip path, so you typically save the graphics state before installing the clipping path and restore the state afterward.

If you have a `CGContextRef` and a `UIBezierPath`, here is how you install that path as the clipping path:

```
CGContextSaveGState(currentContext);  
[myPath addClip];  
  
// Draw your gradient here  
  
CGContextRestoreGState(currentContext);
```

The challenge is to fill a triangle with a gradient that goes from yellow at the bottom to green at the top.

Figure 4.23 Gradient Triangle



5

Views: Redrawing and UIScrollView

In this chapter, you are going to see how views are redrawn in response to an event. In particular, you will update Hypnosister so that when the user touches the **BNRHypnosisView**, its circle color will change. A change in color which will require the view to redraw itself. Later in the chapter, you will also add a **UIScrollView** to Hypnosister's view hierarchy.

The first step is to declare a property for the color in **BNRHypnosisView**. In earlier applications, you declared properties in header files. You can also declare properties in *class extensions*.

Open **BNRHypnosisView.m** and add the following code near the top of the file.

```
#import "BNRHypnosisView.h"

@interface BNRHypnosisView : NSObject
@property (strong, nonatomic) UIColor *circleColor;
@end

@implementation BNRHypnosisView
```

These three lines of code are a class extension with one property declaration. Why is this property declared in a class extension and not in the header file? Hold on to that question, and we will get back to it after you have finished implementing the color change. In the meantime, think of `circleColor` as just another property on **BNRHypnosisView**.

In **BNRHypnosisView.m**, update the **initWithFrame:** method to create a default `circleColor` for instances of **BNRHypnosisView**.

```
- (instancetype)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        self.backgroundColor = [UIColor clearColor];
        self.circleColor = [UIColor lightGrayColor];
    }
    return self;
}
```

In **drawRect:**, modify the message that sets the current stroke color to use `circleColor`.

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

+[UIColor lightGrayColor] setStroke];
[self.circleColor setStroke];

// Draw the line!
[path stroke];
```

You can build and run the application to confirm that it works as before. The next step is to write the code that will update `circleColor` when the view is touched.

When the user touches a view, the view is sent the message `touchesBegan:withEvent:`. The `touchesBegan:withEvent:` method is a touch event handler. You will dive into touch events and touch event handlers in detail in Chapter 12. Right now, you are simply going to override `touchesBegan:withEvent:` to change the `circleColor` property of the view to a random color.

In `BNRHypnosisView.m`, override `touchesBegan:withEvent:` to generate a log message, create a random-colored `UIColor`, and set `circleColor` to this color.

```
// When a finger touches the screen
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    NSLog(@"%@", self);

    // Get 3 random numbers between 0 and 1
    float red = (arc4random() % 100) / 100.0;
    float green = (arc4random() % 100) / 100.0;
    float blue = (arc4random() % 100) / 100.0;

    UIColor *randomColor = [UIColor colorWithRed:red
                                         green:green
                                         blue:blue
                                         alpha:1.0];

    self.circleColor = randomColor;
}
```

Build and run the application and touch anywhere on the view. Your message will appear in the console, but the circle color will not change. Your view is not being redrawn. To understand why and how to fix the problem, you need to know about the run loop.

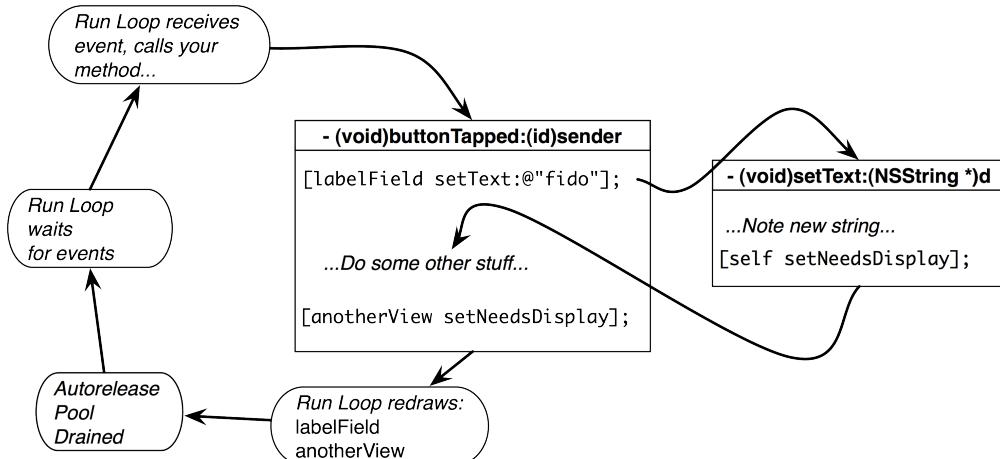
The Run Loop and Redrawing Views

When an iOS application is launched, it starts a *run loop*. The run loop's job is to listen for events, such as a touch. 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 5.1 shows where redrawing the screen happens in the run loop using an example of the user entering text into a text field.

Figure 5.1 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 your touch event is being routed correctly to `BNRHypnosisView`'s touch handler because you see your log message. But when `touchesBegan:withEvent:` finishes executing and control returns to the run loop, the run loop does not send `drawRect:` to the `BNRHypnosisView`.

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 it is sent `setText:`, since changing the text of a label requires the label to re-render its layer. In custom `UIView` subclasses, like `BNRHypnosisView`, you must send this message yourself.

In `BNRHypnosisView.m`, implement a custom accessor for the `circleColor` property to send `setNeedsDisplay` to the view whenever this property is changed.

```

- (void)setCircleColor:(UIColor *)circleColor
{
    _circleColor = circleColor;
    [self setNeedsDisplay];
}
  
```

Build and run the application again. Touch the view, and the circle color will change.

(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:` 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:`. 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.)

Class Extensions

Now let's return to the `circleColor` property that you declared in a class extension for `BNRHypnosisView`. What is the difference between a property declared in a class extension and one declared in a header file? Visibility.

A class's header file is visible to other classes. That, in fact, is its purpose. A class declares properties and methods in its header file to advertise to other classes how they can interact with the class or its instances.

Not every property or method is for public consumption, however. Properties and methods that are used internally by the class belong in a class extension. The `circleColor` property is only used by the `BNRHypnosisView` class. No other class needs to know about this property. Thus, it belongs in the class extension.

Putting properties and methods in a class extension is not being paranoid or overly proprietary. It is good practice to keep your header file as brief as it can be. This makes it easier for others to understand how they can use your class.

Syntactically, a class extension looks a little like a header file. It begins with `@interface` followed by an empty set of parentheses. The `@end` marks the end of the class extension. Typically, you put the class extension at the top of the implementation file before the `@implementation` keyword announces the start of the method definitions.

```
#import "BNRHypnosisView.h"

@interface BNRHypnosisView ()

@property (strong, nonatomic) UIColor *circleColor;

@end

@implementation BNRHypnosisView
```

The same visibility rules hold for subclasses. If you were to subclass `BNRHypnosisView`, the subclass and its instances would not know about `circleColor`.

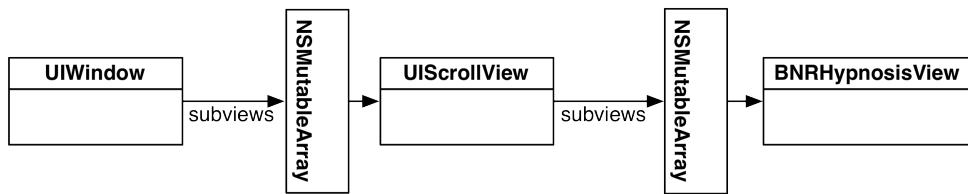
If you need limited visibility for certain properties and methods, you can create a class extension in an external file and import it into the implementation files of classes on a need-to-know basis.

We will use class extensions appropriately throughout the book to hide implementation details that do not need to be visible outside of the class.

Using UIScrollView

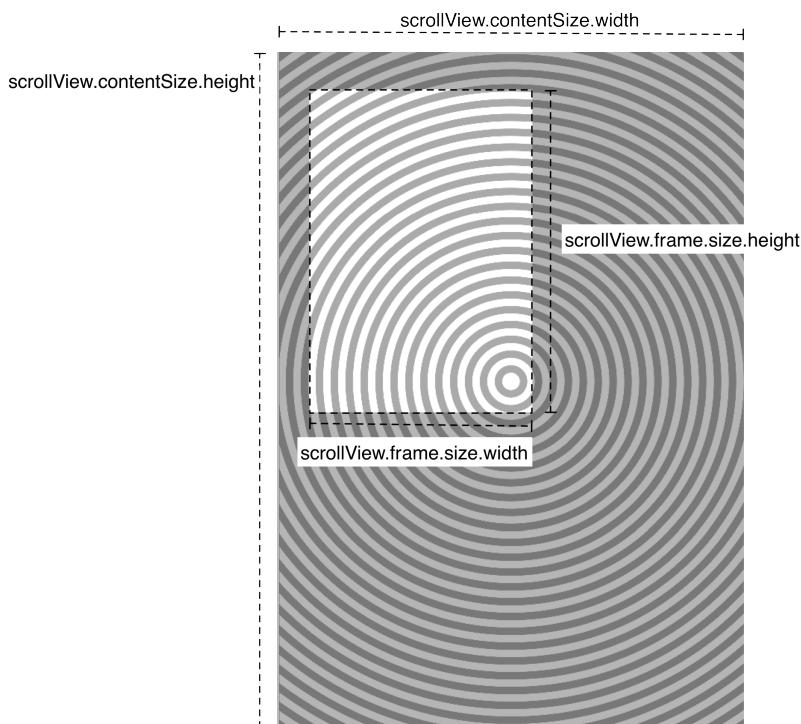
In this section, you are going to add an instance of `UIScrollView` to Hypnosister. This scroll view will be a direct subview of the window, and the instance of `BNRHypnosisView` will be a subview of the scroll view, as shown in Figure 5.2.

Figure 5.2 View hierarchy with **UIScrollView**



Scroll views are typically used for views that are larger than the screen. 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 5.3). 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 5.3 **UIScrollView** and its content area



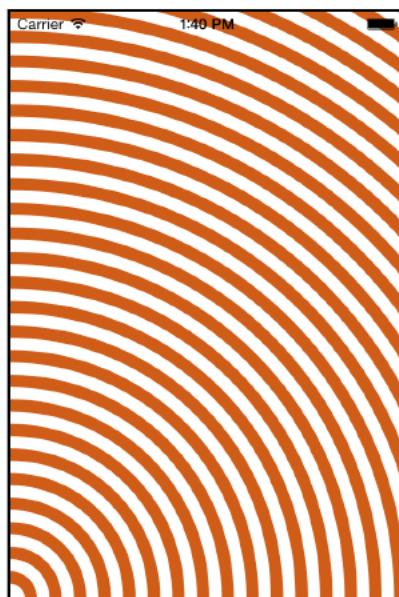
UIScrollView is a subclass of **UIView**, so it can be initialized using `initWithFrame:` and it can be added as a subview to another view.

In `BNRAppDelegate.m`, put a super-sized version of **BNRHypnosisView** inside a scroll view and add that scroll view to the window:

```
- (BOOL)application:(UIApplication *)application  
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions  
{  
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];  
    // Override point for customization after application launch  
  
    CGRect firstFrame = self.window.bounds;  
BNRHypnosisView *firstView = [[BNRHypnosisView alloc] initWithFrame: firstFrame];  
[self.window addSubview: firstView];  
  
    // Create CGRects for frames  
    CGRect screenRect = self.window.bounds;  
    CGRect bigRect = screenRect;  
    bigRect.size.width *= 2.0;  
    bigRect.size.height *= 2.0;  
  
    // Create a screen-sized scroll view and add it to the window  
    UIScrollView *scrollView = [[UIScrollView alloc] initWithFrame:screenRect];  
    [self.window addSubview:scrollView];  
  
    // Create a super-sized hypnosis view and add it to the scroll view  
    BNRHypnosisView *hypnosisView = [[BNRHypnosisView alloc] initWithFrame:bigRect];  
    [scrollView addSubview:hypnosisView];  
  
    // Tell the scroll view how big its content area is  
    scrollView.contentSize = bigRect.size;  
  
    self.window.backgroundColor = [UIColor whiteColor];
```

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

Figure 5.4 Top right quadrant of big **BNRHypnosisView**



When you go to pan around the **BNRHypnosisView**, the circle color changes. You cannot pan without beginning a touch, so the run loop sends touch events to the **UIScrollView** and to the **BNRHypnosisView**. In Chapter 13, you will see how to recognize and handle a “tap” gesture so that it can be distinguished from a touch or a drag.

“Pinch-to-zoom” is also implemented using **UIScrollView**. It does not take many lines of code, but it involves a technique that we have not covered yet. So adding pinch-to-zoom to Hypnosister will be a challenge in Chapter 7.

Panning and paging

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

In `BNRAppDelegate.m`, shrink the **BNRHypnosisView** back to the size of the screen and add a second screen-sized **BNRHypnosisView** 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
CGRect screenRect = self.window.bounds;
CGRect bigRect = screenRect;
bigRect.size.width *= 2.0;
bigRect.size.height *= 2.0;

// Create a screen-sized scroll view and add it to the window
UIScrollView *scrollView = [[UIScrollView alloc] initWithFrame:screenRect];
[self.window addSubview:scrollView];

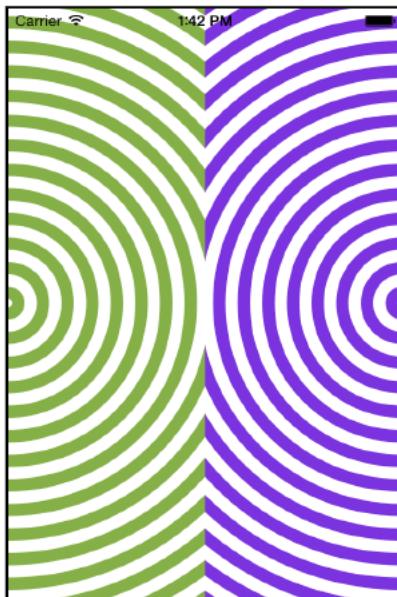
// Create a super-sized hypnosis view and add it to the scroll view
BNRHypnosisView *hypnosisView = [[BNRHypnosisView alloc] initWithFrame:bigRect];
// Create a screen-sized hypnosis view and add it to the scroll view
BNRHypnosisView *hypnosisView = [[BNRHypnosisView alloc] initWithFrame:screenRect];
[scrollView addSubview:hypnosisView];

// Add a second screen-sized hypnosis view just off screen to the right
screenRect.origin.x += screenRect.size.width;
BNRHypnosisView *anotherView = [[BNRHypnosisView alloc] initWithFrame: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 **BNRHypnosisView**. Notice that you can stop in between the two views.

Figure 5.5 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 `BNRAAppDelegate.m`.

```
UIScrollView *scrollView = [[UIScrollView alloc] initWithFrame:screenRect];
scrollView.pagingEnabled = YES;
[self.window addSubview:scrollView];
```

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.

6

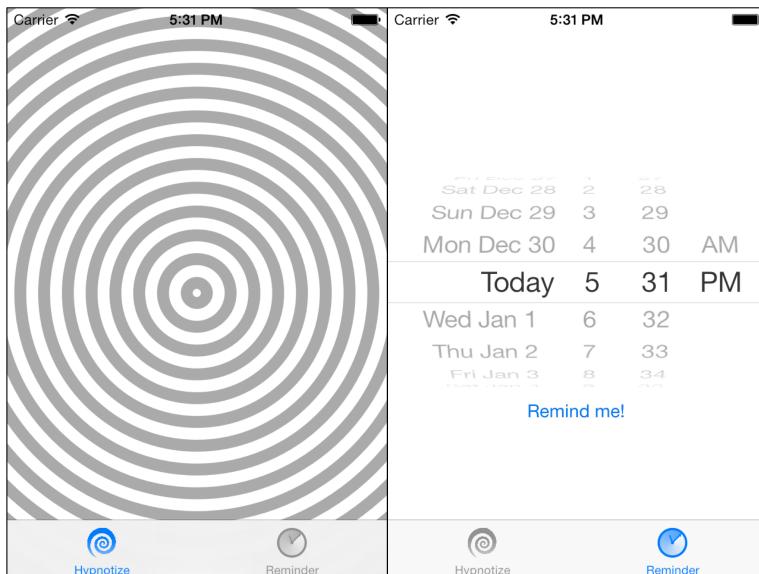
View Controllers

In Chapter 5, 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 create an application named HypnoNerd. In HypnoNerd, the user will be able to switch between two view hierarchies – one for being hypnotized and the other for setting a reminder for hypnosis on a future date.

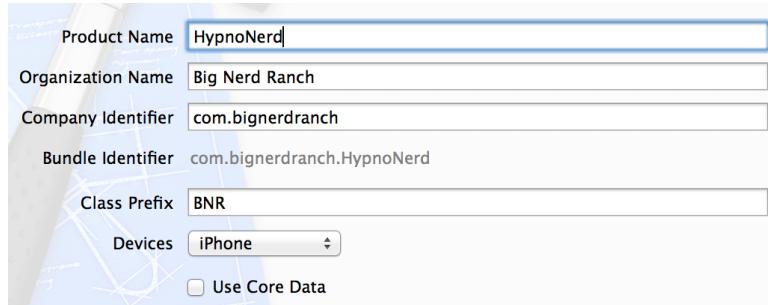
Figure 6.1 The two faces of HypnoNerd



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

Create a new iOS project (Command-Shift-N) from the Empty Application template. Name this project HypnoNerd and configure the project as shown in Figure 6.2.

Figure 6.2 Creating a new project

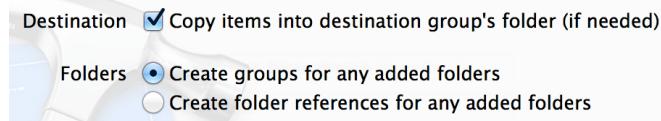


You are going to reuse the `BNRHypnosisView` class from Hypnosister in this project.

In Finder, locate the directory containing your Hypnosister project. Drag the `BNRHypnosisView.h` and `BNRHypnosisView.m` files from Finder into the project navigator in Xcode.

In the sheet that appears, check the box to Copy items into destination group's folder (if needed) and the box next to the HypnoNerd target and click Finish (Figure 6.3).

Figure 6.3 Copy files to HypnoNerd



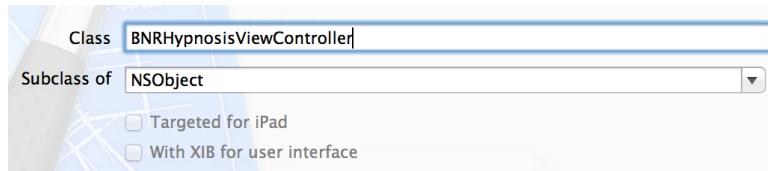
This will create copies of the two files and add them to HypnoNerd's directory on the filesystem and to the HypnoNerd project.

Subclassing UIViewController

From the File menu, select New → File... From the iOS section, select Cocoa Touch and then choose Objective-C class. Click Next.

Name this class `BNRHypnosisViewController` and choose `NSObject` as its superclass (Figure 6.4). Click Next and save the files to finish creating the class.

Figure 6.4 Creating `BNRHypnosisViewController`



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 `BNRHypnosisViewController.h` and change the superclass to `UIViewController`.

```
@interface BNRHypnosisViewController : NSObject
@interface BNRHypnosisViewController : UIViewController
@end
```

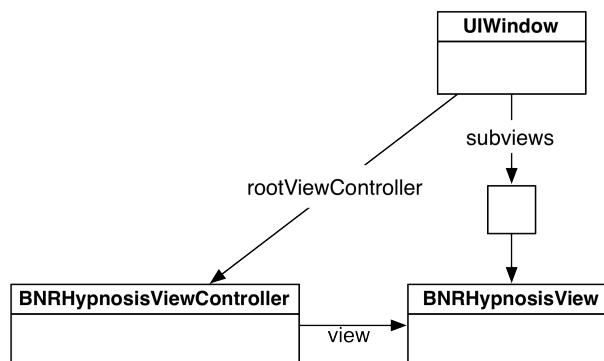
The view of a view controller

As a subclass of `UIViewController`, `BNRHypnosisViewController` inherits an important property:

```
@property (nonatomic, strong) UIView *view;
```

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 6.5 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 `BNRHypnosisViewController` consists of only one view, it is a good candidate for being created programmatically.

Creating a view programmatically

Open `BNRHypnosisViewController.m` and import the header file for `BNRHypnosisView`. Then override `loadView` to create a screen-sized instance of `BNRHypnosisView` and set it as the view of the view controller.

```
#import "BNRHypnosisViewController.h"
#import "BNRHypnosisView.h"

@implementation BNRHypnosisViewController

- (void)loadView
{
    // Create a view
    BNRHypnosisView *backgroundView = [[BNRHypnosisView alloc] init];

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

@end
```

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 `BNRHypnosisViewController` to the application window so that it will appear on screen to users.

Setting the root view controller

There is a convenient method for adding a view controller's view hierarchy to the window: `UIWindow`'s `setRootViewController`:. 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 `BNRAppDelegate.m`, import `BNRHypnosisViewController.h` at the top of the file. Then create an instance of `BNRHypnosisViewController` and set it as the `rootViewController` of the window.

```
#import "BNRAppDelegate.h"
#import "BNRHypnosisViewController.h"

@implementation BNRAppDelegate

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch

    BNRHypnosisViewController *hvc = [[BNRHypnosisViewController alloc] init];
    self.window.rootViewController = hvc;

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

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 4, you can imagine what the core of `setRootViewController`: looks like:

```
- (void)setRootViewController:(UIViewController *)viewController
{
    // Get the view of the root view controller
    UIView *rootView = viewController.view;

    // Make a frame that fits the window's bounds
    CGRect viewFrame = self.bounds;
    rootView.frame = viewFrame;

    // Insert this view as window's subview
    [self addSubview:rootView];

    // Update the instance variable
    _rootViewController = viewController;
}
```

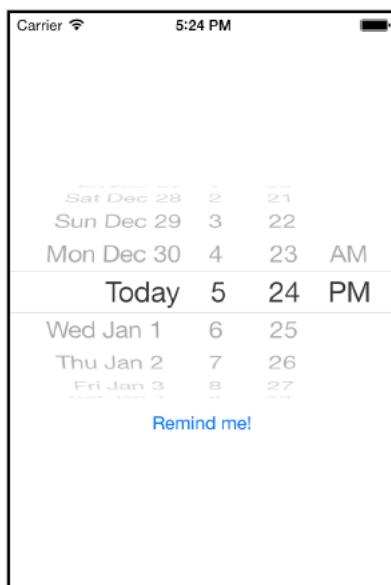
At the beginning of this implementation, the **BNRHypnosisViewController** is asked for its **view**. Because the **BNRHypnosisViewController** has just been created, its **view** is **nil**. So it is sent the **loadView** message that creates its **view**.

Build and run the application. HypnoNerd looks a lot like Hypnosister did. Under the hood, however, it is quite different. You are using a view controller to present the **BNRHypnosisView** 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 **BNRReminderViewController** 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 HypnoNerd is not running at the time.

Figure 6.6 **BNRReminderViewController**



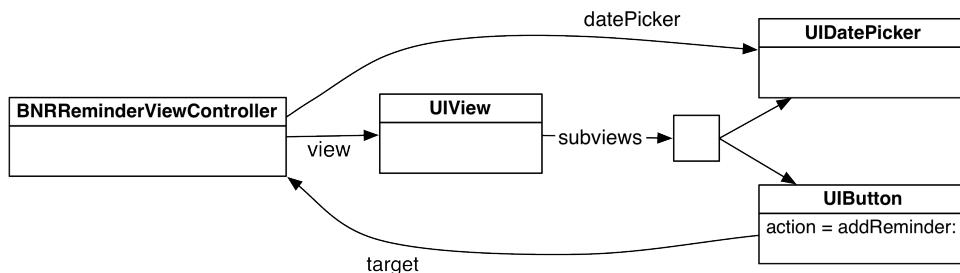
Create a new Objective-C class (Command-N). Name it **BNRReminderViewController** and make it a subclass of **NSObject**.

In **BNRReminderViewController.h**, change the superclass to **UIViewController**.

```
@interface BNRReminderViewController : NSObject
@interface BNRReminderViewController : UIViewController
```

The **BNRReminderViewController**'s view will be a full-screen **UIView** with two subviews – an instance of **UIDatePicker** and an instance of **UIButton** (Figure 6.7).

Figure 6.7 Object diagram of **BNRReminderViewController**'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 **BNRReminderViewController**'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 **BNRReminderViewController.m**. Add a class extension for **BNRReminderViewController** that includes a declaration of the **datePicker** property. Then add a simple implementation for **addReminder:** that logs the picked date.

```
#import "BNRReminderViewController.h"

@interface BNRReminderViewController ()
@property (nonatomic, weak) IBOutlet UIDatePicker *datePicker;
@end

@implementation BNRReminderViewController

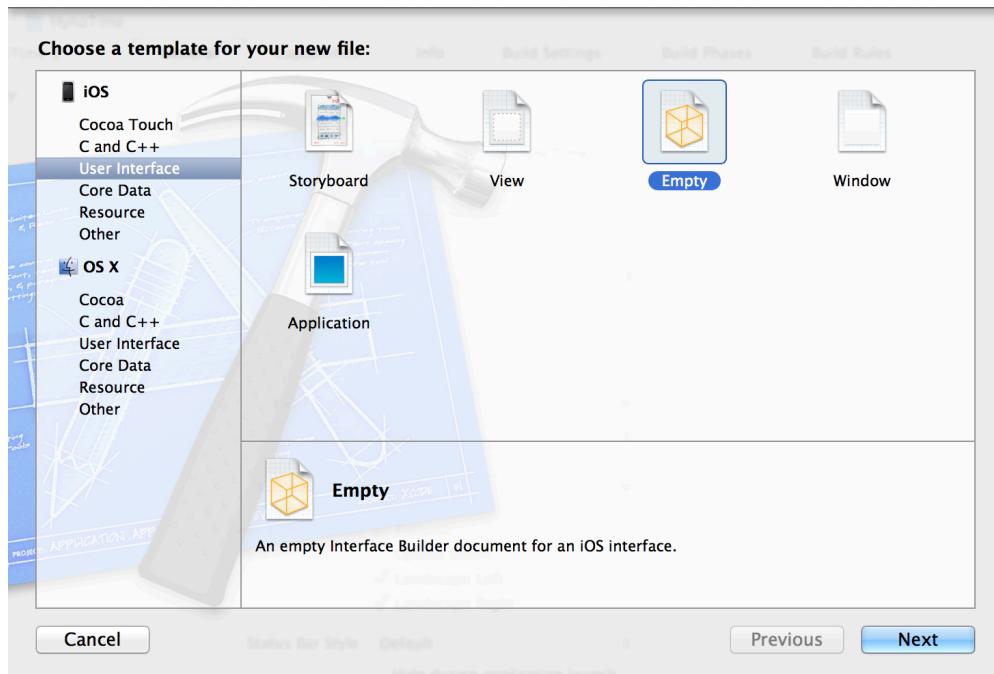
- (IBAction)addReminder:(id)sender
{
    NSDate *date = self.datePicker.date;
    NSLog(@"Setting a reminder for %@", date);
}

@end
```

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 6.8).

Figure 6.8 Creating an empty XIB



Select iPhone from the pop-up menu that appears and click `Next`.

Name this file `BNRReminderViewController.xib` 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.)

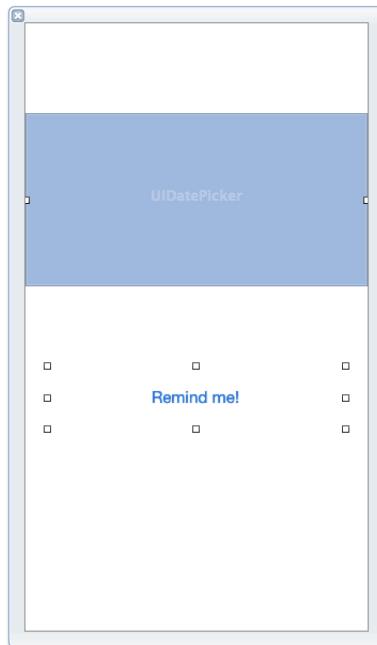
You now have a new file: `BNRReminderViewController.xib`. Select this file in the project navigator to open it in Interface Builder.

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. By default, it will be screen-sized, which is what you want.

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 6.9. Remember that you can double-click the button to change its title.

Figure 6.9 **BNRReminderViewController**'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 6.10 Hierarchy in **BNRReminderViewController.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 **BNRReminderViewController** does need to know which NIB file to load. You can do this in **UIViewController**'s designated initializer:

```
- (instancetype)initWithNibName:(NSString *)NibName bundle:(NSBundle *)nibBundleOrNil;
```

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 `BNRAppDelegate.m`, import `BNRReminderViewController.h`. Then create an instance of `BNRReminderViewController` and tell it where to find its NIB file. Finally, make the `BNRReminderViewController` object the `rootViewController` of the window.

```
#import "BNRReminderViewController.h"

@implementation BNRAppDelegate

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch

    // This line will generate a warning, ignore it for now
    BNRHypnosisViewController *hvc = [[BNRHypnosisViewController alloc] init];

    // This will get a pointer to an object that represents the app bundle
    NSBundle *appBundle = [NSBundle mainBundle];

    // Look in the appBundle for the file BNRReminderViewController.xib
    BNRReminderViewController *rvc =
        [[BNRReminderViewController alloc] initWithNibName:@"BNRReminderViewController"
                                                 bundle:appBundle];

    self.window.rootViewController = hvc;
    self.window.rootViewController = rvc;

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

The bundle that you are getting by sending the `mainBundle` message 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 `BNRReminderViewController.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
"BNRReminderViewController" nib but the view outlet was not set.'
```

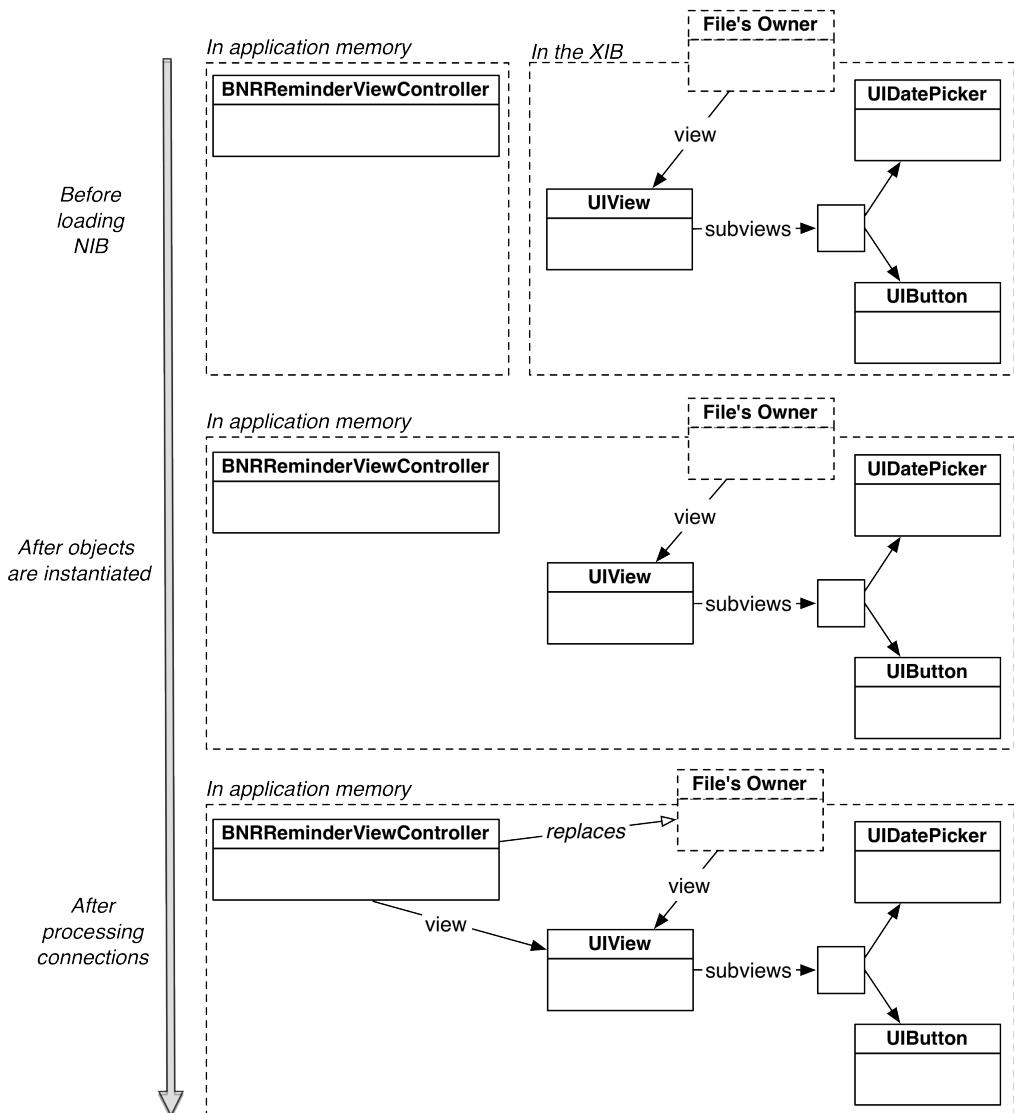
When the corresponding NIB file was loaded, these objects were instantiated. But you have not made connections to link the instantiated objects with the `BNRReminderViewController` in the running application. This includes the view controller's `view` property. Thus, when the view controller tries to 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 6.11).

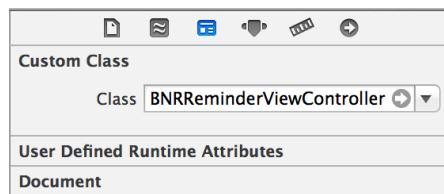
Figure 6.11 NIB loading timeline



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 **BNRReminderViewController**.

Reopen `BNRReminderViewController.xib`. Select the File's Owner object in the document outline. Then click the **Identity** tab in the inspector area to show the *identity inspector*. Change the Class for File's Owner to **BNRReminderViewController** (Figure 6.12).

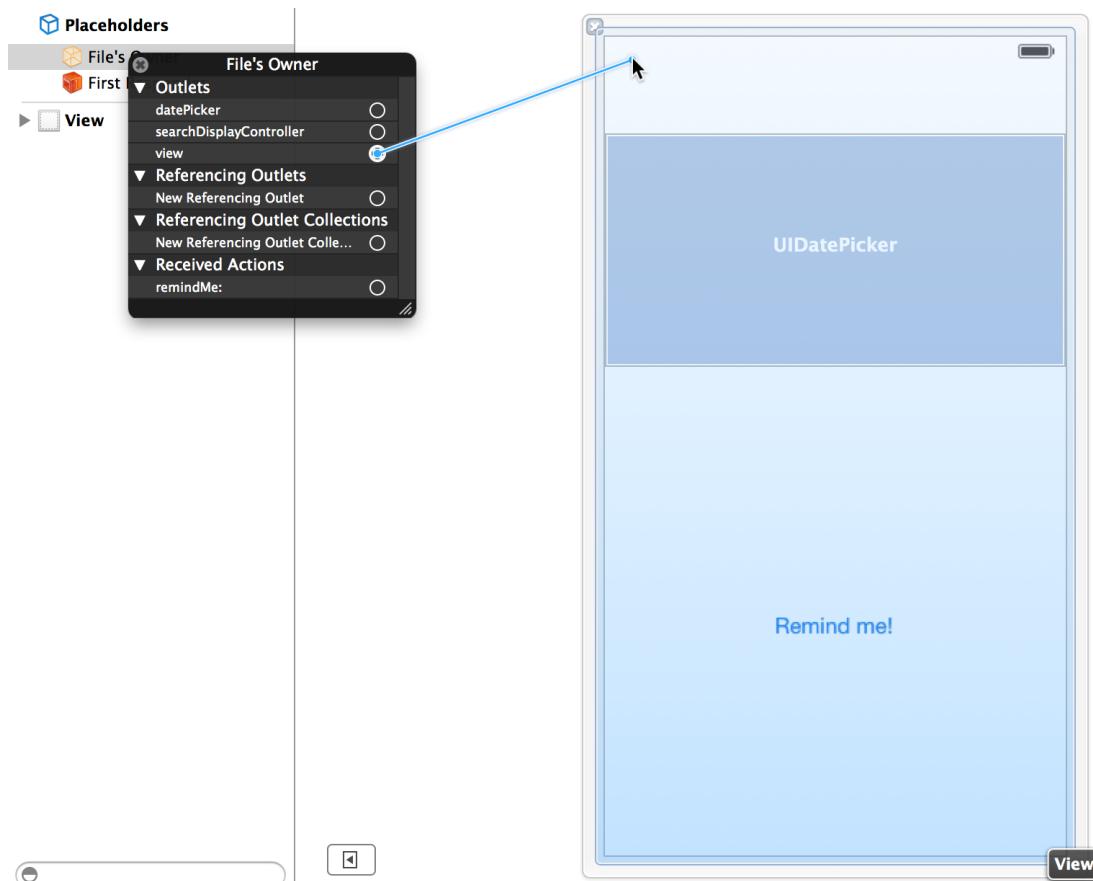
Figure 6.12 Identity inspector for File's Owner



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 6.13).

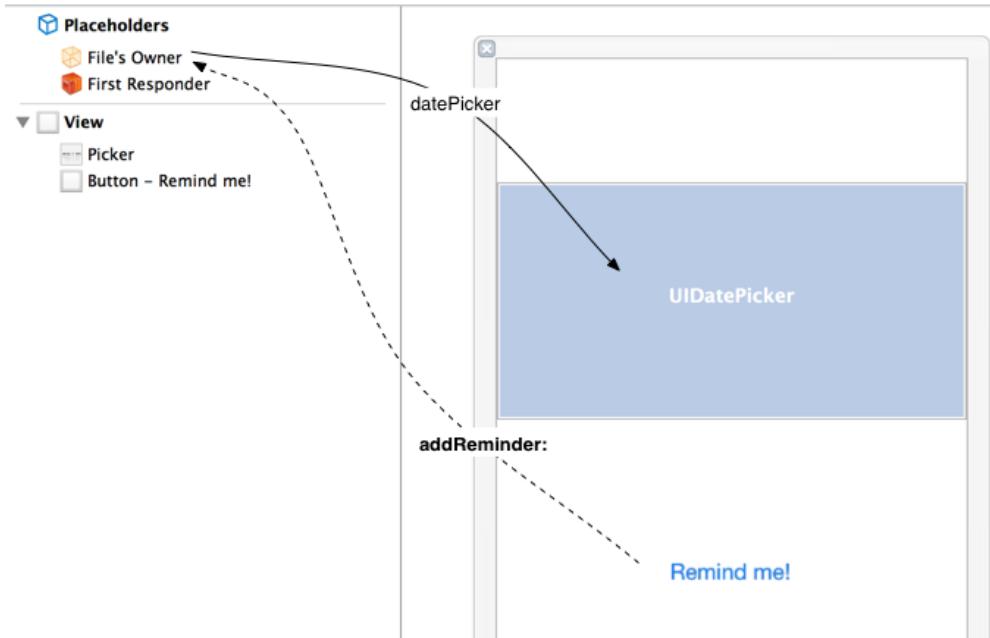
Figure 6.13 Set view outlet



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

Finish by making the remaining connections (Figure 6.14). Right-click to reveal the File's Owner's outlets and drag to connect the **datePicker** outlet to the **UIDatePicker**. Then Control-drag from the **UIButton** in the canvas to the File's Owner and select **addReminder:** to set the action.

Figure 6.14 **BNRReminderViewController** XIB connections



Build and run the application. Select a time, tap the Remind Me button, and check the console for your reminder date. Later in the chapter, you will update **addReminder:** to register a local notification.

Earlier in **BNRReminderViewController.m**, you declared the **datePicker** outlet as **weak**. Declaring outlets as **weak** is a convention from earlier versions of iOS. In these versions, a view controller's view was automatically destroyed any time that system memory was low and then was recreated later if needed. Ensuring that the view controller only had weak ownership of the subviews meant that destroying the view also destroyed all of its subviews and avoided memory leaks.

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 **BNRHypnosisViewController** and **BNRReminderViewController**.

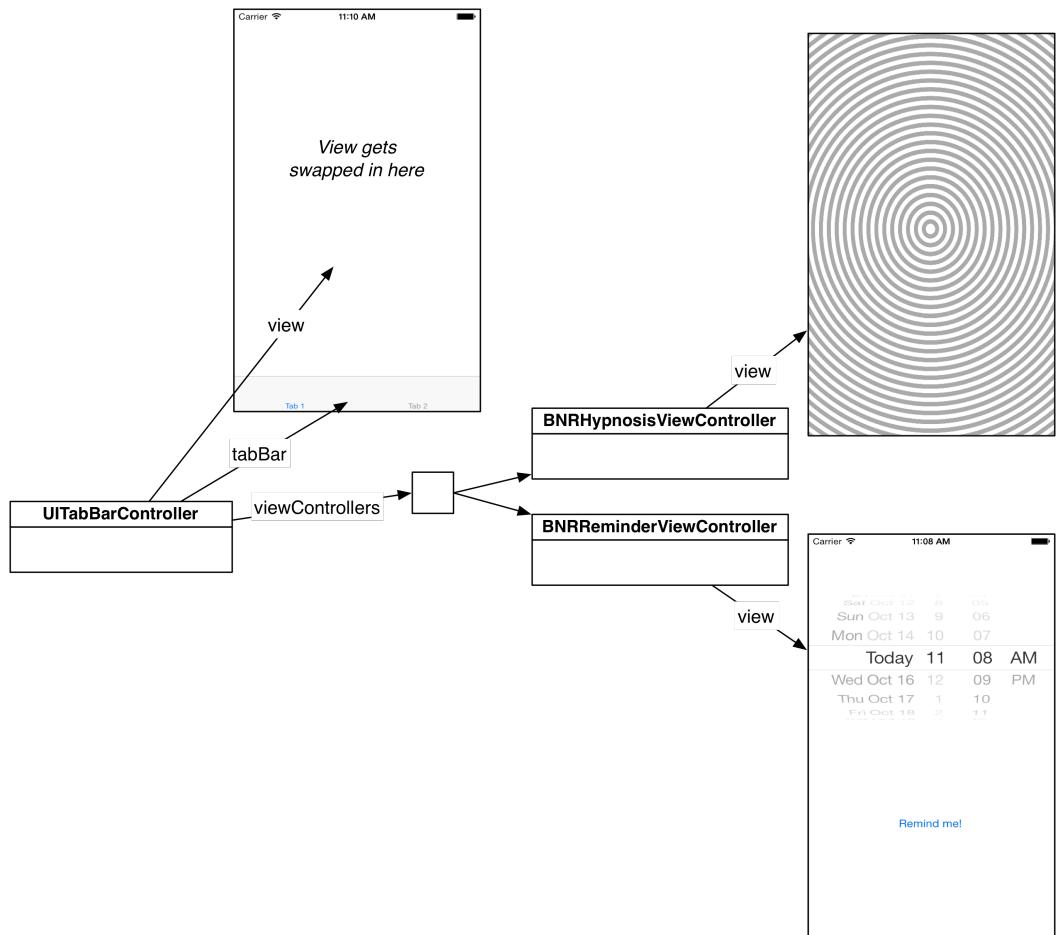
UITabBarController keeps an array of view controllers. It also maintains a tab bar at the bottom of the screen with a tab for each view controller in this array. Tapping on a tab results in the presentation of the view of the view controller associated with that tab.

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

```
- (BOOL)application:(UIApplication *)application  
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions  
{  
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];  
    // Override point for customization after application launch  
  
    BNRHypnosisViewController *hvc = [[BNRHypnosisViewController alloc] init];  
  
    // This will get a pointer to an object that represents the app bundle  
    NSBundle *appBundle = [NSBundle mainBundle];  
  
    // Look in the AppBundle for the file BNRReminderViewController.xib  
    BNRReminderViewController *rvc = [[BNRReminderViewController alloc]  
        initWithNibName:@"BNRReminderViewController"  
        bundle:appBundle];  
  
    UITabBarController *tabBarController = [[UITabBarController alloc] init];  
    tabBarController.viewControllers = @[hvc, rvc];  
  
    self.window.rootViewController = rvc;  
    self.window.rootViewController = tabBarController;  
  
    self.window.backgroundColor = [UIColor whiteColor];  
    [self.window makeKeyAndVisible];  
    return YES;  
}
```

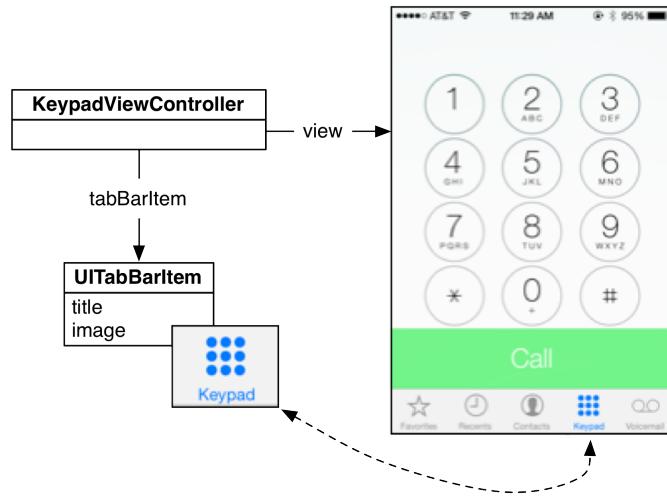
Build and run the application. The bar at the bottom is actually two tabs. Tap on the left and right sides of the tab bar to switch between the two view controllers. In the next section, you will create tab bar items to make the two tabs obvious.

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 6.15).

Figure 6.15 **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 6.16 shows an example of this relationship in the iPhone's Phone application.

Figure 6.16 **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 `BNRHypnosisViewController.m`, override `UIViewController`'s designated initializer, `initWithNibName:bundle:`, to get and set a tab bar item for `BNRHypnosisViewController`.

```

- (instancetype)initWithNibName:(NSString *)NibNameOrNil
                           bundle:(NSBundle *)nibBundleOrNilOrNil
{
    self = [super initWithNibName:nibNameOrNilOrNil
                           bundle:nibBundleOrNilOrNil];
    if (self) {
        // Set the tab bar item's title
        self.tabBarItem.title = @"Hypnotize";

        // Create a UIImage from a file
        // This will use Hypno@2x.png on retina display devices
        UIImage *i = [UIImage imageNamed:@"Hypno.png"];

        // Put that image on the tab bar item
        self.tabBarItem.image = i;
    }
    return self;
}

```

In `BNRReminderViewController.m`, do the same thing.

```
- (instancetype)initWithNibName:(NSString *)NibNameOrNil
                      bundle:(NSBundle *)bundleOrNil
{
    self = [super initWithNibName:nibNameOrNilOrNil
                           bundle:nibBundleOrNilOrNil];
    if (self) {
        // Get the tab bar item
        UITabBarItem *tbi = self.tabBarItem;

        // Give it a label
        tbi.title = @"Reminder";

        // Give it an image
        UIImage *i = [UIImage imageNamed:@"Time.png"];
        tbi.image = i;
    }
    return self;
}
```

Build and run the application, and you will see helpful images and titles in the tab bar. (Figure 6.17).

Figure 6.17 Tab bar items with labels and icons



UIViewController Initializers

When you created a tab bar item for `BNRHypnosisViewController`, you overrode `initWithNibName:bundle:`. However, when you initialized the `BNRHypnosisViewController` instance in `BNRAppDelegate.m`, you sent it `init` and still got the tab bar items. This is because `initWithNibName:bundle:` is the designated initializer of `UIViewController`. Sending `init` to a view controller calls `initWithNibName:bundle:` and passes `nil` for both arguments.

`BNRHypnosisViewController` 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 `BNRAppDelegate.m`, change your code to initialize the `BNRReminderViewController` with `init` rather than `initWithNibName:bundle:`.

```
BNRHypnosisViewController *hvc = [[BNRHypnosisViewController alloc] init];

// This will get a pointer to an object that represents the app bundle
NSBundle *appBundle = [NSBundle mainBundle];

// Look in the appBundle for the file BNRReminderViewController.xib
BNRReminderViewController *rvc =
    [[BNRReminderViewController alloc] initWithNibName:@"BNRReminderViewController"
                                         bundle:appBundle];

BNRReminderViewController *rvc = [[BNRReminderViewController alloc] init];
UITabBarController *tabBarController = [[UITabBarController alloc] init];
```

Build and run the application, and it will work just as before. When a view controller is initialized with `nil` as its NIB name, it searches for a NIB file with the name of the class. Passing `nil` as the bundle means that the view controller will look in the main application bundle. Thus, `BNRReminderViewController` will still search for `BNRReminderViewController.xib` in the main bundle.

This is why we warned you earlier about sticking to the given names when naming files. If you are creating a `FidoViewController` class that fetches its view from a NIB file, then the only appropriate name for that XIB file is `FidoViewController.xib`.

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*.)

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:

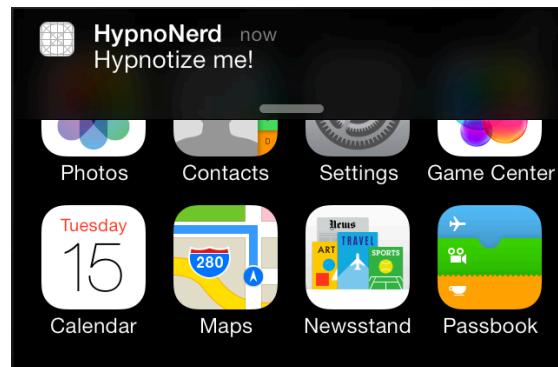
```
- (IBAction)addReminder:(id)sender
{
    NSDate *date = self.datePicker.date;
    NSLog(@"Setting a reminder for %@", date);

    UILocalNotification *note = [[UILocalNotification alloc] init];
    note.alertBody = @"Hypnotize me!";
    note.fireDate = date;

    [[UIApplication sharedApplication] scheduleLocalNotification:note];
}
```

Build and run the application. Use the date picker to select a time in the very near future and tap the Remind Me button. To see the notification, HypnoNerd needs to not be in the foreground. 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 6.18). Tapping on the notification will launch the HypnoNerd application.

Figure 6.18 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 **BNRHypnosisViewController**. This means that the **BNRReminderViewController**'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 **BNRHypnosisViewController.m**, override **viewDidLoad** to log a statement to the console.

```
- (void)viewDidLoad
{
    // Always call the super implementation of viewDidLoad
    [super viewDidLoad];

    NSLog(@"BNRHypnosisViewController loaded its view.");
}
```

In **BNRReminderViewController.m**, override the same method.

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    NSLog(@"BNRReminderViewController loaded its view.");
}
```

Build and run the application. The console reports that **BNRHypnosisViewController** loaded its view right away. Tap **BNRReminderViewController**'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 `initWithNibName: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 pointers that the view controller declares that will eventually point to subviews will be pointing to `nil`. The compiler will not complain if you send a message to one of these pointers, but whatever you intended to happen to that view object will not happen.

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 `BNRReminderViewController`'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 `BNRReminderViewController.m`, override `viewWillAppear:` to set the `minimumDate` of the date picker.

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    self.datePicker.minimumDate = [NSDate dateWithTimeIntervalSinceNow: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 likely 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. Sometimes a view controller's `view` may get destroyed and reloaded, but that is not the typical behavior on newer devices.

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 10, 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.

- **initWithNibName:bundle:** is the designated initializer for **UIViewController**.

When a view controller instance is created, its **initWithNibName: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, **BNRReminderViewController**'s **viewDidLoad** method will be called once, but **viewWillAppear:** will be called dozens of times.

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

For the More Curious: Key-Value Coding

When a NIB file is read in, the outlets are set using a mechanism called *Key-value coding* (or KVC). Key-value coding is a set of methods defined in **NSObject** that enable you to set and get the values of properties by name. Here are two of the methods:

- **(id)valueForKey:(NSString *)k;**
- **(void)setValue:(id)v forKey:(NSString *)k;**

valueForKey: is a universal getter method. You can ask any object for the value of its **fido** property like this:

```
id currentFido = [selectedObj valueForKey:@"fido"];
```

If there is a **fido** method (the **fido**-specific getter), it will be called and the returned value will be used. If there is no **fido** method, the system will go looking for an instance variable named **_fido** or **fido**. If either instance variable exists, the value of the instance variable will be used. If neither an accessor nor an instance variable exists, an exception is thrown.

setValue:forKey: is a universal setter method. It lets you set the value of an object's **fido** property like this:

```
[selectedObject setValue:userChoice forKey:@"fido"];
```

If there is a **setFido:** method, it will be called. If there is no such method, the system will go looking for a variable named **_fido** or **fido** and set the value of that variable directly. If neither the accessor nor either instance variable exists, an exception will be thrown.

When the NIB file is being loaded, the outlets are set using **setValue:forKey:**. Thus, if you set an outlet **rex** for an object in Interface Builder, that object must have an accessor called **setRex:**, an instance variable called **rex**, or an instance variable called **_rex**. If you have none of those, an exception will be thrown when the NIB file is read in at runtime. The error will look like this:

```
[<BNRSunsetViewController 0x68c0740> setValue:forUndefinedKey:]:  
this class is not key value coding-compliant for the key rex.'
```

Typically, a developer sees this error when he creates an outlet property, connects it in Interface Builder, and then renames the property.

The most important moral of this section: Using the accessor method naming conventions is more than just something nice you do for other people who might read your code. The system expects that a method called **setFido:** is the setter for the **fido** property. The system expects that the method **fido** is the getter for the **fido** property. Bad things happen when you violate the naming conventions.

Let me give you an example. When I was a young buck, I created a controller class with an outlet called **clock** that pointed to a clock-like view. I also had a button that triggered an action method (in that same controller) that went out to the Internet, got the correct time, and updated the clock-like view. I, being dumb, named this action method thusly:

```
- (IBAction)setClock:(id)sender;
```

This generated the strangest bug: When the NIB file was loaded, the action method was triggered immediately. And the **clock** outlet never got set properly, even though I had connected it correctly in the NIB file. Why? The system was trying to use my **setClock:** action method as if it were an accessor for setting my **clock** outlet.

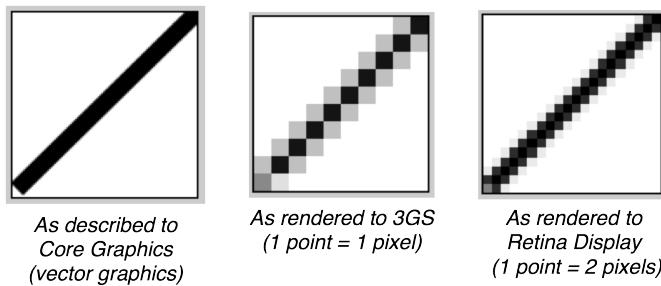
I renamed the method **updateClock:**, and everything worked perfectly – but that was four hours of my life that I will never get back. Following naming conventions is really important for iOS developers.

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.

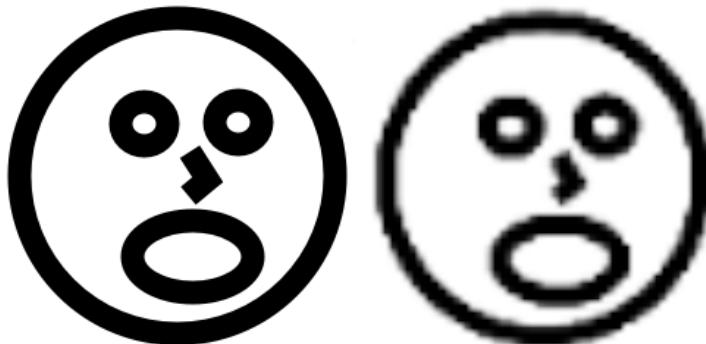
For vector graphics, like `BNRHypnosisView`'s `drawRect:` method and drawn text, you do not need to do anything; the same code will render as crisply as the device allows. However, if you draw using Core Graphics functions, these graphics will appear differently on different devices. In Core Graphics, 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 6.19).

Figure 6.19 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 6.20).

Figure 6.20 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. Then, when you use **UIImage's `imageNamed:`** method to load the image, this method looks in the bundle and gets the file that is appropriate for the particular device.

This page intentionally left blank

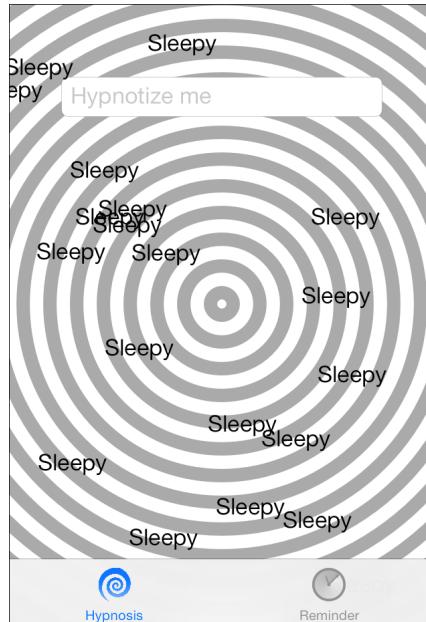
7

Delegation and Text Input

In this chapter, we will introduce **delegation**, a recurring design pattern of Cocoa Touch development. In addition, you will see how to use the debugger that Xcode provides to find and fix problems in your code.

By the end of the chapter, the HypnoNerd user will be able to display hypnotic messages on the screen using a text field (Figure 7.1).

Figure 7.1 Finished HypnoNerd



Text Fields

Open the HypnoNerd application that you started in the previous chapter.

You have already seen one way to display text on your user interfaces using a **UILabel**. Now let's take a look at another way to display text using a **UITextField**. An instance of **UITextField** allows the user to modify the text, much like a username or password field on a website.

Open `BNRHypnosisViewController.m` and modify the `loadView` to add a **UITextField** to its view.

```
- (void)loadView
{
    CGRect frame = [UIScreen mainScreen].bounds;
    BNRHypnosisView *backgroundView = [[BNRHypnosisView alloc] initWithFrame:frame];

    CGRect textFieldRect = CGRectMake(40, 70, 240, 30);
    UITextField *textField = [[UITextField alloc] initWithFrame:textFieldRect];

    // Setting the border style on the text field will allow us to see it more easily
    textField.borderStyle = UITextBorderStyleRoundedRect;
    [backgroundView addSubview:textField];

    self.view = backgroundView;
}
```

Build and run the application and you should see the text field on the Hypnotize tab. Tap on the text field, and the keyboard will slide up from the bottom of the screen, allowing you to input text. To understand how this is happening under the hood, you need to understand the *first responder*.

UIResponder

UIResponder is an abstract class in the UIKit framework. It is the superclass of three classes that you have already encountered:

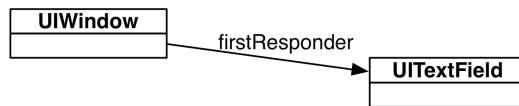
- **UIView**
- **UIViewController**
- **UIApplication**

UIResponder defines methods for handling (or “responding to”) events: touch events, motion events (like a shake), and remote control events (like pausing or playing). Subclasses override these methods to customize how they respond to events.

With touch events, it is obvious which view the user has touched. Touch events are sent directly to that view. You saw an example of this in Chapter 5.

What about the other types of events? The **UIWindow** has a pointer called **FirstResponder** which indicates who should respond to the other types of events. When you select a text field, for example, the window moves its **FirstResponder** pointer to that text field. Motion and remote control events are sent to the first responder.

Figure 7.2 **FirstResponder**



When a text field or a text view becomes **FirstResponder**, it shows its keyboard. When it loses first responder status, it hides its keyboard. If you want one of these views to become first responder, you send it the message **becomeFirstResponder** and the keyboard appears. When you want to hide the keyboard, you send it the message **resignFirstResponder**.

Most views refuse to become first responder; they do not want to steal focus from the currently selected text field or text view. An instance of **UISlider**, for example, handles touch events but will never accept first responder status.

Configuring the keyboard

The keyboard's appearance is determined by a set of the **UITextField**'s properties called **UITextInputTraits**. Let's modify some of these to give the text field some placeholder text and to modify the keyboard's return type.

```
- (void)loadView
{
    CGRect frame = [UIScreen mainScreen].bounds;
    BNRHypnosisView *backgroundView = [[BNRHypnosisView alloc] initWithFrame:frame];

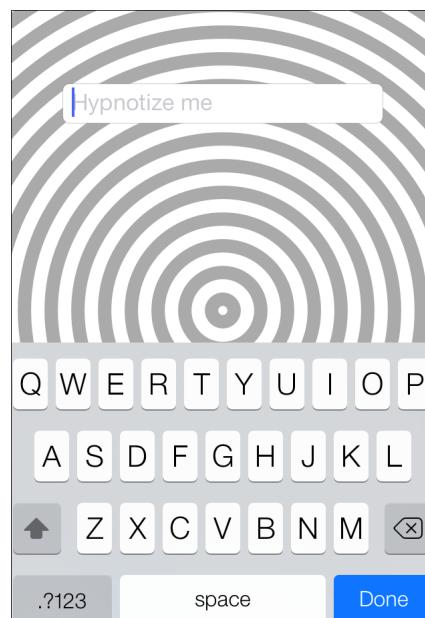
    CGRect textFieldRect = CGRectMake(40, 70, 240, 30);
    UITextField *textField = [[UITextField alloc] initWithFrame:textFieldRect];

    // Setting the border style on the text field will allow us to see it more easily
    textField.borderStyle = UITextBorderStyleRoundedRect;
    textField.placeholder = @"Hypnotize me";
    textField.returnKeyType = UIReturnKeyDone;

    [backgroundView addSubview:textField];
    self.view = backgroundView;
}
```

Build and run the application. Now the text field has a placeholder string that will be displayed until the user types in some text. Also, the return key now says Done instead of the default Return. Figure 7.3 shows what the interface looks like with these changes.

Figure 7.3 Configured text field



If you tap the Done key, you will notice that nothing happens. Changing the return key type has no impact on the functionality of the return key. In fact, the return key does not do anything automatically; you have to implement the return key functionality yourself. Before you do that, though, let's take a look at some of the other useful properties that you can use to configure the keyboard.

| | |
|-------------------------------|--|
| autocapitalizationType | This determines how capitalization is handled. The options are none, words, sentences, or all characters. |
| autocorrectionType | This will suggest and correct unknown words. This value can be YES or NO. |
| enablesReturnKeyAutomatically | This value can be YES or NO. If set to yes, the return key will be disabled if no text has been typed. As soon as any text is entered, the return key becomes enabled. |
| keyboardType | This determines the type of keyboard that will be displayed. Some examples are the ASCII keyboard, email address keyboard, number pad, and the URL keyboard. |
| secureTextEntry | Setting this to YES makes the text field behave like a password field, hiding the text that is entered. |

Delegation

You have already seen the Target-Action pattern. This is one form of callbacks that is used by UIKit: When a button is tapped, it sends its action message to its target. This typically triggers code that you have written.

A button's life is relatively simple. For objects with more complex lives, like a text field, Apple uses the *delegation pattern*. You introduce the text field to one of your objects: "This is your delegate, when anything interesting happens in your life, send a message to him." The text field keeps a pointer to its delegate. Many of the messages it sends to its delegates are informative: "OK, I am done editing!". Here are some of those:

```
- (void)textFieldDidEndEditing:(UITextField *)textField;
- (void)textFieldDidBeginEditing:(UITextField *)textField;
```

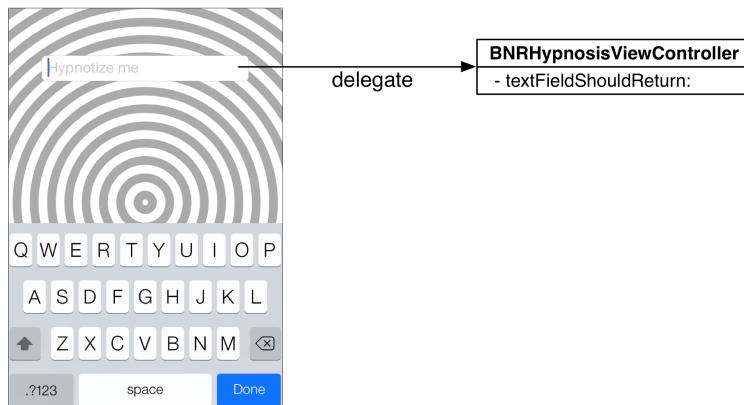
Notice that it always sends itself as the first argument to the delegate method.

Some of the messages it sends to its delegate are queries: "I am about to end editing and hide the keyboard. OK?" Here are some of those:

```
- (BOOL)textFieldShouldEndEditing:(UITextField *)textField;
- (BOOL)textFieldShouldBeginEditing:(UITextField *)textField;
- (BOOL)textFieldShouldClear:(UITextField *)textField;
- (BOOL)textFieldShouldReturn:(UITextField *)textField;
```

You are now going to make your **BNRHypnosisViewController** the delegate of the text field. You will implement the **textFieldShouldReturn:** method. When you run it, you will see that the method gets called automatically when the user taps the Done button.

Figure 7.4 **BNRHypnosisViewController** as UITextField delegate



In `BNRHypnosisViewController.m`, update `loadView` to set the delegate property of the `UITextField` to point at the `BNRHypnosisViewController`.

```
- (void)loadView
{
    CGRect frame = [UIScreen mainScreen].bounds;
    BNRHypnosisView *backgroundView = [[BNRHypnosisView alloc] initWithFrame:frame];

    CGRect textFieldRect = CGRectMake(40, 70, 240, 30);
    UITextField *textField = [[UITextField alloc] initWithFrame:textFieldRect];

    // Setting the border style on the text field will allow us to see it more easily
    textField.borderStyle = UITextBorderStyleRoundedRect;
    textField.placeholder = @"Hypnotize me";
    textField.returnKeyType = UIReturnKeyDone;

    // There will be a warning on this line. We will discuss it shortly.
    textField.delegate = self;

    [backgroundView addSubview:textField];
    self.view = backgroundView;
}
```

The method `textFieldShouldReturn:` takes in just one argument: the text field whose return key was tapped. For now, the application will just print the text of the text field to the console.

In `BNRHypnosisViewController.m`, implement the `textFieldShouldReturn:`. Be very careful that there are no typos or capitalization errors, or the method will not be called. The selector of the message the text field sends must exactly match the selector of the method implemented.

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField
{
    NSLog(@"%@", textField.text);
    return YES;
}
```

Build and run the application, type some text into the text field, and tap the return key. The text should print to the console.

Notice that you did not need to implement *all* of text field's delegate methods, just the one that you cared about. At runtime the text field will ask its delegate if it implements a method before calling it.

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.

(If you are coming from Java or C#, you would use the word “interface” instead of “protocol”.)

The protocol for **UITextField**'s delegate looks like this:

```
@protocol UITextFieldDelegate <NSObject>

@optional

- (BOOL)textFieldShouldBeginEditing:(UITextField *)textField;
- (void)textFieldDidBeginEditing:(UITextField *)textField;
- (BOOL)textFieldShouldEndEditing:(UITextField *)textField;
- (void)textFieldDidEndEditing:(UITextField *)textField;
- (BOOL)textField:(UITextField *)textField
    shouldChangeCharactersInRange:(NSRange)range
        replacementString:(NSString *)string;
- (BOOL)textFieldShouldClear:(UITextField *)textField;
- (BOOL)textFieldShouldReturn:(UITextField *)textField;

@end
```

This protocol, like all protocols, is declared with the directive `@protocol` followed by its name, `UITextFieldDelegate`. The `NSObject` in angled brackets refers to the `NSObject` protocol and tells us that `UITextFieldDelegate` includes all of the methods in the `NSObject` protocol. The methods specific to `UITextFieldDelegate` are declared next, and the protocol is closed with an `@end` directive.

Note that a protocol is not a class; it is simply a list of method declarations. 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.

The `UITextFieldDelegate` protocol is part of the iOS SDK. Protocols in the iOS SDK have reference pages in the developer documentation where you can see what methods are declared. You can also write your own protocol. You will do that in Chapter 22.

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 `UITextFieldDelegate` 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 send that message by sending another message, `respondsToSelector:`. Every object implements this method, which checks at runtime whether an object implements a given method. You can turn a method selector into a value that you can pass as an argument with the `@selector()` directive. For example, `UITextField` could implement a method that looks like this:

```

- (void)clearButtonTapped
{
    // textFieldShouldClear: is an optional method,
    // so we check first
    SEL clearSelector = @selector(textFieldShouldClear);

    if ([self.delegate respondsToSelector:clearSelector]) {
        if ([self.delegate textFieldShouldClear:self]) {
            self.text = @"";
        }
    }
}

```

If a method in a protocol is required, then the message will be sent without checking first. This means that if the delegate does not implement that method, an unrecognized selector exception will be thrown, and the application will crash.

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 either in the class header file or the class extension: the protocols that a class conforms to are added to a comma-delimited list inside angled brackets in the interface declaration.

In `BNRHypnosisViewController.m`, declare that `BNRHypnosisViewController` conforms to the `UITextFieldDelegate` protocol in the class extension. The reason for adding it to the class extension rather than the header file is the same reason as always: add to the class extension if the information (conforming to a particular protocol in this case) does not need to be publicly visible, and add it to the header file if other objects do need to know about the information.

```

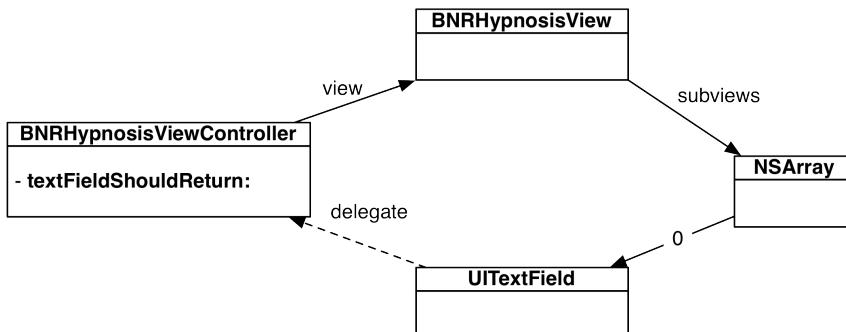
@interface BNRHypnosisViewController () <UITextFieldDelegate>
@end

```

Build the application again. Now that you have declared that `BNRHypnosisViewController` conforms to the `UITextFieldDelegate` protocol, the warning from the line of code where you set the delegate disappears. Furthermore, if you want to implement additional methods from the `UITextFieldDelegate` protocol in `BNRHypnosisViewController`, those methods will now be auto-completed by Xcode.

Many classes have a delegate pointer, and it is nearly always a weak reference to prevent strong reference cycles. In this case, for example, your view controller indirectly owns the text field. If the text field owned its delegate, you would have a strong reference cycle that would cause a memory leak.

Figure 7.5 Preventing strong reference cycle



Adding the Labels to the Screen

To make things a little interesting, you are going to add instances of **UILabel** to the screen at random positions. In **BNRHypnosisViewController.m**, implement a new method that will draw a given string on the screen twenty times at random positions.

```
- (void)drawHypnoticMessage:(NSString *)message
{
    for (int i = 0; i < 20; i++) {

        UILabel *messageLabel = [[UILabel alloc] init];

        // Configure the label's colors and text
        messageLabel.backgroundColor = [UIColor clearColor];
        messageLabel.textColor = [UIColor whiteColor];
        messageLabel.text = message;

        // This method resizes the label, which will be relative
        // to the text that it is displaying
        [messageLabel sizeToFit];

        // Get a random x value that fits within the hypnosis view's width
        int width =
            (int)(self.view.bounds.size.width - messageLabel.bounds.size.width);
        int x = arc4random() % width;

        // Get a random y value that fits within the hypnosis view's height
        int height =
            (int)(self.view.bounds.size.height - messageLabel.bounds.size.height);
        int y = arc4random() % height;

        // Update the label's frame
        CGRect frame = messageLabel.frame;
        frame.origin = CGPointMake(x, y);
        messageLabel.frame = frame;

        // Add the label to the hierarchy
        [self.view addSubview:messageLabel];
    }
}
```

In **BNRHypnosisViewController.m**, update the **textFieldShouldReturn:** method to call this new method, passing in the text field's text, clear the text that the user typed, and then dismiss the keyboard by calling **resignFirstResponder**.

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField
{
    NSLog(@"%@", textField.text);
    [self drawHypnoticMessage:textField.text];

    textField.text = @"";
    [textField resignFirstResponder];

    return YES;
}
```

Build and run the application, and enter some text into the text field. After tapping the return key, the text should be displayed on instances of **UILabel** across the view.

Motion Effects

iOS devices have a lot of powerful components embedded within them. A few of these – the accelerometer, magnetometer, and gyroscope – help determine the orientation of the device. They are how the device knows, for example, whether to display in portrait or landscape orientation. Starting in iOS 7, Apple introduced a way for applications to easily take advantage of these sensors by adding built-in parallax.

When you drive down the road, the signs along the shoulder appear to move much more quickly than trees in the distance. Your brain interprets this difference in apparent speed as movement in space. This visual effect is called “parallax”. With iOS 7, you have probably noticed this on the home screen where the icons appear to move relative to the wallpaper when you tilt the device. It is used subtly (and not so subtlety) in various places across the operating system and bundled apps, including the red badges on Home screen icons, the volume changer pop-up, and alert views.

Applications can access the same technology that powers those effects by using the **UIInterpolatingMotionEffect** class. Instances are given an axis (either horizontal or vertical), a key path (which property of the view do you want to impact), and a relative minimum and maximum value (how much the key path is allowed to sway in either direction).

In `BNRHypnosisViewController.m`, modify the **drawHypnoticMessage:** method to add a vertical and horizontal motion effect to each label that allows its center to sway 25 points in either direction.

```
[self.view addSubview:messageLabel];

UIInterpolatingMotionEffect *motionEffect;
motionEffect = [[UIInterpolatingMotionEffect alloc] initWithKeyPath:@"center.x"
    type:UIInterpolatingMotionEffectTypeTiltAlongHorizontalAxis];
motionEffect.minimumRelativeValue = @(-25);
motionEffect.maximumRelativeValue = @(25);
[messageLabel addMotionEffect:motionEffect];

motionEffect = [[UIInterpolatingMotionEffect alloc] initWithKeyPath:@"center.y"
    type:UIInterpolatingMotionEffectTypeTiltAlongVerticalAxis];
motionEffect.minimumRelativeValue = @(-25);
motionEffect.maximumRelativeValue = @(25);
[messageLabel addMotionEffect:motionEffect];
}
```

In order to test motion effects, the application must be running on a device. If you have a device provisioned for developer use, build and run the application on the device. Add some hypnotic messages to the view, and tilt the device slightly relative to your face. You will notice the magical illusion of depth that the motion effects provide.

Using the Debugger

When an application is launched from Xcode, the debugger is attached to it. The debugger monitors the current state of the application, like what method it is currently executing and the values of the

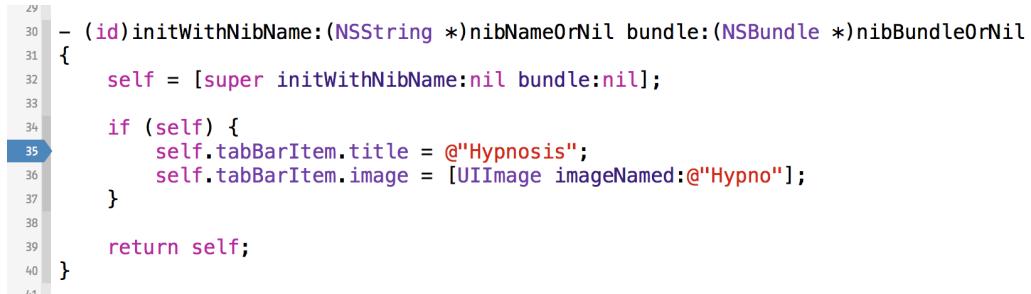
variables that are accessible from that method. Using the debugger can help you understand what an application is actually doing, which, in turn, helps you find and fix bugs.

Using breakpoints

One way to use the debugger is to set a *breakpoint*. Setting a breakpoint on a line of code pauses the execution of the application at that line (before it executes). Then you can execute the subsequent code line by line. This is useful when your application is not doing what you expected and you need to isolate the problem.

In the project navigator, select `BNRHypnosisView.m` (not `BNRHypnosisViewController.m`). Find the line of code in `initWithFrame:` that sets the `circleColor` property to light gray. Set a breakpoint by clicking the gutter (the lightly shaded bar on the left side of the editor area) next to that line of code (Figure 7.6). The blue indicator shows where the application will “break” the next time you run it.

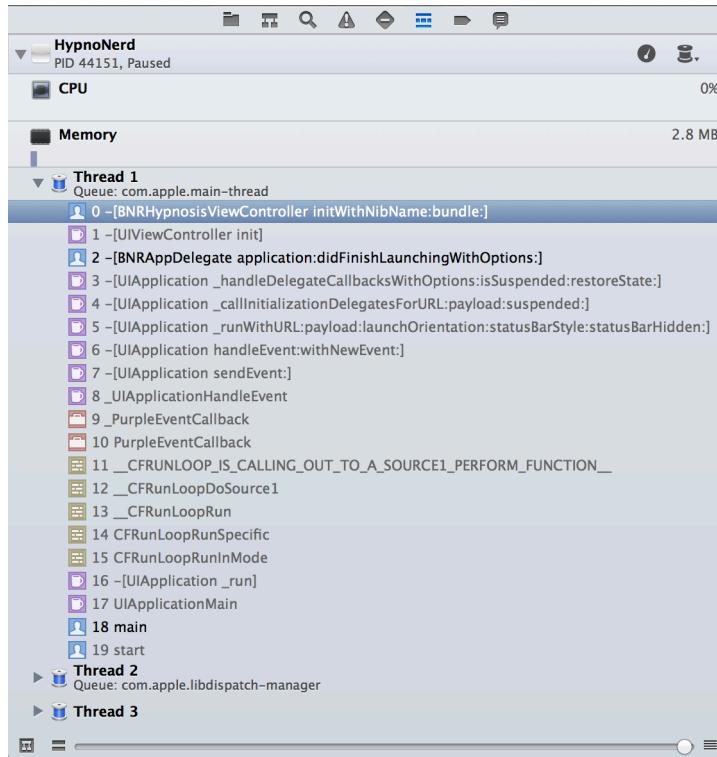
Figure 7.6 A breakpoint



Build and run the application. The application will start and then halt before the line of code where you put the breakpoint is executed. Notice the light green indicator and shading that appear to show the current point of execution.

Now your application is temporarily frozen in time, and you can examine it more closely. In the navigator area, click the `≡` tab to open the *debug navigator*. This navigator shows a *stack trace* of where the breakpoint stopped execution (Figure 7.7). A stack trace shows you the methods and functions whose frames were in the stack when the application broke. The slider at the bottom of the debug navigator expands and collapses the stack. Drag it to the right to see all of the methods in the stack trace.

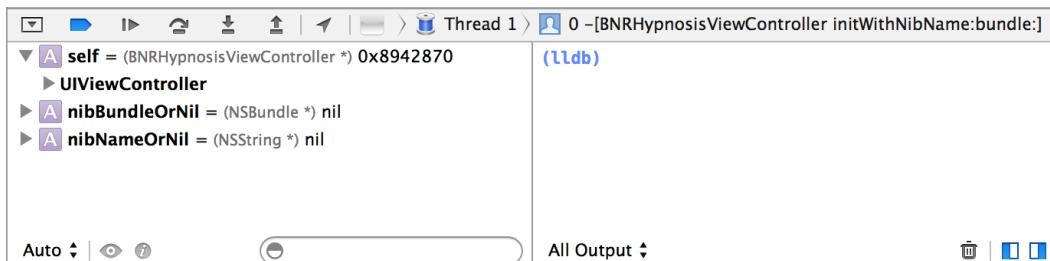
Figure 7.7 The debug navigator



The method where the break occurred is at the top of the stack trace. It was called by the method just below it, which was called by the method just below it, and so on. Notice that the methods that you have written code for are in black text while the methods belonging to Apple are in gray.

Select the method at the top of the stack. In the debug area below the editor area, check out the variables view to the left of the console. This area shows the variables within the scope of `BNRHypnosisView`'s `initWithFrame:` method along with their current values (Figure 7.8).

Figure 7.8 Debug area with variables view



(If you do not see the variables view, find the control in the bottom righthand corner of the console. Click the left icon to show the variables view.)

In the variables view, a variable that is a pointer shows the object's address. You can see that `self` has an address because in the context of this method, `self` is a pointer to the instance of `BNRHypnosisView`, and this instance was allocated before the application halted.

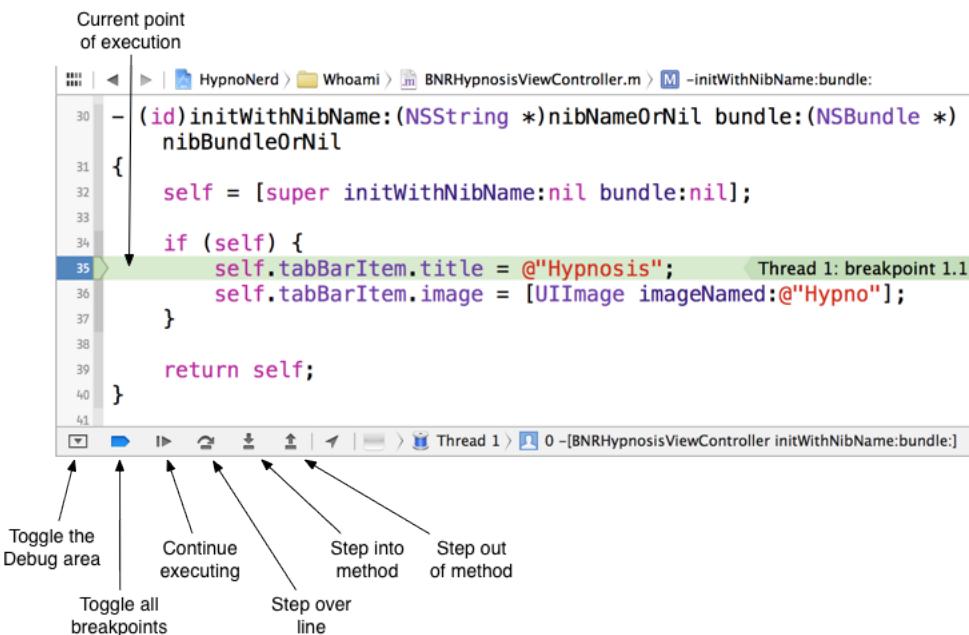
Click the disclosure button next to `self`. The first item under `self` is the superclass. The superclass of `BNRHypnosisView` is `UIView`. Clicking the disclosure button next to `UIView` will show the variables `self` inherits from its superclass.

`BNRHypnosisView` has a variable of its own, `_circleColor`. The breakpoint is set to the line that sets the `circleColor` property. That line of code has yet to be executed, so `_circleColor` currently points to `nil`.

Stepping through code

In addition to giving you a snapshot of the application at a given point, the debugger also allows you to *step through* your code line by line and see what your application does as each line executes. The buttons that control the execution are on the *debugger bar* that sits between the editor area and the debug area (Figure 7.9).

Figure 7.9 Debugger bar



Click the button that steps over a line. This will execute just the current line of code, which sets `circleColor`. Notice that the green execution indicator and shading move to the next line. Even more interesting, the variables view shows that the value of `_circleColor` has changed to a valid address.

At this point, you could continue stepping through the code to see what happens. Or you could click the button to continue executing your code normally. Or you could step into a method. Stepping into a method takes you to the method that is called by the line of code that currently has the green execution indicator. Once you are in the method, you have the chance to step through its code in the same way.

When you step out of a method, you are taken to the method that called it. To try it out, click the button to step out of the current method. You will be taken to the implementation of `loadView` in `BNRHypnosisViewController.m`.

Deleting breakpoints

To run your application normally again, you are going to get rid of the breakpoint. Right-click the blue indicator and select Delete Breakpoint. You can build and run to confirm that the application runs as expected.

Sometimes, a developer will set a breakpoint and forget about it. Then, when the application is run, execution stops, and it looks like the application has crashed. If an application of yours unexpectedly stops, make sure you are not halting on a forgotten breakpoint.

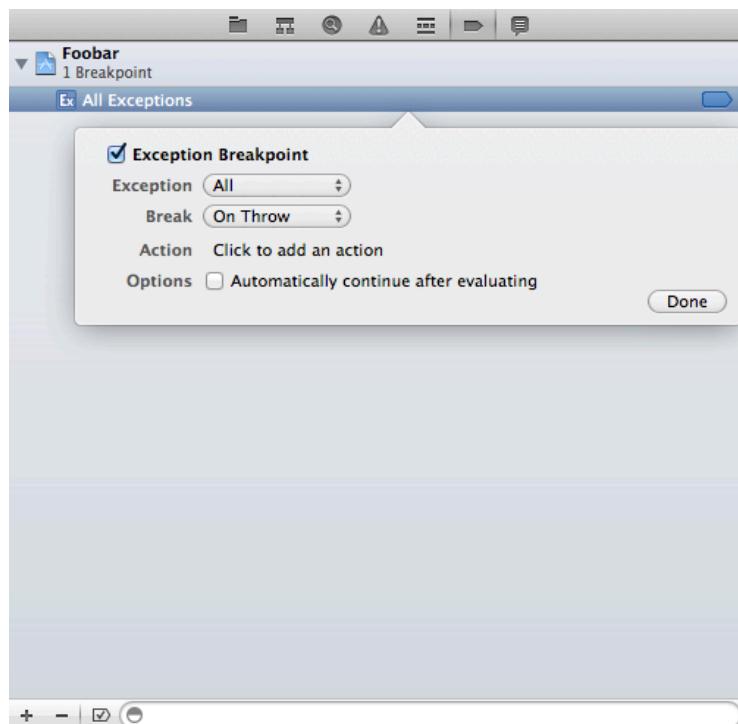
If you are not sure where you may have left a breakpoint, you can view a list of breakpoints in your project in the breakpoint navigator (the  tab in the navigator area).

Setting an exception breakpoint

You can also tell the debugger to break automatically on any line that causes your application to crash or that causes an exception to be thrown.

In the navigator area, select the  tab to open the breakpoint navigator. At the bottom of this navigator, click the + icon and select Add Exception Breakpoint....

Figure 7.10 Adding an exception breakpoint



If your application is throwing exceptions and you are not sure why, adding an exception breakpoint will help you pinpoint what is going on.

For the More Curious: main() and UIApplication

A C application begins by executing a `main` function. An Objective-C application is no different, but you have not seen `main()` in any of your iOS applications. Let's take a look now.

Open `main.m` in the HypnoNerd project navigator. It looks like this:

```
int main(int argc, char *argv[])
{
    @autoreleasepool {
        return UIApplicationMain(argc, argv,
                               nil, NSStringFromClass([BNRAppDelegate class]));
    }
}
```

The function `UIApplicationMain` creates an instance of a class called `UIApplication`. For every application, there is a single `UIApplication` instance. This object is responsible for maintaining the run loop. Once the application object is created, its run loop essentially becomes an infinite loop: the executing thread will never return to `main()`.

Another thing the function `UIApplicationMain` does is create an instance of the class that will serve as the `UIApplication`'s delegate. Notice that the final argument to the `UIApplicationMain` function is an `NSString` that is the name of the delegate's class. So, this function will create an instance of `BNRAppDelegate` and set it as the delegate of the `UIApplication` object.

The first event added to the run loop in every application is a special “kick-off” event that triggers the application to send a message to its delegate. This message is `application:didFinishLaunchingWithOptions:`. You implemented this method in `BNRAppDelegate.m` to create the window and the controller objects used in this application.

Every iOS application follows this pattern. If you are still curious, go back and check the `main.m` file in the Quiz application that you wrote in Chapter 1.

Silver Challenge: Pinch to Zoom

Add pinch-to-zoom to the Hypnosister project from Chapter 5.

The first step is to give the scroll view a delegate:

- `BNRAppDelegate` should conform to the `UIScrollViewDelegate` protocol.
- In `application:didFinishLaunchingWithOptions:`, set the scroll view's `delegate` property.

To perform as the scroll view's delegate, `BNRAppDelegate` will need a property that points to the instance of `BNRHypnosisView`. Add this property in a class extension in `BNRAppDelegate.m` and update the rest of the code to use the property instead of the `BNRHypnosisView` local variable.

To set up the scroll view, you will need to give it one `BNRHypnosisView` as a subview and turn off the paging. The scroll view also needs limits on how much it can zoom in and out. Find the relevant `UIScrollView` properties to set in this class's reference page in the documentation.

Finally, you will need to implement the scroll view delegate method `viewForZoomingInScrollView:` to return the `BNRHypnosisView`.

If you get stuck, visit the reference pages for the `UIScrollView` class and for the `UIScrollViewDelegate` protocol.

To simulate two fingers in the simulator to test your zooming, hold down the Option key while using the mouse.

This page intentionally left blank

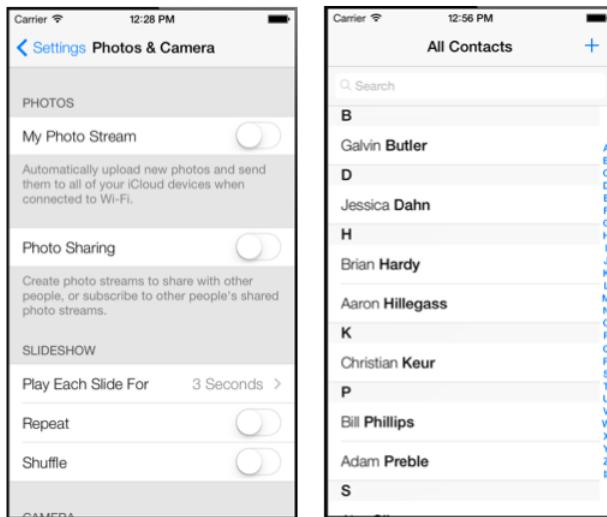
8

UITableView and UITableView Controller

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 8.1 shows some examples of **UITableView**.

Figure 8.1 Examples of **UITableView**

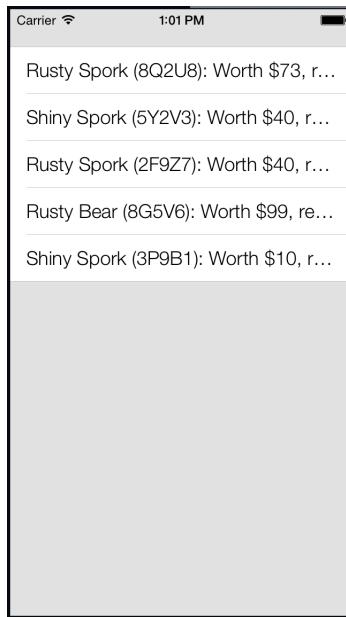


Beginning the Homepwner Application

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.)

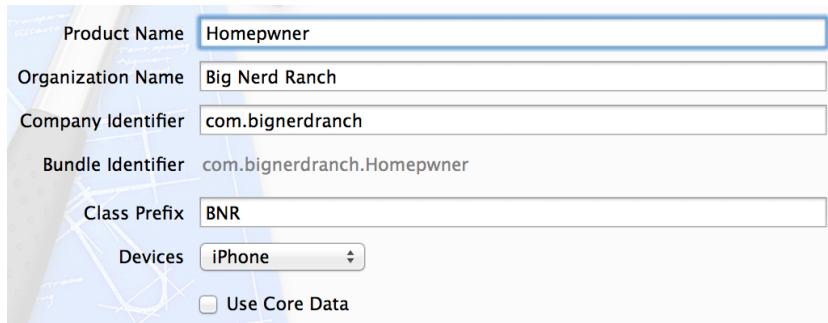
So far, your iOS projects have been small, but Homepwner will grow into a realistically complex application over the course of nine chapters. By the end of this chapter, Homepwner will present a list of **BNRItem** objects in a **UITableView**, as shown in Figure 8.2.

Figure 8.2 Homepwner: phase 1



Create a new iOS Empty Application project and configure it as shown in Figure 8.3.

Figure 8.3 Configuring Homepwner



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:

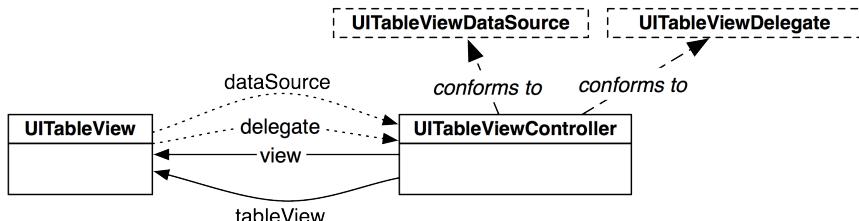
- A **UITableView** typically needs a view controller to handle its appearance on the screen.
- A **UITableView** needs a *data source*. A **UITableView** asks its data source for the number of rows to display, the data to be shown in those rows, and other tidbits that make a **UITableView** a useful user interface. Without a data source, a table view is just an empty container. The **dataSource** for a **UITableView** can be any type of Objective-C 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** instance variables of the **UITableView** are automatically set to point at the **UITableViewController** (Figure 8.4).

Figure 8.4 **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 **NSObject** template. From the File menu, select New and then File.... From the iOS section, select Cocoa Touch, choose Objective-C class, and click Next. Then, select **NSObject** from the pop-up menu and enter **BNRItemsViewController** as the name of the new class. Click Next and then click Create on the next sheet to save your class.

Open `BNRItemsViewController.h` and change its superclass:

```
#import <Foundation/Foundation.h>
@interface BNRItemsViewController : NSObject
#import <UIKit/UIKit.h>
@interface BNRItemsViewController : UITableViewController
```

The designated initializer of `UITableViewController` is `initWithStyle:`, which takes a constant that determines the style of the table view. There are two options: `UITableViewStylePlain` and `UITableViewStyleGrouped`. These looked quite different on iOS 6, but the differences are quite minor as of iOS 7.

You are changing the designated initializer to `init`. As such, you need to follow the two rules of initializers:

- Call the superclass's designated initializer from yours
- Override the superclass's designated initializer to call yours

Do both in `BNRItemsViewController.m`:

```
#import "BNRItemsViewController.h"
@implementation BNRItemsViewController
- (instancetype)init
{
    // Call the superclass's designated initializer
    self = [super initWithStyle:UITableViewStylePlain];
    return self;
}
- (instancetype)initWithStyle:(UITableViewStyle)style
{
    return [self init];
}
```

This will ensure that all instances of `BNRItemsViewController` use the `UITableViewStylePlain` style, no matter what initialization message is sent to them.

Open `BNRAppDelegate.m`. In `application:didFinishLaunchingWithOptions:`, create an instance of `BNRItemsViewController` and set it as the `rootViewController` of the window. Make sure to import the header file for `BNRItemsViewController` at the top of this file.

```
#import "BNRItemsViewController.h"

@implementation BNRAppDelegate

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch

    // Create a BNRItemsViewController
    BNRItemsViewController *itemsViewController =
        [[BNRItemsViewController alloc] init];

    // Place BNRItemsViewController's table view in the window hierarchy
    self.window.rootViewController = itemsViewController;

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

Build and run your application. You should see an empty screen, as shown in Figure 8.5 – but there is an empty table view there. 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 8.5 Empty **UITableview**



Your table view needs some rows to display. Remember the **BNRItem** class you wrote in Chapter 2? Now you are going to use that class again: each row of the table view will display an instance of **BNRItem**.

Locate the header and implementation files for **BNRItem** (**BNRItem.h** and **BNRItem.m**) in Finder and drag them onto Homepwner'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.

You will not need the `container` or `containedItem` properties ever again (they were just to demonstrate strong reference cycles), so delete them from **BNRItem.h**:

```
@property (nonatomic, strong) BNRItem *containedItem;
@property (nonatomic, weak) BNRItem *container;
```

Also delete the `setContainedItem:` method in **BNRItem.m**:

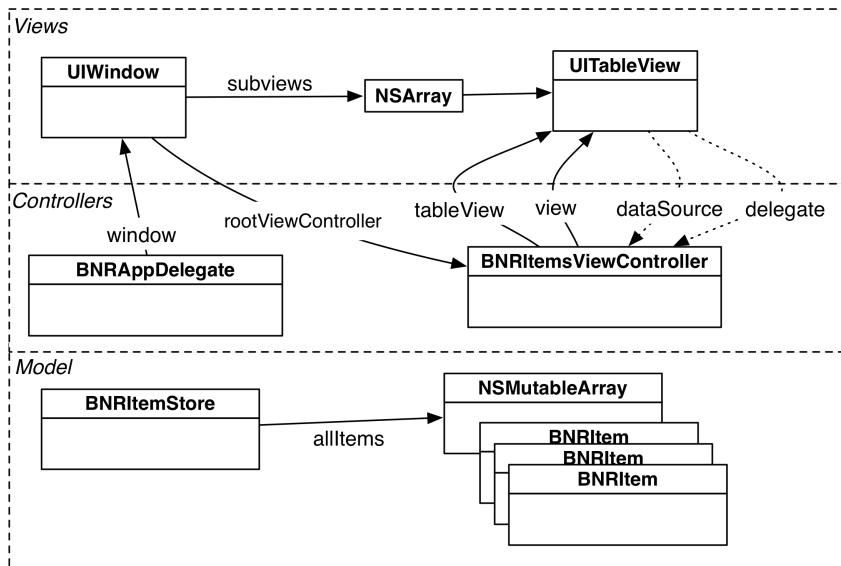
```
- (void)setContainedItem:(BNRItem *)i
{
    containedItem = i;
    // When given an item to contain, the contained
    // item will be given a pointer to its container
    self.containedItem.container = self;
}
```

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 **BNRItemsViewController** is the data source, so it needs a way to store item data.

In Chapter 2, you used an **NSMutableArray** to store **BNRItem** instances. You will do the same thing in this chapter, but with a little twist. The **NSMutableArray** that holds the **BNRItem** instances will be abstracted into another object – a **BNRItemStore** (Figure 8.6). Why not just use an array? Eventually, the **BNRItemStore** object will also take care of the saving and loading of the items.

Figure 8.6 Homepwner object diagram



If an object wants to see all of the items, it will ask the **BNRItemStore** 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 BNRItemStore

From the File menu, select New and then File.... Create a new **NSObject** subclass and name it **BNRItemStore**.

BNRItemStore will be a singleton. This means there will only be one instance of this type in the application; if you try to create another instance, the class will quietly return the existing instance instead. A singleton is useful when you have an object that many objects will talk to. Those objects can ask the singleton class for its one instance.

To get the (single instance of) **BNRItemStore**, you will send the **BNRItemStore** class the message **sharedStore**.

In **BNRItemStore.h**, declare **sharedStore**.

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

@interface BNRItemStore : NSObject

// Notice that this is a class method and prefixed with a + instead of a -
+ (instancetype)sharedStore;

@end
```

When this message is sent to the **BNRItemStore** class, the class will check to see if the single instance of **BNRItemStore** has already been created. If it has, the class will return the instance. If not, it will create the instance and return it.

In **BNRItemStore.m**, implement **sharedStore**, an **init** method that throws an exception, and a secret designated initializer named **initPrivate**.

```
@implementation BNRItemStore

+ (instancetype)sharedStore
{
    static BNRItemStore *sharedStore = nil;

    // Do I need to create a sharedStore?
    if (!sharedStore) {
        sharedStore = [[self alloc] initPrivate];
    }

    return sharedStore;
}

// If a programmer calls [[BNRItemStore alloc] init], let him
// know the error of his ways
- (instancetype)init
{
    @throw [NSError exceptionWithName:@"Singleton"
                                reason:@"Use +[BNRItemStore sharedStore]"
                                userInfo:nil];
    return nil;
}

// Here is the real (secret) initializer
- (instancetype)initPrivate
{
    self = [super init];

    return self;
}
```

Notice that the variable **sharedStore** is declared as **static**. A *static variable* is not destroyed when the method is done executing. Like a global variable, it is not kept on the stack.

The initial value of **sharedStore** is **nil**. The first time the **sharedStore** method is called, an instance of **BNRItemStore** will be created, and **sharedStore** will be set to point to it. In subsequent calls to this method, **sharedStore** will still point at that instance of **BNRItemStore**. This variable has a strong reference to the **BNRItemStore** and, since this variable will never be destroyed, the object it points to will never be destroyed either.

The **BNRItemsViewController** controller will send a message to the **BNRItemStore** when it wants a new **BNRItem** to be created. The **BNRItemStore** will oblige, create the object, and add it to an array of instances of **BNRItem**. The **BNRItemsViewController** will also ask the **BNRItemStore** for all of the items in the store when it wants to populate its **UITableView**.

In **BNRItemStore.h**, declare a method and property for these purposes.

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

@class BNRIItem;

@interface BNRIItemStore : NSObject

@property (nonatomic, readonly) NSArray *allItems;

+ (instancetype)sharedStore;
- (BNRIItem *)createItem;

@end
```

See the `@class` directive? That tells the compiler that there is a `BNRIItem` class and that the compiler does not need to know this class's details in the current file – only that it exists. This allows you to use the `BNRIItem` symbol in the declaration of `createItem` without importing `BNRIItem.h`. Using the `@class` directive can speed up compile times considerably because fewer files have to be recompiled when one file changes.

In files that actually send messages to the `BNRIItem` class or instances of it, you must import the file it was declared in so that the compiler will have all of its details. At the top of `BNRIItemStore.m`, import `BNRIItem.h`, since it will have to send messages to `BNRIItem` instances at some point.

```
#import "BNRIItemStore.h"
#import "BNRIItem.h"
```

Here is where things get a little interesting. You have a `BNRIItemStore` that is going to oversee the array of items – this includes adding items to the array and will later include removing them and reordering them. Because the `BNRIItemStore` wants this kind of control over the array, it returns an immutable `NSArray` to represent the array of items and declares the property as `readonly`. No other object can change the `BNRIItemStore`'s `allItems` property, either by giving it a new array or modifying the array it has.

However, internally, the `BNRIItemStore` needs to be able to mutate the array to add new items (and later remove and reorder them). This is a pretty common design for a class that wants strict control over its internal data: an object hangs onto a mutable data structure, but other objects only get access to an immutable version of it.

In `BNRIItemStore.m`, declare a mutable array in the class extension.

```
#import "BNRIItem.h"

@interface BNRIItemStore ()

@property (nonatomic) NSMutableArray *privateItems;

@end

@implementation BNRIItemStore
```

Implement `initPrivate` to immediately instantiate `privateItems`. Also, override the getter for `allItems` to return the `privateItems`.

```
- (instancetype)initPrivate
{
    self = [super init];
    if (self) {
        _privateItems = [[NSMutableArray alloc] init];
    }
    return self;
}

- (NSArray *)allItems
{
    return self.privateItems;
}
```

This is possible because `NSMutableArray` is a subclass of `NSArray`. Therefore, an `NSMutableArray` *is an* `NSArray` because it can do everything an `NSArray` can do. (Note that this would not work if the property returned an `NSMutableArray`, but the instance variable was an `NSArray`, because an `NSArray` cannot do everything its mutable counterpart can.)

There is one issue: even though the `allItems` property says it is returning an `NSArray`, you know that all Objective-C objects know their type and that the type of a variable or return value does not change that type. Thus, any object that sends `allItems` to the `BNRItemStore` will get an `NSMutableArray` instance back – even though the object may not know that.

When using a class, like `BNRItemStore`, you can only rely on what the interface file tells you when it comes to interacting with that class. If the interface file tells you that an object is an `NSArray`, you should treat it like an `NSArray`. To do otherwise would violate the contract that `BNRItemStore` specifies with its public interface. However, if you were being really cautious, you could override `allItems` to return an immutable copy of its `privateItems` property. You would write that code like this (but do not do this here, because you are going to rely on convention instead of rigid rules):

```
- (NSArray *)allItems
{
    return [self.privateItems copy];
}
```

With that discussion out of the way, implement `createItem` in `BNRItemStore.m`.

```
- (BNRItem *)createItem
{
    BNRItem *item = [BNRItem randomItem];
    [self.privateItems addObject:item];
    return item;
}
```

An interesting quirk of this example: There is no `_allItems` instance variable at all. You declared an `allItems` property, but then you implemented your own accessors (well, just the one: `allItems` was declared `readonly`). The compiler only auto-synthesizes an instance variable if you let it synthesize at least one accessor.

Implementing data source methods

In `BNRItemsViewController.m`, import `BNRItemStore.h` and `BNRItem.h` and update the designated initializer to add five random items to the `BNRItemStore`.

```
#import "BNRItemsViewController.h"
#import "BNRItemStore.h"
#import "BNRItem.h"

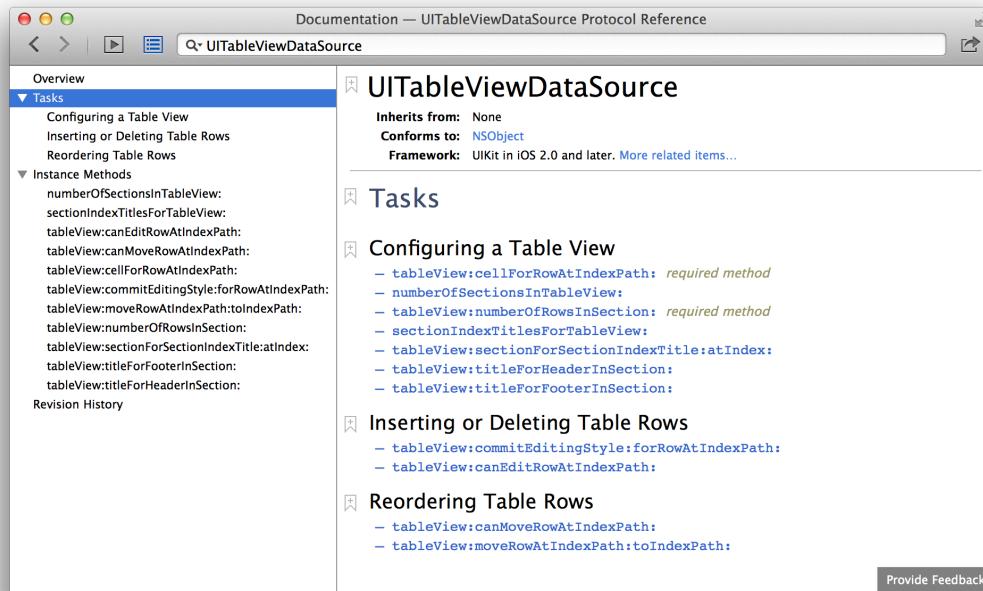
@implementation BNRItemsViewController

- (instancetype)init
{
    // Call the superclass's designated initializer
    self = [super initWithStyle:UITableViewStylePlain];
    if (self) {
        for (int i = 0; i < 5; i++) {
            [[BNRItemStore sharedStore] createItem];
        }
    }
    return self;
}
```

Now that there are some items in the store, you need to teach `BNRItemsViewController` 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 8.7).

Figure 8.7 `UITableViewDataSource` protocol documentation



In the Configuring a Table View task, notice the two methods marked ***required method***. For **BNRItemsViewController** 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 Homepwner, there should be a row for each entry in the store.

In **BNRItemsViewController.m**, implement **tableView:numberOfRowsInSection:**.

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return [[[BNRItemStore sharedStore] allItems] count];
}
```

Notice that this method returns an **NSInteger**. When Apple started supporting both **32-bit** and **64-bit** systems, they needed an integer type that was a **32-bit int** in **32-bit** applications and a **64-bit int** in **64-bit** applications. Thus, **NSInteger** (which is signed) and **NSUInteger** (which is unsigned) were born. These types are used extensively throughout Apple's frameworks.

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.

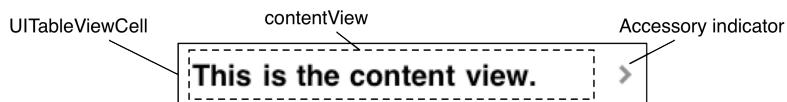
The second required method in the **UITableViewDataSource** protocol is **tableView:cellForRowAtIndexPath:**. To implement this method, you need to learn about another class – **UITableViewCell**.

UITableViewCell

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. In Chapter 19, you will create a custom subclass of **UITableViewCell**.

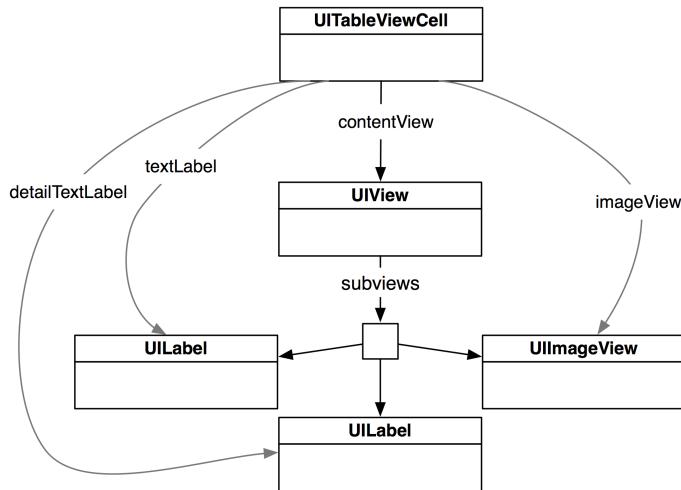
A cell itself has one subview – its **contentView** (Figure 8.8). The **contentView** is the superview for the content of the cell. The cell may also draw an accessory indicator. The accessory indicator shows an action-oriented icon, such as a checkmark, a disclosure icon, or a fancy blue dot with a chevron inside. 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 Chapter 19. (Curious now? See the reference page for **UITableViewCell** for more details.)

Figure 8.8 **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 8.9). In this chapter, you will only use **textLabel**.

Figure 8.9 **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 8.10.

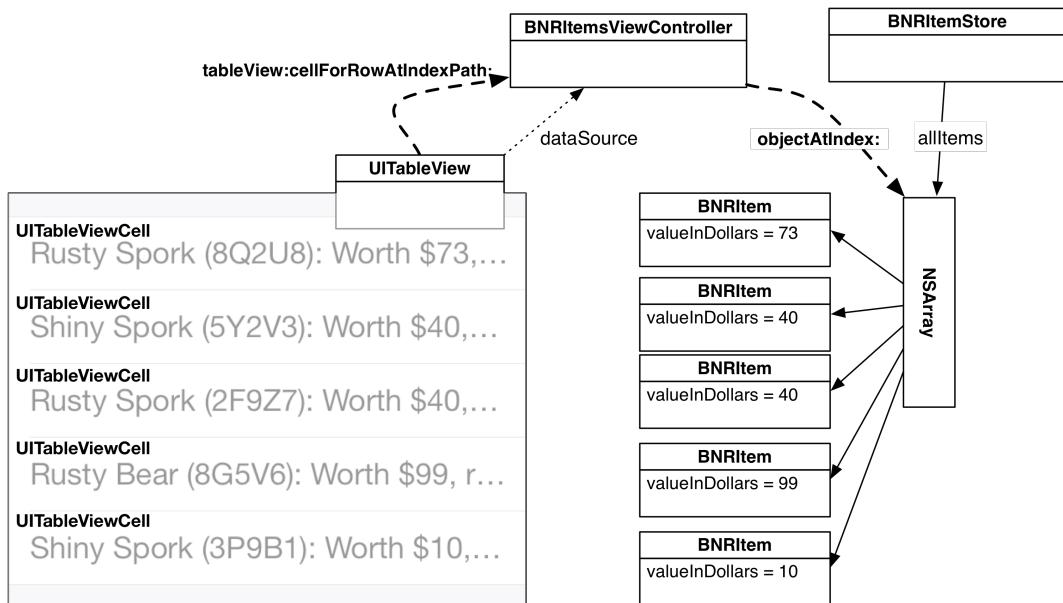
Figure 8.10 **UITableViewCellStyles**

| | | | |
|-------------------------------------|--|--|---|
| UITableViewCellStyleDefault | | textLabel | > |
| UITableViewCellStyleSubtitle | | textLabel detailTextLabel | > |
| UITableViewCellStyleValue1 | | textLabel detailTextLabel | > |
| UITableViewCellStyleValue2 | | textLabel detailTextLabel | > |

Creating and retrieving UITableViewCells

In this chapter, each cell will display the **description** of a **BNRItem** as its **textLabel**. To make this happen, you need to implement the second required method from the **UITableViewDataSource** protocol, **tableView:cellForRowAtIndexPath:**. This method will create a cell, set its **textLabel** to the **description** of the corresponding **BNRItem**, and return it to the **UITableView** (Figure 8.11).

Figure 8.11 UITableViewCell retrieval



How do you decide which cell a **BNRItem** corresponds to? One of the parameters sent to **tableView:cellForRowAtIndexPath:** 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 **BNRItemsViewController.m**, implement **tableView:cellForRowAtIndexPath:** so that the *n*th row displays the *n*th entry in the **allItems** array.

```

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Create an instance of UITableViewCell, with default appearance
    UITableViewCell *cell =
        [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
            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
    NSArray *items = [[BNRItemStore sharedStore] allItems];
    BNRItem *item = items[indexPath.row];

    cell.textLabel.text = [item description];

    return cell;
}

```

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

Think back to your RandomItems project from Chapter 3. You created the **BNRItem** class, created instances of **BNRItem** (model objects), and printed their data to the console.

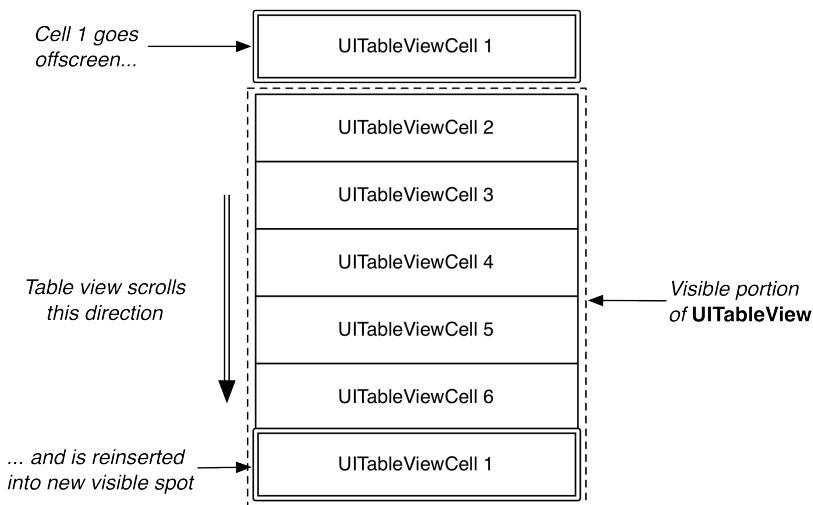
Now you are reusing that class with a different controller and having the controller interface with a different view. You did not have to change anything about **BNRItem**, but you can now show its data in an entirely different way. This is the Model-View-Controller pattern in action. If you design your classes and applications according to MVC, then it is far simpler to reuse those classes in different contexts.

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 8.12 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 **NSString**. 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.

In `BNRItemsViewController.m`, update `tableView:cellForRowAtIndexPath:` to reuse cells:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell =
        [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:@"UITableViewCell"];
    // Get a new or recycled cell
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:@"UITableViewCell"
            forIndexPath:indexPath];
    NSArray *items = [[BNRItemStore sharedStore] allItems];
    BNRItem *item = items[indexPath.row];
    cell.textLabel.text = [item description];
    return cell;
}
```

Previously, you created the table view cell explicitly, but now you are giving that control to Apple to get the benefits of the reuse identifier. For this to work, you need to tell the table view which kind of cell it should instantiate if there are no cells in the reuse pool.

In `BNRItemsViewController.m`, override `viewDidLoad` to register `UITableViewCell` class with the table view.

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    [self.tableView registerClass:[UITableViewCell class]
        forCellReuseIdentifier:@"UITableViewCell"];
}
```

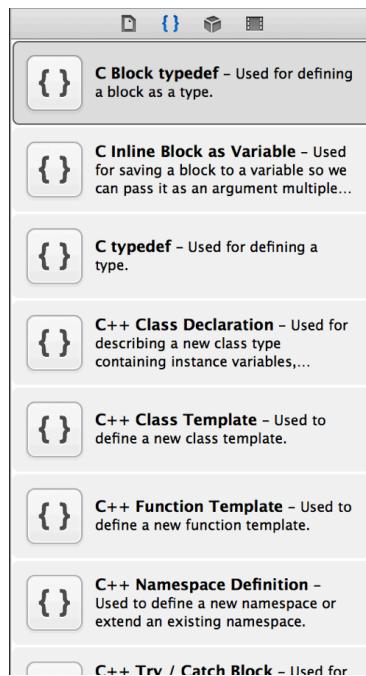
Reusing cells means that you only have to create a handful of cells, which puts fewer demands on memory. Your application’s users (and their devices) will thank you. Build and run the application. The behavior of the application should remain the same.

Code Snippet Library

You may have noticed that when you start typing the method definition for `init` in an implementation file, Xcode will automatically add an `init` implementation in your source file. If you have not noticed this, go ahead and type `init` in an implementation file and wait for the code-completion to kick in.

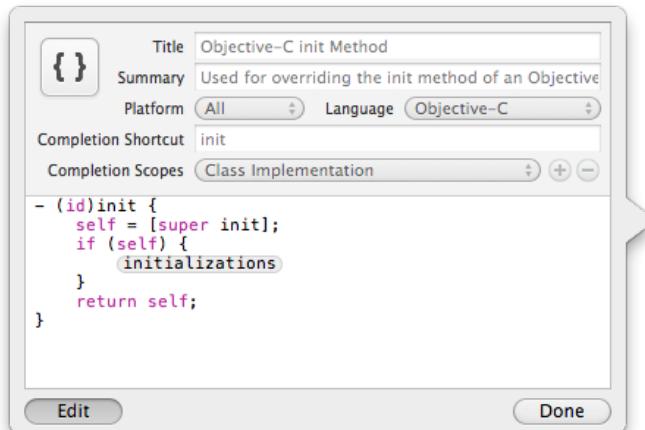
The freebie code comes from the *code snippet library*. You can see the code snippet library by opening the utility area and selecting the {} tab in the library selector (Figure 8.13). Alternatively, you can use the shortcut **Command-Control-Option-2**, which reveals the utility area and the Code Snippet Library. Substituting another number in the shortcut selects the corresponding library.

Figure 8.13 Code snippet library



Notice that there are a number of code snippets available (Figure 8.13). Click on one and, in a moment, a window will appear with the details for that snippet. Click the **Edit** button on the code snippet detail window (Figure 8.14).

Figure 8.14 Snippet editing window



The Completion Shortcut field in the edit window shows you what to type in a source file to have Xcode add the snippet. This window also tells you that this snippet can be used in an Objective-C file as long as you are in the scope of a class implementation.

You cannot edit any of the pre-defined code snippets, but you can create your own. In `BNRItemsViewController.m`, locate the implementation of `tableView:numberOfRowsInSection:`. Highlight the entire method:

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return [[[BNRItemStore sharedStore] allItems] count];
}
```

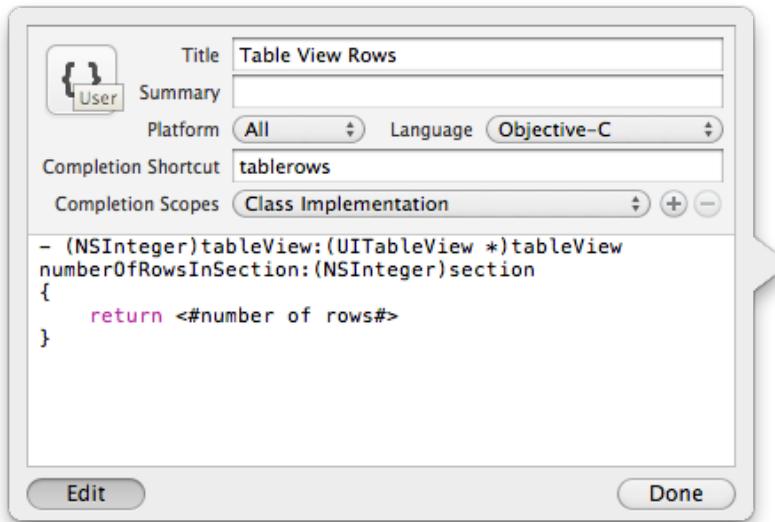
Drag this highlighted code into the code snippet library. The edit window will appear again, allowing you to fill out the details for this snippet.

One issue with this snippet is that the return statement is really specific to this application – it would be much more useful if the value returned was a code completion placeholder that you could fill in easily. In the edit window, modify the code snippet so it looks like this:

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return <#number of rows#>;
}
```

Then fill out the rest of the fields in the edit window as shown in Figure 8.15 and click Done.

Figure 8.15 Creating a new snippet



In `BNRItemsViewController.m`, start typing `tablerows`. Xcode will recommend this code snippet and pressing the return key will automatically complete it for you – and the `number of rows` placeholder will be selected. (If there are several placeholders, Control-/ will hop you to the next.)

Before continuing, make sure to remove the code entered by the snippet because you have already defined `tableView:numberOfRowsInSection:` in `BNRItemsViewController.m`.

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

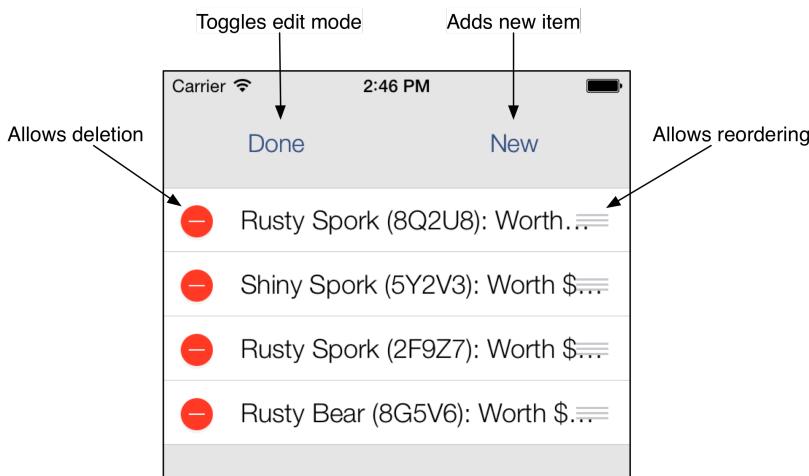
Make each row's height 60 points except for the last row from the silver challenge, which should remain 44 points. Then, change the font size of every row except the last to 20 points. Finally, make the background of the `UITableView` display an image. (To make this pixel-perfect, you will need an image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

This page intentionally left blank

Editing UITableView

In the last chapter, you created an application that displays a list of `BNRItem` 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 9.1 shows what Homepwner will look like by the end of this chapter.

Figure 9.1 Homepwner in editing mode



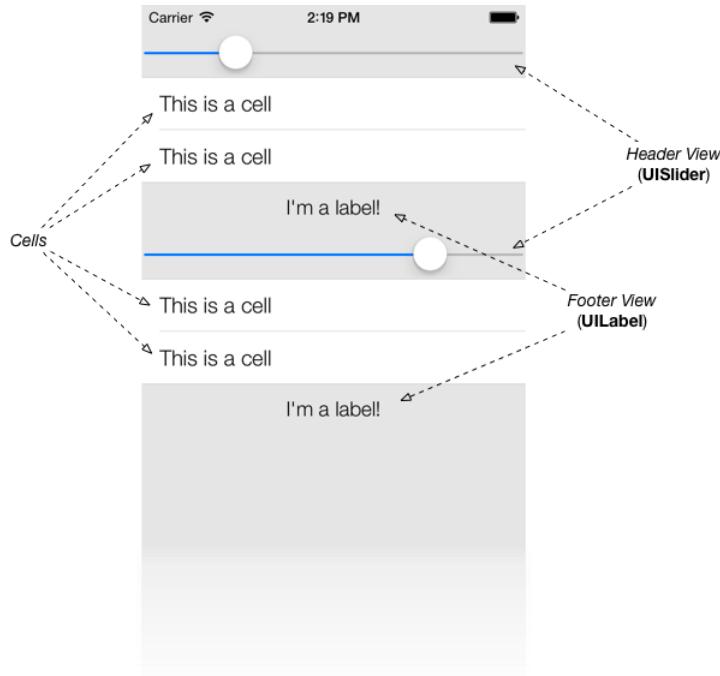
Editing Mode

`UITableView` has an `editing` property, and when this property is set to YES, 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 9.2 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 **BNRItem** 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 **BNRItemsViewController** 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 **BNRItemsViewController.m**, add a class extension with the following property. Also stub out two methods in the implementation.

```

@interface BNRItemsViewController : UITableViewController

@property (nonatomic, strong) IBOutlet UIView *headerView;

@end

@implementation BNRItemsViewController

// Other methods here

- (IBAction)addNewItem:(id)sender
{
}

- (IBAction)toggleEditMode:(id)sender
{
}

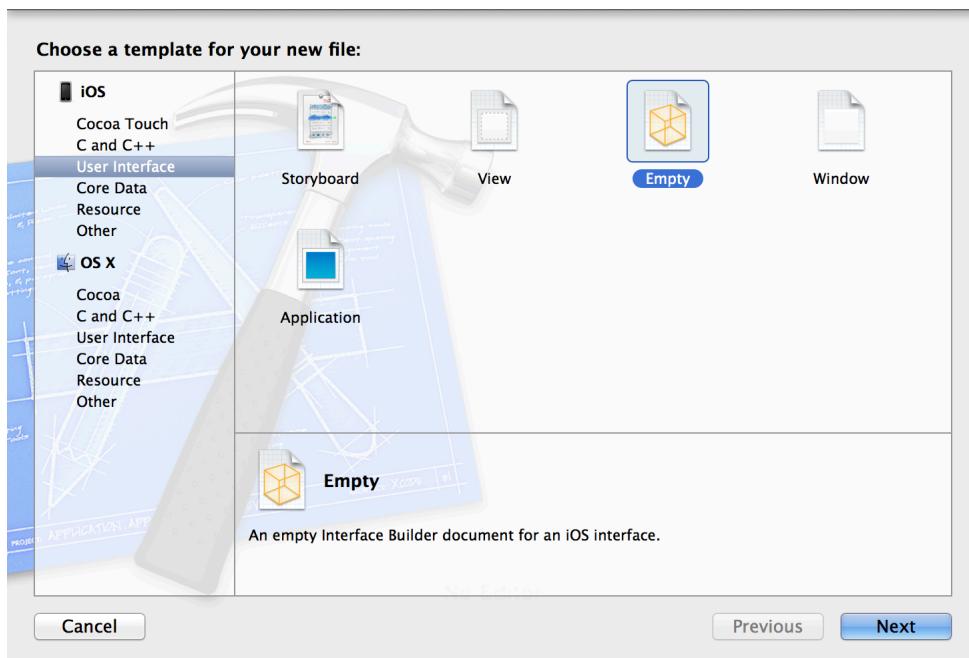
```

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.

Now you need to create the new XIB file. Unlike the previous XIB files you created, this XIB file will not deal with the view controller's view. (As a subclass of `UITableViewController`, `BNRItemsViewController` already knows how to create its view.) XIB files are typically used to create the view for a view controller, but they can also be used any time you want to lay out view objects, archive them, and have them loaded at runtime.

Create a new file (Command-N). From the iOS section, select User Interface, choose the Empty template, and click Next (Figure 9.3).

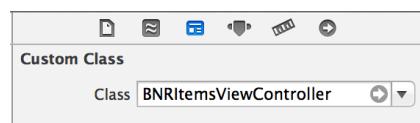
Figure 9.3 Creating a new XIB file



On the next pane, select iPhone. Save this file as `HeaderView`.

In `HeaderView.xib`, select the File's Owner object and change its Class to `BNRItemsViewController` in the identity inspector (Figure 9.4).

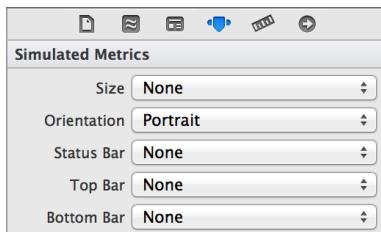
Figure 9.4 Changing the File's Owner



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

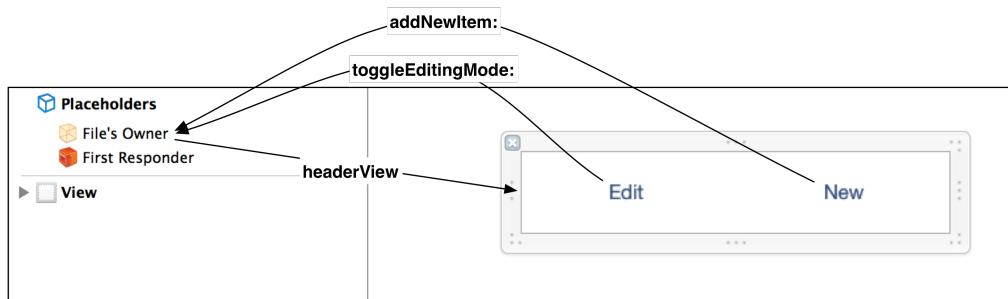
locked. To unlock the size, select the **UIView** on the canvas and open the attributes inspector. Under the Simulated Metrics section, select None for the Size option (Figure 9.5).

Figure 9.5 Unlocking a view's size



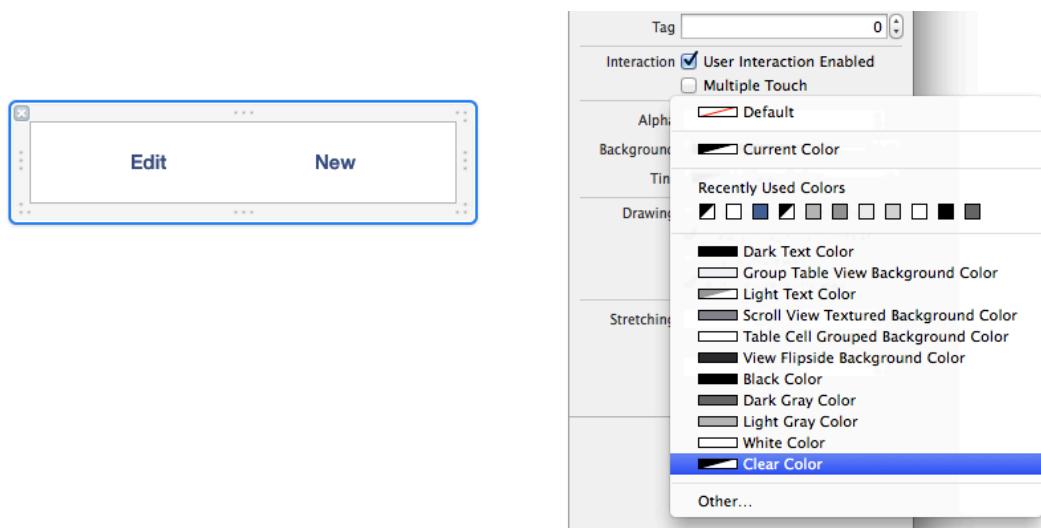
Now that the view can be resized, resize it and make the connections shown in Figure 9.6.

Figure 9.6 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 9.7).

Figure 9.7 Setting background color to clear



So far, your XIB files have been loaded automatically by the implementation of **UIViewController**. For example, **BNRReminderViewController** in Chapter 6 knew how to load **BNRReminderViewController.xib** because of code written in its superclass, **UIViewController**. For **HeaderView.xib**, you are going to write the code to have the **BNRItemsViewController** 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 message **mainBundle** to **NSBundle**.

Once you have a pointer to the main bundle object, you can ask it to load a XIB file. In **BNRItemsViewController.m**, implement **headerView**.

```
- (UIView *)headerView
{
    // If you have not loaded the headerView yet...
    if (![_headerView]) {

        // Load HeaderView.xib
        [[NSBundle mainBundle] loadNibNamed:@"HeaderView"
                                         owner:self
                                         options:nil];
    }

    return _headerView;
}
```

Notice that this is a **getter method** that does more than just get. This is a common pattern: *Lazy Instantiation* puts off creating the object until it is actually needed. In some cases this approach can significantly lower the normal memory footprint of your app.

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 **BNRItemsViewController** instance.

The first time the **headerView** message is sent to the **BNRItemsViewController**, it will load **HeaderView.xib** and keep a pointer to the view object in the instance variable **headerView**. The buttons in this view will send messages to the **BNRItemsViewController** when tapped.

Now you just need to tell the table view about its header view. In **BNRItemsViewController.m**, add this to the **viewDidLoad** method:

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    [self.tableView registerClass:[UITableViewCell class]
                           forCellReuseIdentifier:@"UITableViewCell"];

    UIView *header = self.headerView;
    [self.tableView setTableHeaderView:header];
}
```

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, `BNRReminderViewController.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 `loadNibNamed: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:

```
- (void)loadView
{
    // Which bundle is the NIB in?
    // Was a bundle passed to initWithNibName:bundle:?
    NSBundle *bundle = [self nibBundle];
    if (!bundle) {

        // Use the default
        bundle = [NSBundle mainBundle];
    }

    // What is the NIB named?
    // Was a name passed to initWithNibName:bundle:?
    NSString *nibName = [self nibName];
    if (!nibName) {

        // Use the default
        nibName = NSStringFromClass([self class]);
    }

    // Try to find the NIB in the bundle
    NSString *nibPath = [bundle pathForResource:nibName
                                    ofType:@"nib"];

    // Does it exist?
    if (nibPath) {

        // 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
        self.view = [[UIView alloc] init];
    }
}
```

Now let's implement the `toggleEditingMode:` 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 `BNRItemsViewController.m`, implement `toggleEditingMode:`.

```

- (IBAction)toggleEditingStyle:(id)sender
{
    // If you are currently in editing mode...
    if (self.isEditing) {

        // Change text of button to inform user of state
        [sender setTitle:@"Edit" forState:UIControlStateNormal];

        // Turn off editing mode
        [self setEditing:NO animated:YES];
    } else {

        // Change text of button to inform user of state
        [sender setTitle:@"Done" forState:UIControlStateNormal];

        // Enter editing mode
        [self setEditing:YES animated:YES];
    }
}

```

Build and run your application, tap the Edit button, and the **UITableView** will enter editing mode (Figure 9.8).

Figure 9.8 **UITableView** in editing mode

| Done | New |
|-----------------------------------|-----|
| – Rusty Spork (8Q2U8): Worth... | |
| – Shiny Spork (5Y2V3): Worth... | |
| – Rusty Spork (2F9Z7): Worth... | |
| – Rusty Bear (8G5V6): Worth \$... | |
| – Shiny Spork (3P9B1): Worth... | |

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 `BNRItemsViewController.m`, implement `addNewItem:`.

```
- (IBAction)addItem:(id)sender
{
    // Make a new index path for the 0th section, last row
    NSInteger lastRow = [self.tableView numberOfRowsInSection:0];
    NSIndexPath *indexPath = [NSIndexPath indexPathForRow:lastRow inSection:0];

    // Insert this new row into the table.
    [self.tableView insertRowsAtIndexPaths:@[indexPath]
        withRowAnimation:UITableViewRowAnimationTop];
}
```

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. `BNRItemsViewController` 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 `BNRItem` to the `BNRItemStore` before you insert the new row.

In `BNRItemsViewController.m`, update `addNewItem:`.

```
- (IBAction)addItem:(id)sender
{
    NSInteger lastRow = [[self tableView] numberOfRowsInSection:0];

    // Create a new BNRItem and add it to the store
    BNRItem *newItem = [[BNRItemStore sharedStore] createItem];

    // Figure out where that item is in the array
    NSInteger lastRow = [[[BNRItemStore sharedStore] allItems] indexOfObject:newItem];

    NSIndexPath *indexPath = [NSIndexPath indexPathForRow:lastRow inSection:0];

    // Insert this new row into the table
    [self.tableView insertRowsAtIndexPaths:@[indexPath]
        withRowAnimation:UITableViewRowAnimationTop];
}
```

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 sending the message `tableView` to the `BNRItemsViewController` to get at the table view. This method is inherited from `UIViewController`, and it returns the controller’s table view. While you can send the message `view` to an instance of `UIViewController` and get a pointer to the same object, using `tableView` tells the compiler that the object returned will be an instance of class `UITableView`. Thus, sending a message that is specific to `UITableView`, like `insertRowsAtIndexPaths:withRowAnimation:`, will not generate a warning.

Now that you have the ability to add rows and items, remove the code in the `init` method in `BNRItemsViewController.m` that puts five random items into the store.

```

- (instancetype)init
{
    // Call the superclass's designated initializer
    self = [super initWithStyle:UITableViewStylePlain];
    if (self) {
        for (int i = 0; i < 5; i++) {
            [[BNRItemStore sharedStore] createItem];
        }
    }
    return self;
}

```

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 9.8) 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 sends its data source a message about the proposed deletion and waits for a confirmation message before pulling the trigger.

When deleting a cell, you must do two things: remove the row from the **UITableView** and remove the **BNRItem** associated with it from the **BNRItemStore**. To pull this off, the **BNRItemStore** must know how to remove objects from itself.

In **BNRItemStore.h**, declare a new method.

```

@interface BNRItemStore : NSObject
+ (BNRItemStore *)sharedStore;

@property (nonatomic, strong, readonly) NSArray *allItems;

- (BNRItem *)createItem;
- (void)removeItem:(BNRItem *)item;

@end

```

In **BNRItemStore.m**, implement **removeItem:**.

```

- (void)removeItem:(BNRItem *)item
{
    [self.privateItems removeObjectIdenticalTo:item];
}

```

You could use **NSMutableArray**'s **removeObject:** method here instead of **removeObjectIdenticalTo:**, but consider the difference: **removeObject:** goes to each object in the array and sends it the message **isEqual:**. A class can implement this method to return YES or NO based on its own determination. For example, two **BNRItem** objects could be considered equal if they had the same **valueInDollars**.

The method **removeObjectIdenticalTo:**, on the other hand, removes an object if and only if it is the exact same object as the one passed in this message. While **BNRItem** does not currently

override **isEqual:** to do special checking, it could in the future. Therefore, you should use **removeObjectIdenticalTo:** when you are specifying a particular instance.

Now you will implement **tableView:commitEditingStyle:forRowAtIndexPath:**, a method from the **UITableViewDataSource** protocol. (This message is sent to the **BNRItemsViewController**. Keep in mind that while the **BNRItemStore** is where the data is kept, the **BNRItemsViewController** is the table view's **dataSource**.)

When **tableView:commitEditingStyle:forRowAtIndexPath:** is sent to the data source, two extra arguments are passed along with it. The first is the **UITableViewCellEditingStyle**, which, in this case, is **UITableViewCellEditingStyleDelete**. The other argument is the **NSIndexPath** of the row in the table.

In **BNRItemsViewController.m**, implement this method to have the **BNRItemStore** remove the right object and to confirm the row deletion by sending the message **deleteRowsAtIndexPaths:withRowAnimation:** back to the table view.

```
- (void)tableView:(UITableView *)tableView
    commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
    forRowAtIndexPath:(NSIndexPath *)indexPath
{
    // If the table view is asking to commit a delete command...
    if (editingStyle == UITableViewCellEditingStyleDelete) {
        NSArray *items = [[BNRItemStore sharedStore] allItems];
        BNRItem *item = items[indexPath.row];
        [[BNRItemStore sharedStore] removeItem:item];

        // Also remove that row from the table view with an animation
        [tableView deleteRowsAtIndexPaths:@[indexPath]
                           withRowAnimation:UITableViewRowAnimationFade];
    }
}
```

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 send the message **deleteRowsAtIndexPaths:withRowAnimation:** to 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 sending the message **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 **BNRItemStore** a method to change the order of items in its **allItems** array. In **BNRItemStore.h**, declare this method.

```
- (void)moveItemAtIndex:(NSUInteger)fromIndex
                  toIndex:(NSUInteger)toIndex;
```

In **BNRItemStore.m**, implement **moveItemAtIndex:toIndex:**.

```

- (void)moveItemAtIndexPath:(NSIndexPath *)fromIndexPath
    toIndexPath:(NSIndexPath *)toIndexPath
{
    if (fromIndexPath == toIndexPath) {
        return;
    }
    // Get pointer to object being moved so you can re-insert it
    BNRIItem *item = self.privateItems[fromIndexPath];

    // Remove item from array
    [self.privateItems removeObjectAtIndex:fromIndexPath];

    // Insert item in array at new location
    [self.privateItems insertObject:item atIndex:toIndexPath];
}

```

In BNRItemsViewController.m, implement `tableView:moveRowAtIndexPath:toIndexPath:` to update the store.

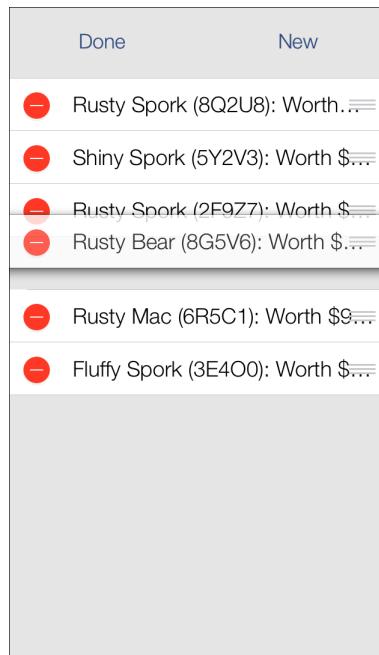
```

- (void)tableView:(UITableView *)tableView
    moveRowAtIndexPath:(NSIndexPath *)sourceIndexPath
    toIndexPath:(NSIndexPath *)destinationIndexPath
{
    [[BNRItemStore sharedStore] moveItemAtIndexPath: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 9.9).

Figure 9.9 Moving a row



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.

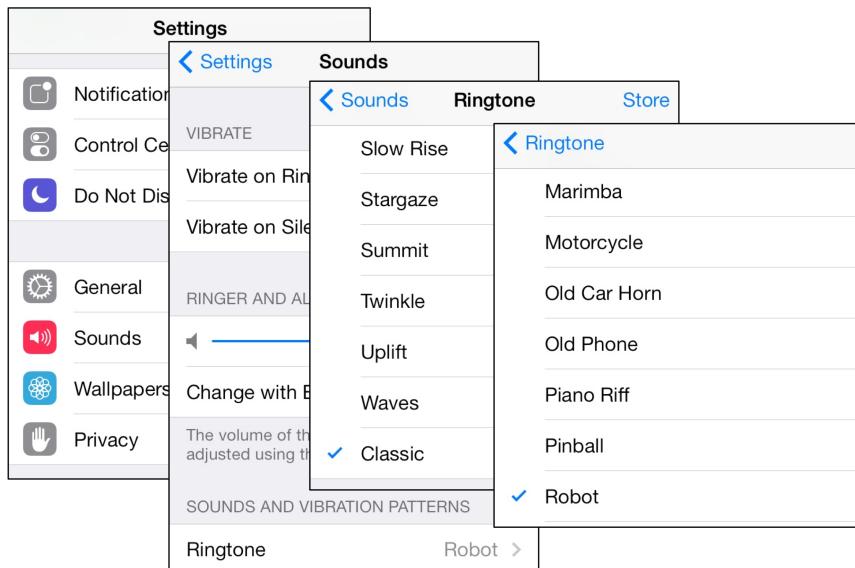
10

UINavigationController

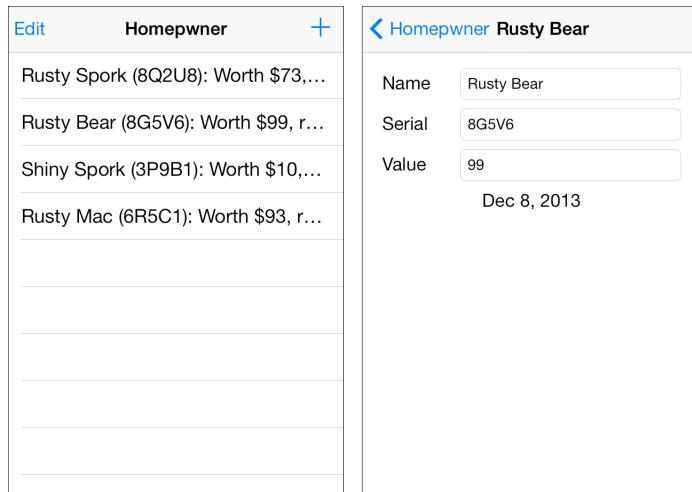
In Chapter 6, you learned about **UITabBarController** and how it allows a user to access different screens. A tab bar controller is great when you have screens that do not rely on each other, but what if you want to move between related screens?

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 10.1 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 a **BNRItem** (Figure 10.2).

Figure 10.2 Homeowner with **UINavigationController**

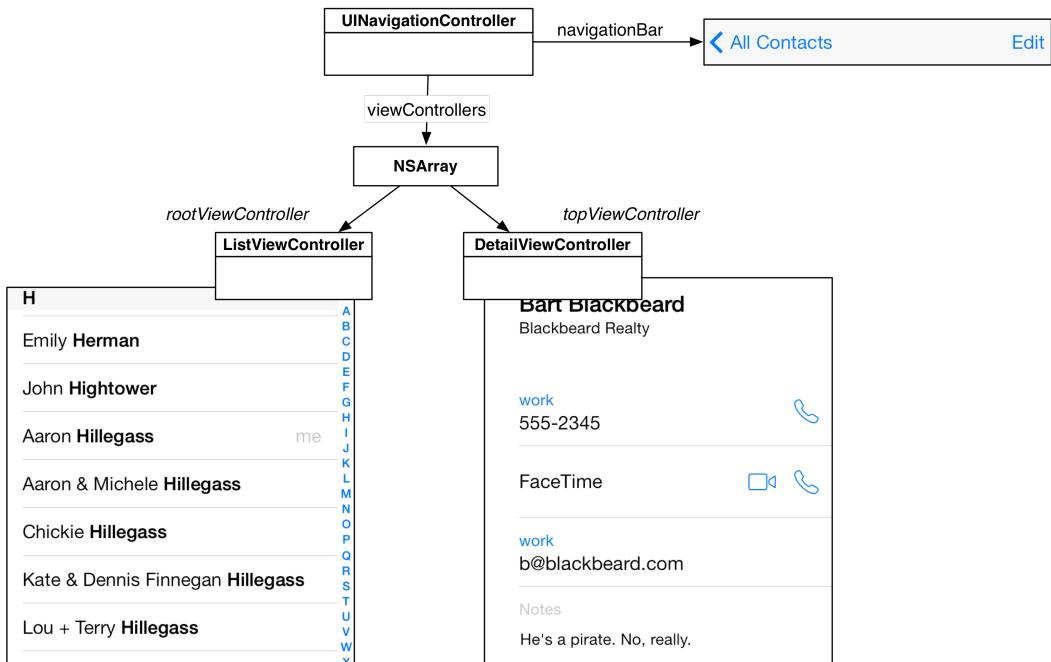
UINavigationController

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.

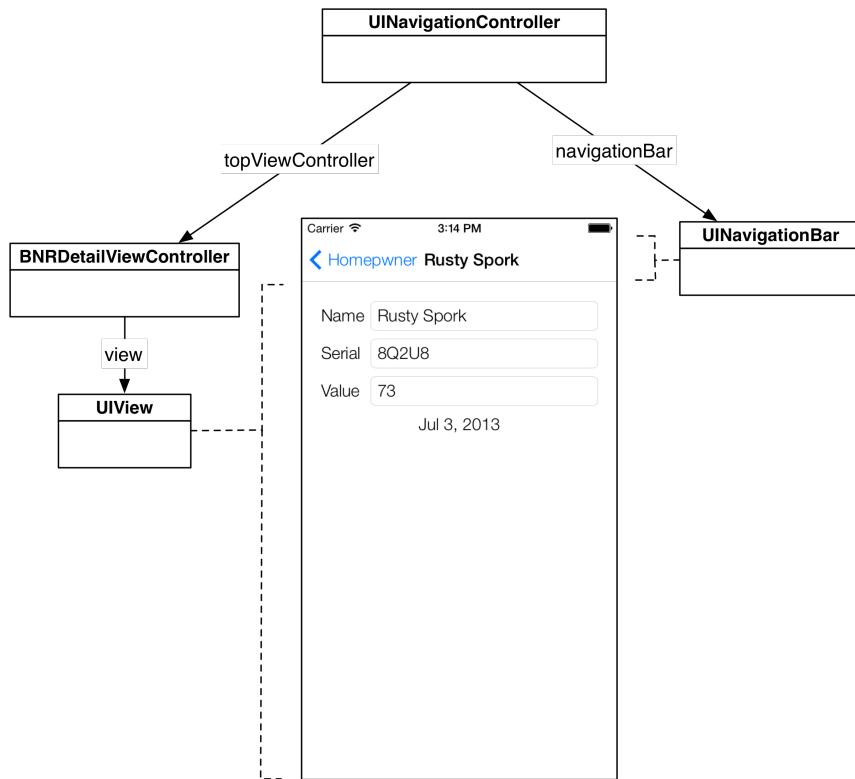
When a **UIViewController** is pushed onto the stack, its view slides onto the screen from the right. When the stack is popped, the top view controller is removed from the stack and its view slides off to the right, exposing the view of next view controller on the stack.

Figure 10.3 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 10.3 **UINavigationController**'s stack

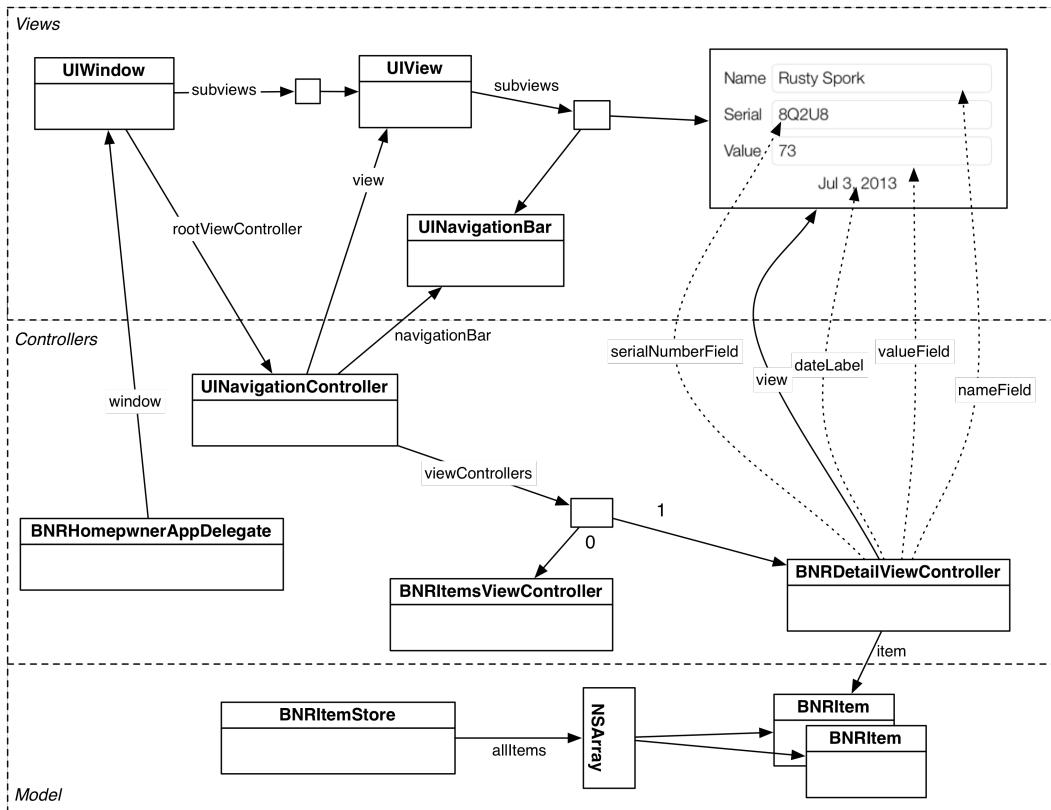
Like **UITabBarController**, **UINavigationController** has a `viewControllers` array. The root view controller is the first object in the array. As more view controllers are pushed onto the stack, they are added to the end of this array. Thus, the last view controller in the array is the top of the 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 10.4). You can set a navigation controller as the `rootViewController` of the window to make its `view` a subview of the window.

Figure 10.4 A **UINavigationController**'s view

In this chapter, you will add a **UINavigationController** to the Homepwner application and make the **BNRItemsViewController** 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 **BNRItem**. The object diagram for the updated Homepwner application is shown in Figure 10.5.

Figure 10.5 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 **BNRAppDelegate.m**. The only requirements for using a **UINavigationController** are that you give it a root view controller and add its view to the window.

In **BNRAppDelegate.m**, 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.

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:UIScreen.mainScreen.bounds]];
    // Override point for customization after application launch

    BNRItemsViewController *itemsViewController
        = [[BNRItemsViewController alloc] init];

    // Create an instance of a UINavigationController
    // its stack contains only itemsViewController
    UINavigationController *navController = [[UINavigationController alloc]
        initWithRootViewController:itemsViewController];

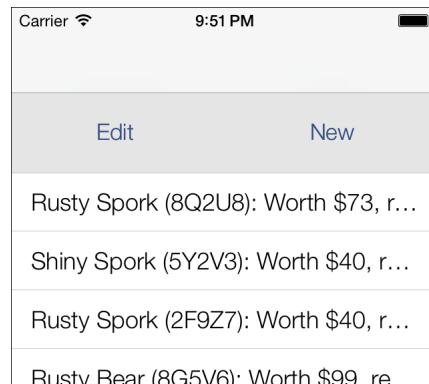
    self.window.rootViewController = itemsViewController;
    // Place navigation controller's view in the window hierarchy
    self.window.rootViewController = navController;

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];

    return YES;
}
```

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 10.6). Notice how **BNRItemsViewController**'s view was resized to fit the screen with a navigation bar. **UINavigationController** did this for you.

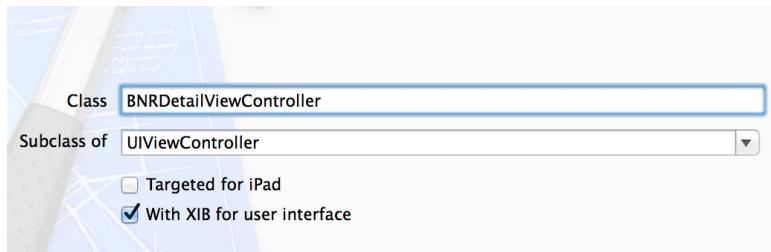
Figure 10.6 Homepwner with an empty navigation bar



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 Objective-C class (File → New → File...). Name this class **BNRDetailViewController** and choose **UIViewController** as the superclass. Check the With XIB for user interface box (Figure 10.7).

Figure 10.7 Create **UIViewController** subclass with XIB



In `BNRDetailViewController.m`, delete all of the code between the `@implementation` and `@end` directives so that the file looks like this:

```
#import "BNRDetailViewController.h"

@interface BNRDetailViewController : UIViewController

@end

@implementation BNRDetailViewController

@end
```

In `Homewner`, you want the user to be able to tap an item to get another screen with editable text fields for each property of that `BNRItem`. This view will be controlled by an instance of `BNRDetailViewController`.

The detail view needs four subviews – one for each instance variable of a `BNRItem` instance. And because you need to be able to access these subviews during runtime, `BNRDetailViewController` needs outlets for these subviews. The plan is to add four new outlets to `BNRDetailViewController`, drag the subviews onto the view in the XIB file, and then make the connections.

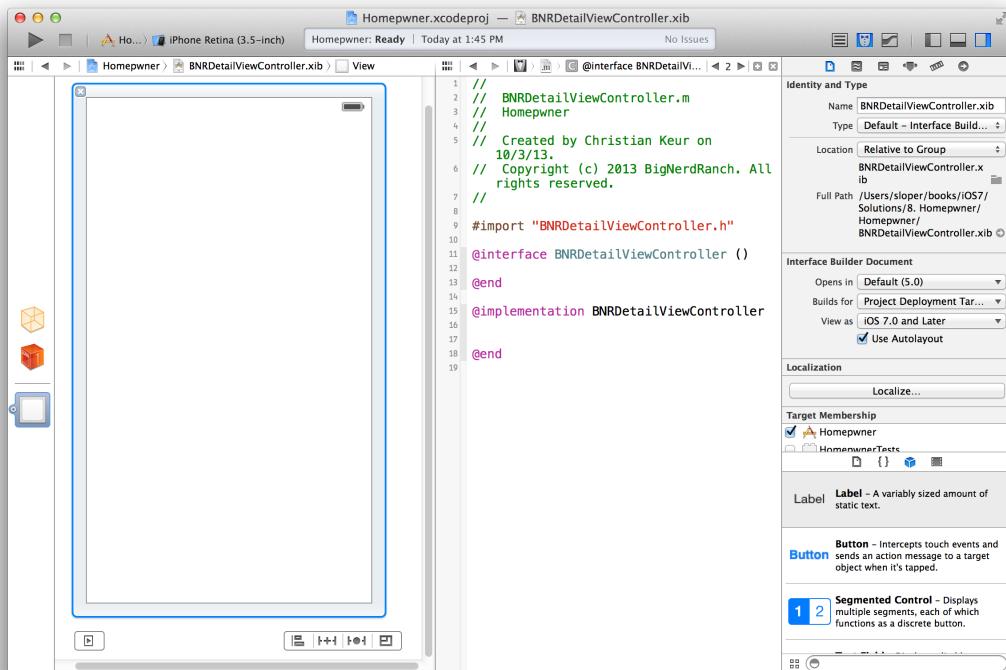
In previous exercises, these were three distinct steps: you added the outlets in the interface file, then you configured the interface in the XIB file, and then you made connections. You can combine these steps using a shortcut in Xcode. First, open `BNRDetailViewController.xib` by selecting it in the project navigator.

Now, Option-click on `BNRDetailViewController.m` in the project navigator. This shortcut opens the file in the *assistant editor*, right next to `BNRDetailViewController.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).

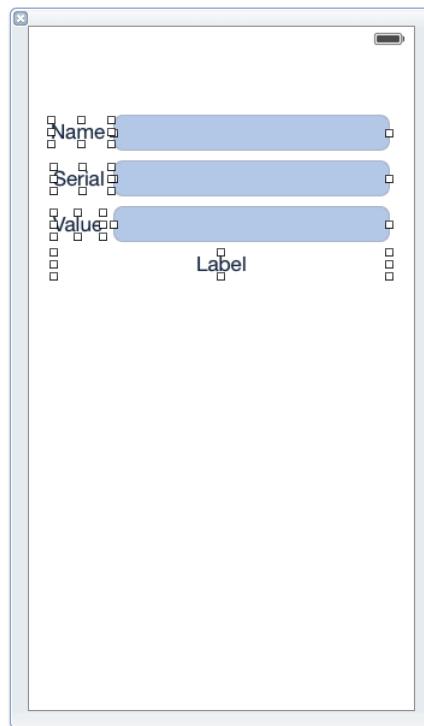
Your window is now sufficiently cluttered. Let's make some temporary space. Hide the navigator area by clicking the left button in the View control at the top of the workspace (the shortcut for this is Command-0). Then, change the dock in Interface Builder to show the icon view by clicking the toggle button in the lower left corner of the editor. Your workspace should now look like Figure 10.8.

Figure 10.8 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 10.9.

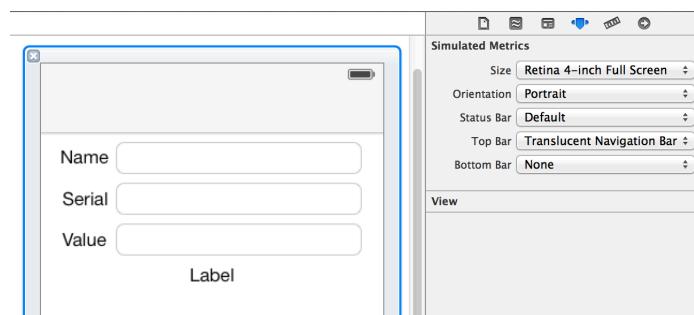
Figure 10.9 Configured **BNRDetailViewController** XIB



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.

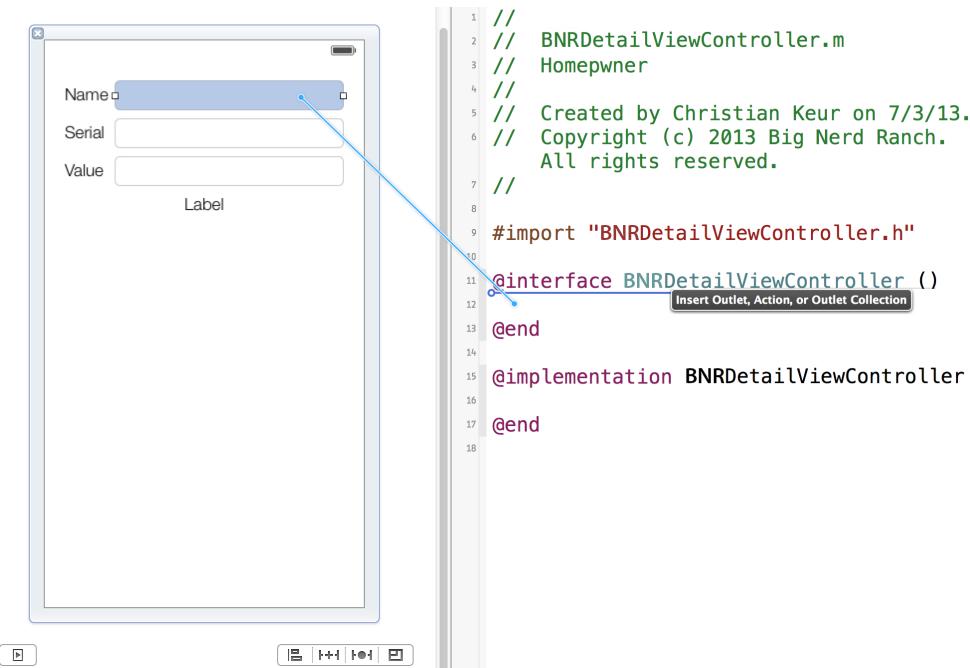
To see the simulated metrics, select the root level view, and open its attributes inspector. At the top, you will see **Simulated Metrics**. For Top Bar, choose Translucent Navigation Bar (Figure 10.10).

Figure 10.10 Simulated metrics



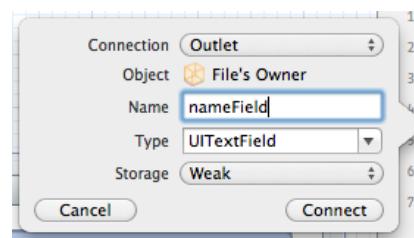
The three instances of **UITextField** and bottom instance of **UILabel** will be outlets in **BNRDetailViewController**. Here comes the exciting part. Control-drag from the **UITextField** next to the Name label to the class extension in **BNRDetailViewController.m**, as shown in Figure 10.11.

Figure 10.11 Dragging from XIB to source file



Let go while still inside the class extension, 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 10.12).

Figure 10.12 Auto-generating an outlet and making a connection



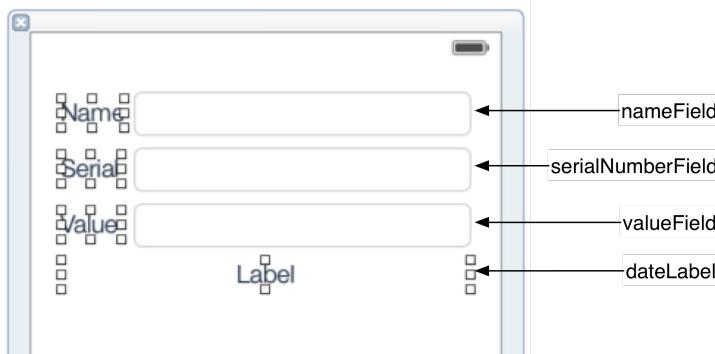
This will create an **IBOutlet** property of type **UITextField** named `nameField` in **BNRDetailViewController**. You chose Weak storage for this property because the object it will point to is not a top-level object in the XIB file.

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 10.13.

Figure 10.13 Connection diagram



After making the connections, `BNRDetailViewController.m` should look like this:

```
#import "BNRDetailViewController.h"

@interface BNRDetailViewController ()

@property (weak, nonatomic) IBOutlet UITextField *nameField;
@property (weak, nonatomic) IBOutlet UITextField *serialNumberField;
@property (weak, nonatomic) IBOutlet UITextField *valueField;
@property (weak, nonatomic) IBOutlet UILabel *dateLabel;

@end

@implementation BNRDetailViewController

@end
```

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 `BNRDetailViewController.m`. 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 `BNRDetailViewController.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.)

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 **BNRItemsViewController**'s table view and have the **BNRDetailViewController**'s view slide onto the screen and display the properties of the selected **BNRItem** instance.

Pushing view controllers

Of course, you need to create an instance of **BNRDetailViewController**. 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 6, 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 **BNRDetailViewController** and be responsible for adding it to the stack.

This object must meet two requirements: it needs to know when to push a **BNRDetailViewController** onto the stack, and it needs a pointer to the navigation controller to send the navigation controller messages, namely, **pushViewController:animated:**.

BNRItemsViewController fills both requirements. First, it knows when a row is tapped in a table view because, as the table view's delegate, it receives the message **tableView:didSelectRowAtIndexPath:** when this event occurs. Second, any view controller in a navigation controller's stack can get a pointer to that navigation controller by sending itself the message **navigationController**. As the root view controller, **BNRItemsViewController** is always in the navigation controller's stack and thus can always access it.

Therefore, **BNRItemsViewController** will be responsible for creating the instance of **BNRDetailViewController** and adding it to the stack. At the top of **BNRItemsViewController.m**, import the header file for **BNRDetailViewController**.

```
#import "BNRDetailViewController.h"

@interface BNRItemsViewController : UITableViewController
```

When a row is tapped in a table view, its delegate is sent **tableView:didSelectRowAtIndexPath:**, which contains the index path of the selected row. In **BNRItemsViewController.m**, implement this method to create a **BNRDetailViewController** and then push it on top of the navigation controller's stack.

```

@implementation BNRItemsViewController

- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    BNRDetailViewController *detailViewController =
        [[BNRDetailViewController alloc] init];

    // Push it onto the top of the navigation controller's stack
    [self.navigationController pushViewController:detailViewController
                                    animated:YES];
}

```

Build and run the application. Create a new item and select that row from the **UITableView**. Not only are you taken to **BNRDetailViewController**'s view, but you also get a free animation and a back button in the **UINavigationBar**. Tap this button to get back to **BNRItemsViewController**.

Since the **UINavigationController**'s stack is an array, it will take ownership of any view controller added to it. Thus, the **BNRDetailViewController** is owned only by the **UINavigationController** after **tableView:didSelectRowAtIndexPath:** finishes. When the stack is popped, the **BNRDetailViewController** is destroyed. The next time a row is tapped, a new instance of **BNRDetailViewController** 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.

(The iPad-only class **UISplitViewController** calls for a different pattern. The iPad's larger screen size allows two view controllers in a drill-down interface to appear on screen simultaneously instead of being pushed onto the same stack. You will learn more about **UISplitViewController** in Chapter 22.)

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 **BNRItem** from the **BNRItemsViewController** to the **BNRDetailViewController**.

To pull this off, you will give **BNRDetailViewController** a property to hold a **BNRItem**. When a row is tapped, **BNRItemsViewController** will give the corresponding **BNRItem** to the instance of **BNRDetailViewController** that is being pushed onto the stack. The **BNRDetailViewController** will populate its text fields with the properties of that **BNRItem**. Editing the text in the text fields on **BNRDetailViewController**'s view will change the properties of that **BNRItem**.

In **BNRDetailViewController.h**, add this property. Also, at the top of this file, forward declare **BNRItem**.

```

#import <UIKit/UIKit.h>

@class BNRItem;

@interface BNRDetailViewController : UIViewController

@property (nonatomic, strong) BNRItem *item;

@end

```

In `BNRDetailViewController.m`, import `BNRItem`'s header file.

```
#import "BNRItem.h"
```

When the `BNRDetailViewController`'s view appears on the screen, it needs to set up its subviews to show the properties of the item. In `BNRDetailViewController.m`, override `viewWillAppear:` to transfer the item's properties to the various instances of `UITextField`.

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    BNRItem *item = self.item;

    self.nameField.text = item.itemName;
    self.serialNumberField.text = item.serialNumber;
    self.valueField.text = [NSString stringWithFormat:@"%@", item.valueInDollars];

    // You need an NSDateFormatter that will turn a date into a simple date string
    static NSDateFormatter *dateFormatter = nil;
    if (!dateFormatter) {
        dateFormatter = [[NSDateFormatter alloc] init];
        dateFormatter.dateStyle = NSDateFormatterMediumStyle;
        dateFormatter.timeStyle = NSDateFormatterNoStyle;
    }

    // Use filtered NSDate object to set dateLabel contents
    self.dateLabel.text = [dateFormatter stringFromDate:item.dateCreated];
}
```

In `BNRItemsViewController.m`, add the following code to `tableView:didSelectRowAtIndexPath:` so that `BNRDetailViewController` has its item before `viewWillAppear:` gets called.

```
- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    BNRDetailViewController *detailViewController =
        [[BNRDetailViewController alloc] init];

    NSArray *items = [[BNRItemStore sharedStore] allItems];
    BNRItem *selectedItem = items[indexPath.row];

    // Give detail view controller a pointer to the item object in row
    detailViewController.item = selectedItem;

    [self.navigationController pushViewController:detailViewController
        animated:YES];
}
```

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 `BNRItem`. 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 `BNRItem` 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 sends out two messages: **viewWillDisappear:** and **viewWillAppear:**. The **UIViewController** that is about to be popped off the stack is sent the message **viewWillDisappear:**. The **UIViewController** that will then be on top of the stack is sent **viewWillAppear:**.

When a **BNRDetailViewController** 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 **BNRDetailViewController.m**, implement **viewWillDisappear:**:

```
- (void)viewWillDisappear:(BOOL)animated
{
    [super viewWillDisappear:animated];

    // Clear first responder
    [self.view endEditing:YES];

    // "Save" changes to item
    BNRItem *item = self.item;
    item.itemName = self.nameField.text;
    item.serialNumber = self.serialNumberField.text;
    item.valueInDollars = [self.valueField.text intValue];
}
```

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 YES ignores that refusal.)

Now the values of the **BNRItem** will be updated when the user taps the Back button on the **UINavigationBar**. When **BNRItemsViewController** 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 **BNRItemsViewController.m**, override **viewWillAppear:**:

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];

    [self.tableView reloadData];
}
```

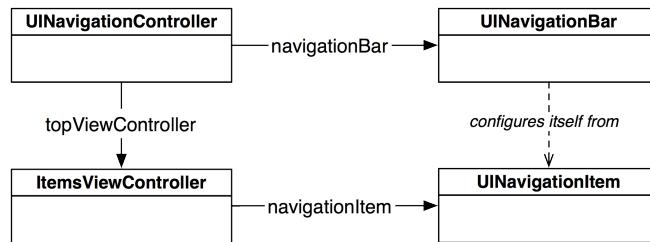
Build and run your application now. Now you can move back and forth between the view controllers that you created and change the data with ease.

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.

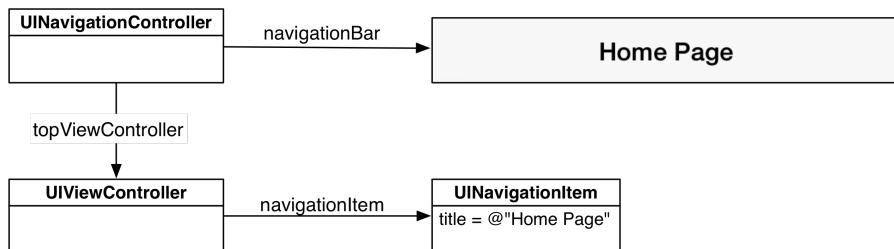
Every **UIViewController** has a **navigationItem** property of type **UINavigationItem**. However, unlike **UINavigationBar**, **UINavigationItem** is not a subclass of **UIView**, so it cannot appear on the screen. Instead, the navigation item supplies the navigation bar with the content it needs to 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 10.14.

Figure 10.14 **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 10.15).

Figure 10.15 **UINavigationItem** with title



In **BNRItemsViewController.m**, modify **init** to set the **navigationItem**'s title to read **Homepwner**.

```

- (instancetype)init
{
    self = [super initWithStyle:UITableViewStylePlain];
    if (self) {
        UINavigationItem *navItem = self.navigationItem;
        navItem.title = @"Homepwner";
    }
    return self;
}
  
```

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 **BNRDetailViewController** a title, too. It would be nice to have the **BNRDetailViewController**'s navigation item title be the name of the **BNRItem** it is displaying. Obviously, you cannot do this in **init** because you do not yet know what its **item** will be.

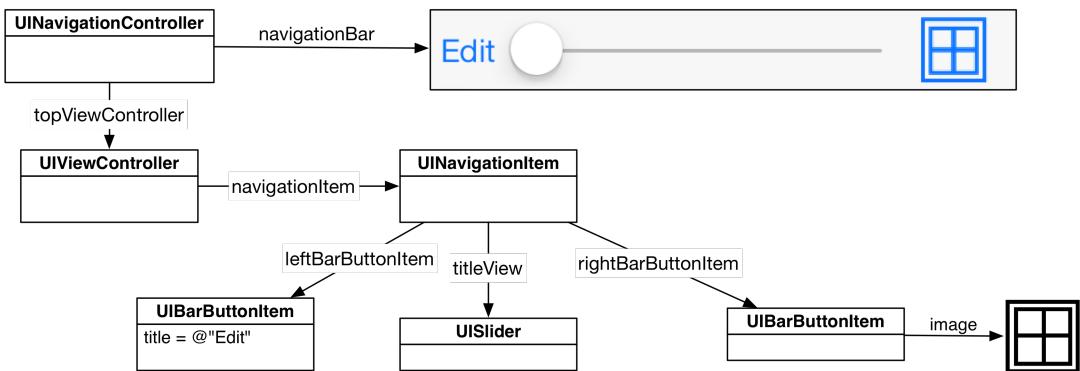
Instead, the **BNRDetailViewController** will set its title when it sets its `item` property. In `BNRDetailViewController.m`, implement `setItem:`, replacing the synthesized setter method for `item`.

```
- (void)setItem:(BNRItem *)item
{
    _item = item;
    self.navigationItem.title = _item.itemName;
}
```

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 **BNRItem** you selected.

A navigation item can hold more than just a title string, as shown in Figure 10.16. 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 10.16 **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 10.16 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 **BNRItemsViewController** is on top of the stack. When tapped, it should add a new **BNRItem** 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 `BNRItemsViewController.m`, create a `UIBarButtonItem` instance and give it its target and action.

```
- (instancetype)init
{
    self = [super initWithStyle:UITableViewStylePlain];
    if (self) {
        UINavigationItem *navItem = self.navigationItem;
        navItem.title = @"Homepwner";

        // Create a new bar button item that will send
        // addNewItem: to BNRItemsViewController
        UIBarButtonItem *bbi = [[UIBarButtonItem alloc]
                               initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
                               target:self
                               action:@selector(addNewItem:));

        // Set this bar button item as the right item in the navigationItem
        navItem.rightBarButtonItem = bbi;
    }
    return self;
}
```

The action is passed as a value of type SEL. Recall that the SEL data type is a pointer to a selector and that a selector is the entire message name including any colons. Note that `@selector()` does not care about the return type, argument types, or names of arguments.

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 `BNRItemsViewController.m`, edit the `init` method.

```
- (instancetype)init
{
    self = [super initWithStyle:UITableViewStylePlain];
    if (self) {
        UINavigationItem *navItem = self.navigationItem;
        navItem.title = @"Homepwner";

        // Create a new bar button item that will send
        // addNewItem: to BNRItemsViewController
        UIBarButtonItem *bbi = [[UIBarButtonItem alloc]
                               initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
                               target:self
                               action:@selector(addNewItem));

        // Set this bar button item as the right item in the navigationItem
        navItem.rightBarButtonItem = bbi;

        navItem.leftBarButtonItem = self.editButtonItem;
    }
    return self;
}
```

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 sends the message **setEditing:animated:** to its **UIViewController** when tapped.

Now that Homeowner has a fully functional navigation bar, you can get rid of the header view and the associated code. In **BNRItemsViewController.m**, delete the following methods.

```
- (UIView *)headerView
{
    if (!_headerView) {
        [[NSBundle mainBundle] loadNibNamed:@"HeaderView" owner:self options:nil];
    }
    return _headerView;
}

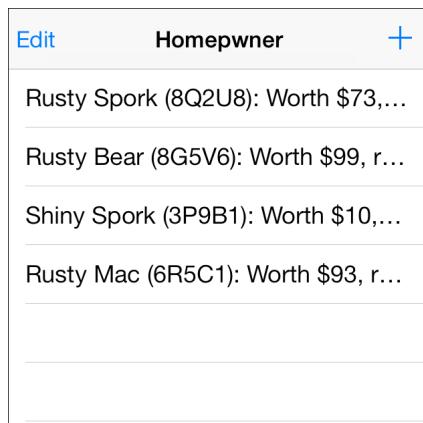
- (IBAction)toggleEditingStyle:(id)sender
{
    if (self.isEditing) {
        [sender setTitle:@"Edit" forState:UIControlStateNormal];
        [self setEditing:NO animated:YES];
    } else {
        [sender setTitle:@"Done" forState:UIControlStateNormal];
        [self setEditing:YES animated:YES];
    }
}
```

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 10.17).

Figure 10.17 Homeowner with navigation bar



Bronze Challenge: Displaying a Number Pad

The keyboard for the **UITextField** that displays a **BNRItem**'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

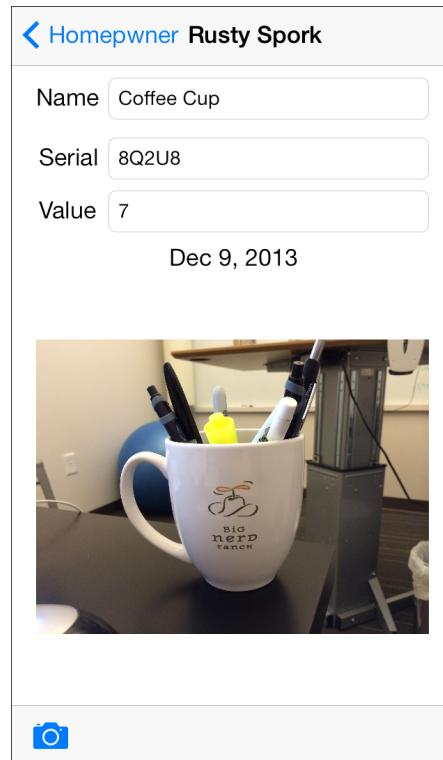
Right now, instances of **BNRItem** cannot have their `dateCreated` property changed. Change **BNRItem** so that they can, and then add a button underneath the `dateLabel` in **BNRDetailViewController** with the title `Change Date`. When this button is tapped, push another view controller instance onto the navigation stack. This view controller should have a **UIDatePicker** instance that modifies the `dateCreated` property of the selected **BNRItem**.

11

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 a **BNRItem** instance and viewable in the item's detail view.

Figure 11.1 Homepwner with camera addition



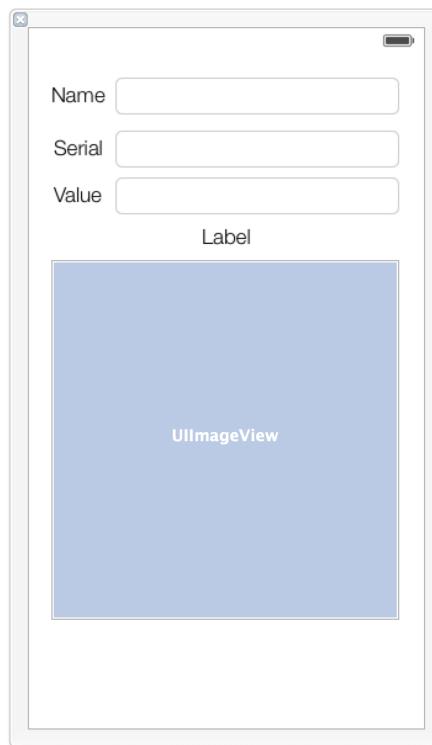
Images tend to be very large, so it is a good idea to store images separately from other data. Thus, in this chapter, you are going to create a second store for images. **BNRImageStore** will fetch and cache images as they are needed. It will also be able to flush the cache when memory runs low.

Displaying Images and UIImageView

Your first step is to have the **BNRDetailViewController** get and display an image. An easy way to display an image is to put an instance of **UIImageView** on the screen.

Open **Homewner.xcodeproj** and **BNRDetailViewController.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 11.2).

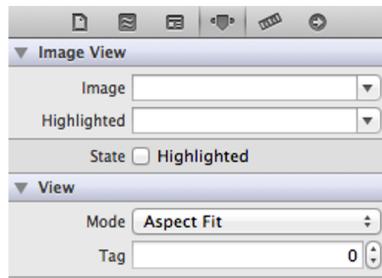
Figure 11.2 **UIImageView** on **BNRDetailViewController**'s view



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 `UIViewContentModeScaleToFill`, 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 11.3). This will resize the image to fit within the bounds of the **UIImageView**.

Figure 11.3 Change **UIImageView**'s mode to Aspect Fit



Next, Option-click `BNRDetailViewController.m` in the project navigator to open it in the assistant editor. Control-drag from the **UIImageView** to the class extension in `BNRDetailViewController.m`. Name the outlet `imageView` and choose **Weak** as the storage type. Click Connect.

`BNRDetailViewController`'s class extension should now look like this:

```
@interface BNRDetailViewController : UIViewController

@property (weak, nonatomic) IBOutlet UITextField *nameField;
@property (weak, nonatomic) IBOutlet UITextField *serialNumberField;
@property (weak, nonatomic) IBOutlet UITextField *valueField;
@property (weak, nonatomic) IBOutlet UILabel *dateLabel;
@property (weak, nonatomic) IBOutlet UIImageView *imageView;

@end
```

Adding a camera button

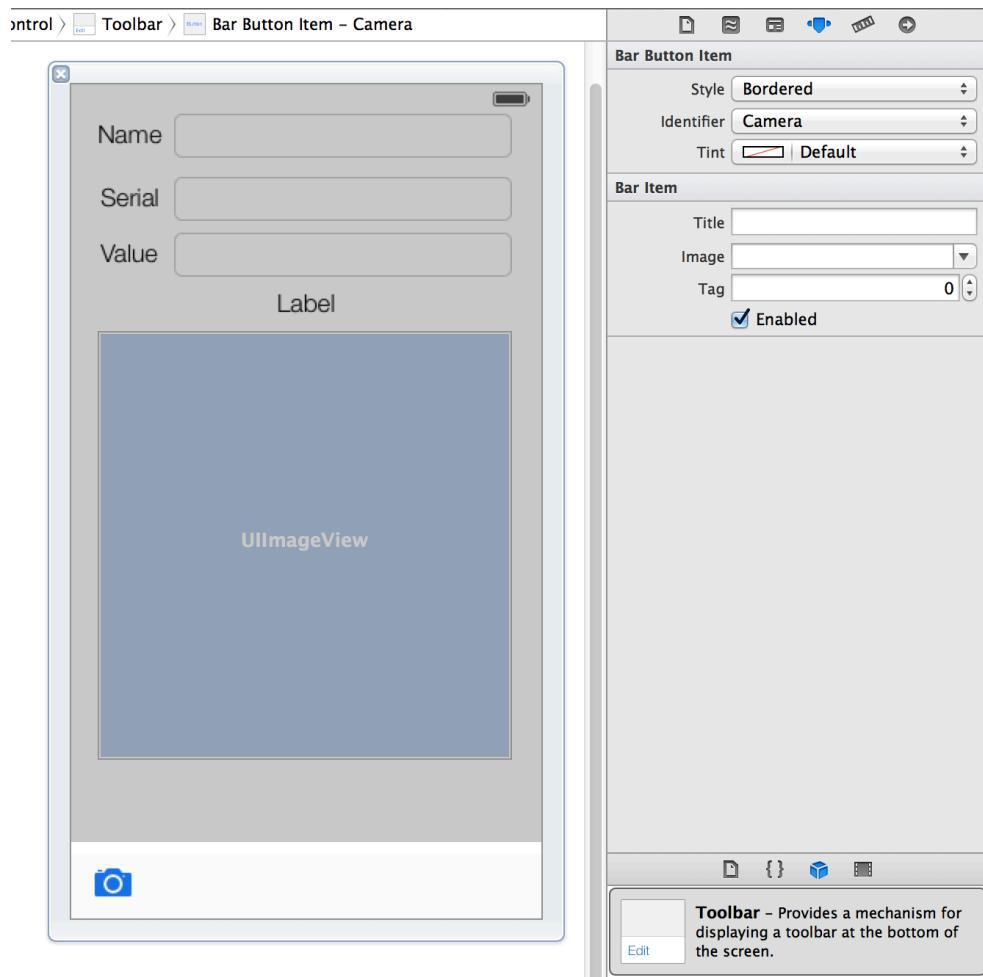
Now you need a button to initiate the photo-taking process. It would be nice to put this button on the navigation bar, but you will need the navigation bar for another button later. Instead, you will create an instance of **UIToolbar** and place it at the bottom of `BNRDetailViewController`'s view.

In `BNRDetailViewController.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 11.4).

Figure 11.4 UIToolbar with bar button item

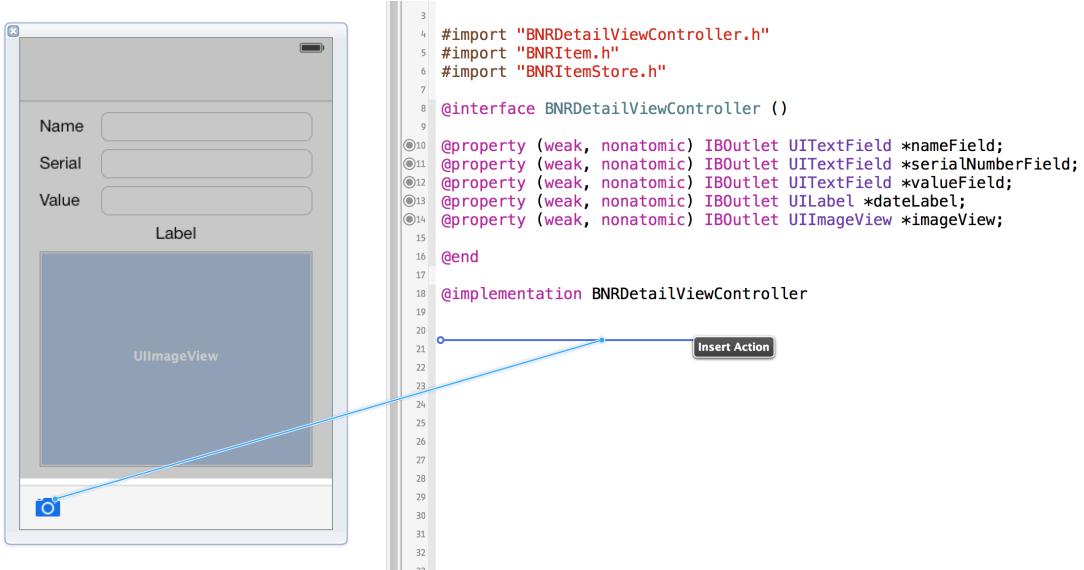


The camera button needs a target and an action. In previous exercises, you connected an action method in two steps: declaring it in code and then making the connection in the XIB file. Just like with outlets, there is a way to do both steps at once.

In the project navigator, Option-click `BNRDetailViewController.m` to open it in the assistant editor.

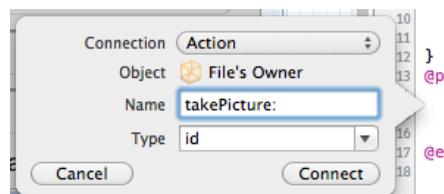
In `BNRDetailViewController.xib`, select the camera button by first clicking on the toolbar and then the button itself. Then Control-drag from the selected button to the implementation part of `BNRDetailViewController.m` (Figure 11.5).

Figure 11.5 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 11.6).

Figure 11.6 Creating the action

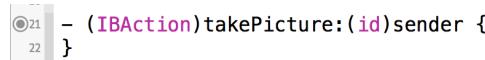


Now the stub of the action method is in `BNRDetailViewController.m`, and the `UIBarButtonItem` instance in the XIB is hooked up to send this message to the `BNRDetailViewController` when tapped. The stub should look like this:

```
- (IBAction)takePicture:(id)sender
{}
```

Xcode is smart enough to know when an action method is connected in the XIB file. In Figure 11.7, 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 11.7 Source file connection status



```
21 - (IBAction)takePicture:(id)sender {  
22 }
```

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 the class extension in **BNRDetailViewController.m**. Name this outlet **toolbar** and ensure that its storage is Weak.

The interface for **BNRDetailViewController** now has a **toolbar** outlet:

```
@interface BNRDetailViewController ()  
  
@property (weak, nonatomic) IBOutlet UITextField *nameField;  
@property (weak, nonatomic) IBOutlet UITextField *serialNumberField;  
@property (weak, nonatomic) IBOutlet UITextField *valueField;  
@property (weak, nonatomic) IBOutlet UILabel *dateLabel;  
@property (weak, nonatomic) IBOutlet UIImageView *imageView;  
@property (weak, nonatomic) IBOutlet UIToolbar *toolbar;  
  
@end
```

If you made any mistakes while making these connections, you will need to open **BNRDetailViewController.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:

UIImagePickerControllerSourceTypeCamera

The user will take a new picture.

UIImagePickerControllerSourceTypePhotoLibrary

The user will be prompted to select an album and then a photo from that album.

UIImagePickerControllerSourceTypeSavedPhotosAlbum

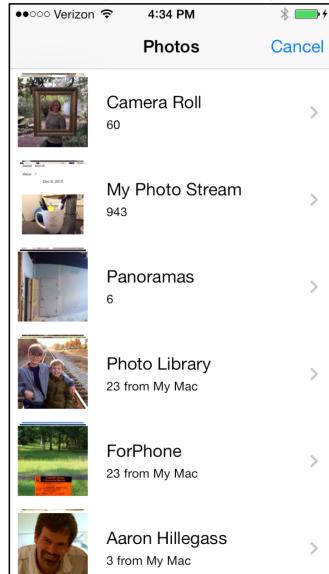
The user picks from the most recently taken photos.

Figure 11.8 Examples of three source types

SourceTypeCamera



SourceTypePhotoLibrary



SourceTypeSavedPhotosAlbum



The first source type, `UIImagePickerControllerSourceTypeCamera`, 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 sending the message `isSourceTypeAvailable:` to the `UIImagePickerController` class:

```
+ (BOOL)isSourceTypeAvailable:(UIImagePickerControllerSourceType)sourceType;
```

Sending this message returns a Boolean value for whether the device supports the passed-in source type.

In `BNRDetailViewController.m`, find the stub for `takePicture:`. Add the following code to create the image picker and set its `sourceType`.

```
- (IBAction)takePicture:(id)sender
{
    UIImagePickerController *imagePicker =
        [[UIImagePickerController alloc] init];

    // If the device has a camera, take a picture, otherwise,
    // just pick from photo library
    if ([UIImagePickerController
        isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera]) {
        imagePicker.sourceType = UIImagePickerControllerSourceTypeCamera;
    } else {
        imagePicker.sourceType = UIImagePickerControllerSourceTypePhotoLibrary;
    }
}
```

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 **BNRDetailViewController**. In **BNRDetailViewController.m**, declare that **BNRDetailViewController** conforms to the **UINavigationControllerDelegate** and the **UIImagePickerControllerDelegate** protocols.

```
@interface BNRDetailViewController ()  
    <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 **BNRDetailViewController.m**, add the following code to **takePicture:** to set the instance of **BNRDetailViewController** to be the image picker's delegate.

```
- (IBAction)takePicture:(id)sender  
{  
    UIImagePickerController *imagePicker =  
        [[UIImagePickerController alloc] init];  
  
    // If the device has a camera, take a picture, otherwise,  
    // just pick from photo library  
    if ([UIImagePickerController  
        isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera]) {  
        imagePicker.sourceType = UIImagePickerControllerSourceTypeCamera;  
    } else {  
        imagePicker.sourceType = UIImagePickerControllerSourceTypePhotoLibrary;  
    }  
  
    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 17.)

In **BNRDetailViewController.m**, add code to the end of **takePicture:** to present the **UIImagePickerController**.

```
imagePickerController.delegate = self;  
  
// Place image picker on the screen  
[self presentViewController:imagePickerController animated:YES completion:nil];  
  
}
```

(The third argument, `completion:`, expects a block. You will learn about completion blocks in Chapter 17.)

You can build and run the application now. Select a `BNRItem` to see its details and then tap the camera button on the `UIToolbar`. `UIImagePickerController`'s interface will appear on the screen (Figure 11.9), 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. Then this image will be shown in the image picker. The simulator can be flaky, so you might have to try a few different images before one saves to the library.)

Figure 11.9 `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 message is sent to the image picker's delegate when a photo has been selected.

In `BNRDetailViewController.m`, implement this method to put the image into the `UIImageView` and then send a message to dismiss the image picker.

```
- (void)imagePickerController:(UIImagePickerController *)picker
didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    // Get picked image from info dictionary
    UIImage *image = info[UIImagePickerControllerOriginalImage];

    // Put that image onto the screen in our image view
    self.imageView.image = image;

    // Take image picker off the screen -
    // you must call this dismiss method
    [self dismissViewControllerAnimated:YES completion:nil];
}
```

Build and run the application again. Take (or select) a photo. The image picker is dismissed, and you are returned to the `BNRDetailViewController`'s view.

You could have hundreds of items, and each one could have a large image associated with it. Keeping hundreds of instances of `BNRItem` 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, `BNRImageStore`. When the `BNRImageStore` receives a low-memory notification, it will flush its cache to free the memory that the fetched images were occupying.

Creating `BNRImageStore`

The image store will hold the pictures the user takes. In Chapter 18, you will have instances of `BNRItem` 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 `BNRImageStore`. Open `BNRImageStore.h` and create its interface:

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

@interface BNRImageStore : NSObject
+ (instancetype)sharedStore;
- (void)setImage:(UIImage *)image forKey:(NSString *)key;
- (UIImage *)imageForKey:(NSString *)key;
- (void)deleteImageForKey:(NSString *)key;
@end
```

In `BNRIImageStore.m`, add a class extension to declare a property to hang onto the images.

```
@interface BNRIImageStore ()  
@property (nonatomic, strong) NSMutableDictionary *dictionary;  
@end  
@implementation BNRIImageStore
```

Like the `BNRItemStore`, the `BNRIImageStore` needs to be a singleton. In `BNRIImageStore.m`, write the following code to ensure `BNRIImageStore`'s singleton status.

```
@implementation BNRIImageStore  
+ (instancetype)sharedStore  
{  
    static BNRIImageStore *sharedStore = nil;  
  
    if (!sharedStore) {  
        sharedStore = [[self alloc] initPrivate];  
    }  
    return sharedStore;  
}  
  
// No one should call init  
- (instancetype)init  
{  
    @throw [NSError exceptionWithName:@"Singleton"  
                      reason:@"Use +[BNRIImageStore sharedStore]"  
                     userInfo:nil];  
    return nil;  
}  
  
// Secret designated initializer  
- (instancetype)initPrivate  
{  
    self = [super init];  
  
    if (self) {  
        _dictionary = [[NSMutableDictionary alloc] init];  
    }  
  
    return self;  
}
```

Then, implement the other three methods declared in the header file.

```
- (void)setImage:(UIImage *)image forKey:(NSString *)key
{
    [self.dictionary setObject:image forKey:key];
}

- (UIImage *)imageForKey:(NSString *)key
{
    return [self.dictionary objectForKey:key];
}

- (void)deleteImageForKey:(NSString *)key
{
    if (!key) {
        return;
    }
    [self.dictionary removeObjectForKey:key];
}
```

NSDictionary

Notice that the dictionary is an instance of **NSMutableDictionary**. Like an array, a *dictionary* is a collection object that has an immutable version (**NSDictionary**) and a mutable version (**NSMutableDictionary**).

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 some object at the beginning of an array
[someArray insertObject:someObject atIndex:0];

// Get that same object out
someObject = [someArray objectAtIndex:0];
```

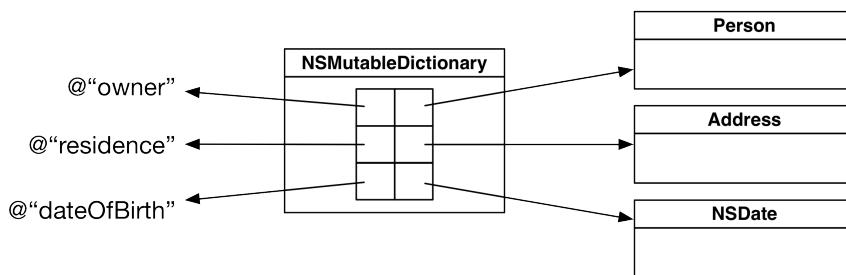
A dictionary's objects are not ordered within the collection. So instead of accessing entries with an index, you use a *key*. The key is usually an instance of **NSString**.

```
// Add some object to a dictionary for the key "MyKey"
[someDictionary setObject:someObject forKey:@"MyKey"];

// Get that same object out
someObject = [someDictionary objectForKey:@"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 11.10 **NSDictionary** diagram



There are a lot of uses for an **NSDictionary**. 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 instance variables like `firstName`, `age`, and other things that you expect a real-life person to have. An instance of **NSDictionary** 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 an **NSDictionary**, 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 **NSDictionary** 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 **BNRItem**. However, **NSDictionary** 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 **NSDictionary** 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 **NSDictionary** is creating lookup tables. Sometime early in your programming career, you probably did something like this:

```

- (void)changeCharacterClass:(id)sender
{
    NSString *enteredText = textField.text;
    CharacterClass *cc = nil;

    if ([enteredText isEqualToString:@"Warrior"]) {
        cc = knight;
    } else if ([enteredText isEqualToString:@"Mage"]) {
        cc = wizard;
    } else if ([enteredText isEqualToString:@"Thief"]) {
        cc = rogue;
    }

    character.characterClass = cc;
}

```

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 **NSDictionary** could be initialized like so:

```
NSMutableDictionary *lookup = [[NSMutableDictionary alloc] init];
[lookup setObject:knight forKey:@"Warrior"];
[lookup setObject:wizard forKey:@"Mage"];
[lookup setObject:rogue forKey:@"Thief"];
```

and then you can change the **changeCharacterClass:** method to something much cleaner:

```
- (void)changeCharacterClass:(id)sender
{
    character.characterClass = [lookup objectForKey: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 **BNRImageStore** 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 object for each key. If you add an object to a dictionary with a key that matches the key of an object already present in the dictionary, the earlier object is removed. If you need to store multiple objects under one key, you can put them in an array and add the array to the dictionary as the value.

Dictionaries, like arrays, can be created using shorthand syntax. The shorthand syntax for dictionary creation uses curly braces (@{}), unlike the square brackets that **NSArray** uses (@[]). When initializing a dictionary using shorthand syntax, each key-value pair is separated by a comma (,), A colon (:) is placed between the key and its value.

```
NSDictionary *dictionary = @{@"key": object, @"anotherKey": anotherObject};
```

Dictionaries also have a shorthand syntax for retrieving objects:

```
id object = dictionary[@"key"];
// same as
id object = [dictionary objectForKey:@"key"];
```

If you have an **NSMutableDictionary**, you can set the object for a key with shorthand syntax:

```
dictionary[@"key"] = object;
// same as
[dictionary setObject:object forKey:@"key"];
```

Update the image store to use the shorthand form of accessing and modifying dictionaries.

```
- (void)setImage:(UIImage *)image forKey:(NSString *)key
{
    [self.dictionary setObject:image forKey:key];
    self.dictionary[key] = image;
}

- (UIImage *)imageForKey:(NSString *)key
{
    return [self.dictionary objectForKey:key];
    return self.dictionary[key];
}
```

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.

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 **BNRItem** object will be given that key. When the **BNRDetailViewController** 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 `BNRItem.h` to store the key.

```
@property (nonatomic, readonly, strong) NSDate *dateCreated;
@property (nonatomic, copy) NSString *itemKey;
```

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

Import `BNRImageStore.h` at the top of `BNRDetailViewController.m`.

```
#import "BNRDetailViewController.h"
#import "BNRItem.h"
#import "BNRImageStore.h"
```

In `BNRItem.m`, modify the designated initializer to generate a UUID and set it as the `itemKey`.

```
- (instancetype)initWithItemName:(NSString *)name
                           valueInDollars:(int)value
                            serialNumber:(NSString *)sNumber
{
    // Call the superclass's designated initializer
    self = [super init];
    // Did the superclass's designated initializer succeed?
    if (self) {
        // Give the instance variables initial values
        _itemName = name;
        _serialNumber = sNumber;
        _valueInDollars = value;
        // set _dateCreated to the current date and time
        _dateCreated = [[NSDate alloc] init];

        // Create an NSUUID object - and get its string representation
        NSUUID *uuid = [[NSUUID alloc] init];
        NSString *key = [uuid UUIDString];
        _itemKey = key;
    }
    // Return the address of the newly initialized object
    return self;
}
```

Then, in `BNRDetailViewController.m`, update `imagePickerController:didFinishPickingMediaWithInfo:` to store the image in the `BNRImageStore`.

```
- (void)imagePickerController:(UIImagePickerController *)picker
didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    UIImage *image = info[UIImagePickerControllerOriginalImage];

    // Store the image in the BNRImageStore for this key
    [[BNRImageStore sharedStore] setImage:image
                                    forKey:self.item.itemKey];

    imageView.image = image;
    [self dismissViewControllerAnimated:YES completion:nil];
}
```

Each time an image is captured, it will be added to the store. Both the `BNRImageStore` and the `BNRItem` 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. At the top of `BNRItemStore.m`, import the header for the `BNRImageStore` and add the following code to `removeItem`:

```
#import "BNRImageStore.h"

@implementation BNRItemStore

- (void)removeItem:(BNRItem *)item
{
    NSString *key = item.itemKey;

    [[BNRImageStore sharedStore] deleteImageForKey:key];
    [self.privateItems removeObjectIdenticalTo:item];
}
```

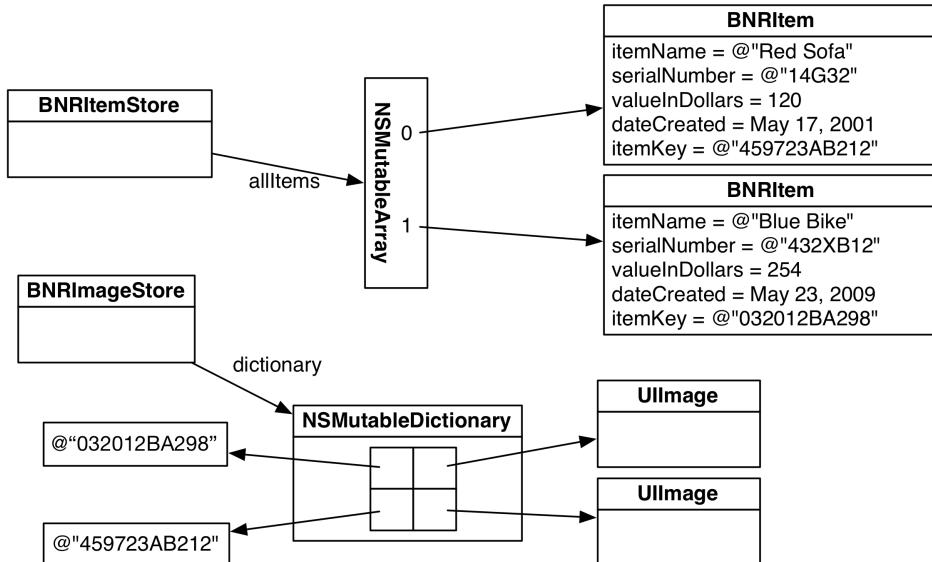
You might be thinking, “Why not give the `BNRItem` a pointer to the image? After all, isn’t a pointer to the image a more direct way of referring to the image?” While this is correct, you must consider what happens when you begin saving the items and their images to the filesystem in Chapter 18.

When a `UIImage` is first created, it exists in memory at a specific address. A pointer holds onto this address so you can refer to the image again. The next time the application launches, however, the image will not be at the same address in memory, so you cannot use the same pointer to access it. Instead, the key will be used to name the image file on the filesystem and each `BNRItem` will hang on to its key. When you want to load the image back into memory, the `BNRImageStore` will use the `itemKey` of a `BNRItem` to find the image file on the filesystem, load it into memory, and return a pointer to the new `UIImage` instance. Therefore, the key is a persistent way of referring to an image.

Wrapping up BNRImageStore

Now that the **BNRImageStore** can store images and instances of **BNRItem** have a key to get that image (Figure 11.11), you need to teach **BNRDetailViewController** how to grab the image for the selected **BNRItem** and place it in its `imageView`.

Figure 11.11 Cache



The **BNRDetailViewController**'s view will appear at two times: when the user taps a row in **BNRItemsViewController** and when the **UIImagePickerController** is dismissed. In both of these situations, the `imageView` should be populated with the image of the **BNRItem** being displayed.

In `BNRDetailViewController.m`, add code to `viewWillAppear:` to do this.

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    self.nameField.text = item.itemName;
    self.serialNumberField.text = item.serialNumber;
    self.valueField.text = [NSString stringWithFormat:@"%@", item.valueInDollars];

    static NSDateFormatter *dateFormatter = nil;
    if (!dateFormatter) {
        dateFormatter = [[NSDateFormatter alloc] init];
        dateFormatter.dateStyle = NSDateFormatterMediumStyle;
        dateFormatter.timeStyle = NSDateFormatterNoStyle;
    }

    self.dateLabel.text = [dateFormatter stringFromDate:item.dateCreated];

    NSString *imageKey = self.item.imageKey;

    // Get the image for its image key from the image store
    UIImage *imageToDisplay = [[BNRImageStore sharedStore] imageForKey:imageKey];

    // Use that image to put on the screen in the imageView
    self.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 `BNRItem` 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 `BNRDetailViewController`'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 `BNRDetailViewController.m`, have `BNRDetailViewController` conform to the `UITextFieldDelegate` protocol.

```
@interface BNRDetailViewController : UIViewController
<UINavigationControllerDelegate, UIImagePickerControllerDelegate,
UITextFieldDelegate>
```

In `BNRDetailViewController.m`, implement `textFieldShouldReturn:`.

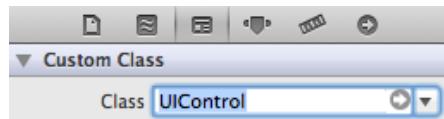
```
- (BOOL)textFieldShouldReturn:(UITextField *)textField
{
    [textField resignFirstResponder];
    return YES;
}
```

It would be stylish to also dismiss the keyboard if the user taps anywhere else on `BNRDetailViewController`'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 `BNRDetailViewController` from an instance of `UIView` to an instance of `UIControl` so that it can handle touch events.

In `BNRDetailViewController.xib`, select the main View object. Open the identity inspector and change the view's class to `UIControl` (Figure 11.12).

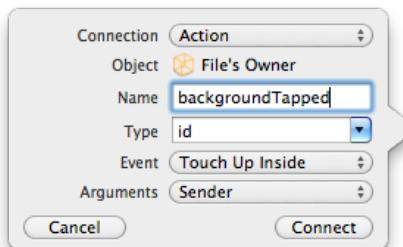
Figure 11.12 Changing the class of `BNRDetailViewController`'s view



Then, open `BNRDetailViewController.m` in the assistant editor. Control-drag from the view (now a `UIControl`) to the implementation of `BNRDetailViewController`. 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 as it does in Figure 11.13 and click Connect.

Figure 11.13 Configuring a `UIControl` action



This will create a stub method in `BNRDetailViewController.m`. Update that method:

```
- (IBAction)backgroundTapped:(id)sender
{
    [self.view endEditing:YES];
}
```

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 `BNRDetailViewController`.

Silver Challenge: Removing an Image

Add a button that clears the image for an item.

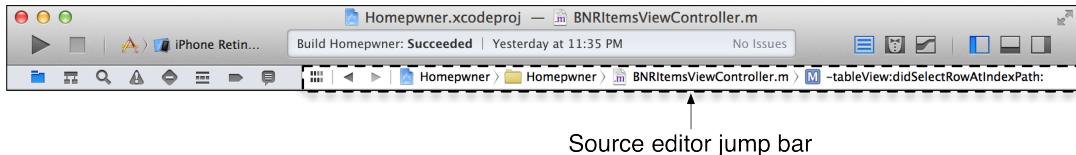
Gold Challenge: Camera Overlay

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

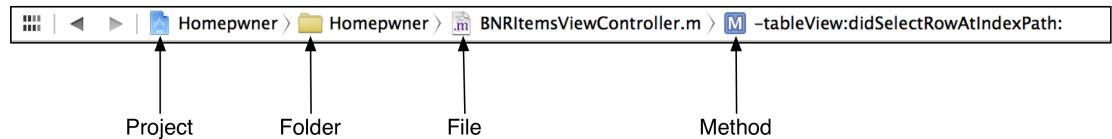
Both of your view controllers have quite a few methods in their implementation files. To be effective iOS developers, you must be able to go to the code you are looking for quickly and easily. The source editor jump bar in Xcode is one tool at your disposal to help out with this (Figure 11.14).

Figure 11.14 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 11.15 breaks down the jump bar details.

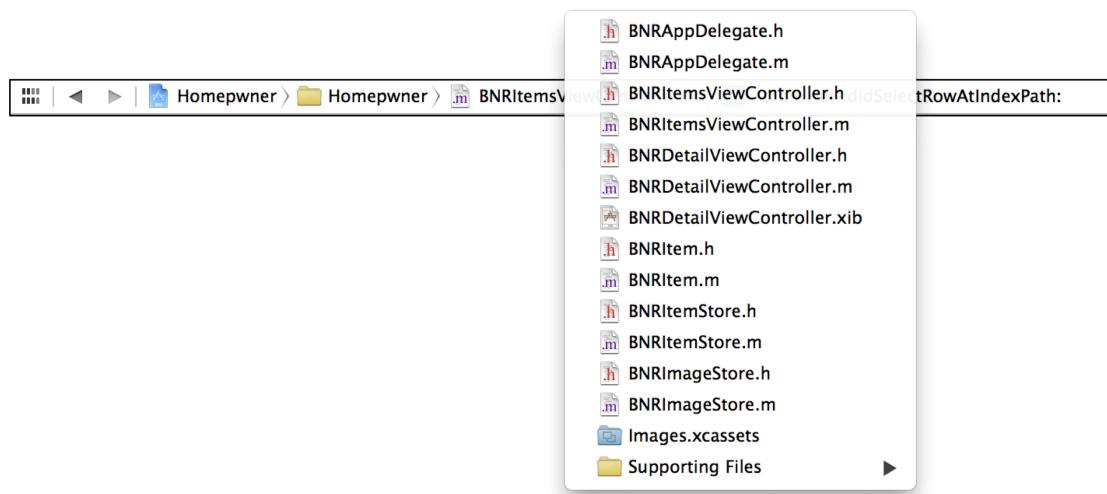
Figure 11.15 Jump bar details



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 11.16 shows off the file popover in the Homepwner application.

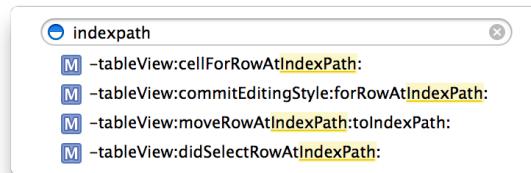
Figure 11.16 File popover



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 11.17 shows what you get when you search for “indexPath” in `BNRItemsViewController.m`.

Figure 11.17 File popover with “indexPath” search



#pragma mark

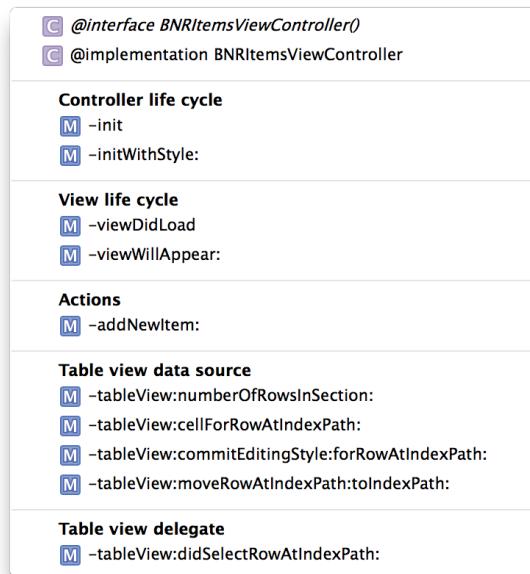
As your classes get longer, it can get more difficult to find the method you are looking for when it is buried in a long list of methods. A good way to organize your methods to help with the mess is by using the `#pragma mark` preprocessor directive.

```
#pragma mark - View life cycle
- (void)viewDidLoad {...}
- (void)viewWillAppear:(BOOL)animated {...}
```

```
#pragma mark - Actions
- (void)addNewItem:(id)sender {...}
```

Adding `#pragma` marks 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 11.18 shows the results of a well-organized `BNRItemsViewController.m`.

Figure 11.18 File popover with `#pragma` marks



Two useful `#pragma` marks are the divider and the label.

```
// This is a divider  
#pragma mark -  
  
// This is a label  
#pragma mark My Awesome Methods  
  
// They can be combined as well  
#pragma mark - My Awesome Methods
```

By using the `#pragma mark` directive, 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.

For the More Curious: Recording Video

Once you understand how to use `UIImagePickerController` to take pictures, making the transition to recording video is trivial. Recall that an image picker controller has a `sourceType` property that determines whether an image comes from the camera, photo library, or saved photos album. Image picker controllers also have a `mediaTypes` property, which is an array of strings that contains identifiers for what types of media can be selected from the three source types.

There are two types of media a **UIImagePickerController** can select: still images and video. By default, the `mediaTypes` array only contains the constant string `kUTTypeImage`. Thus, if you do not change the `mediaTypes` property of an image picker controller, the camera will only allow the user to take still photos, and the photo library and saved photos album will only display images.

Adding the ability to record video or choose a video from the disk is as simple as adding the constant string `kUTTypeMovie` to the `mediaTypes` array. However, not all devices support video through the **UIImagePickerController**. Just like the class method `isSourceTypeAvailable:` allows you to determine if the device has a camera, the `availableMediaTypesForSourceType:` method checks to see if that camera can capture video. To set up an image picker controller that can record video or take still images, you would write the following code:

```
UIImagePickerController *ipc = [[UIImagePickerController alloc] init];
NSArray *availableTypes = [UIImagePickerController
    availableMediaTypesForSourceType:UIImagePickerControllerSourceTypeCamera];
ipc.mediaTypes = availableTypes;
ipc.sourceType = UIImagePickerControllerSourceTypeCamera;
ipc.delegate = self;
```

Now when this image picker controller interface is presented to the user, there will be a switch that allows them to choose between the still image camera or the video recorder. If the user chooses to record a video, you need to handle that in the **UIImagePickerController** delegate method `imagePickerController:didFinishPickingMediaWithInfo:`.

When dealing with still images, the `info` dictionary that is passed as an argument contains the full image as a **UIImage** object. However, there is no “**UIVideo**” class. (Loading an entire video into memory at once would be tough to do with iOS device memory constraints.) Therefore, recorded video is written to disk in a temporary directory. When the user finalizes the video recording, `imagePickerController:didFinishPickingMediaWithInfo:` is sent to the image picker controller’s delegate, and the path of the video on the disk is in the `info` dictionary. You can get the path like so:

```
- (void)imagePickerController:(UIImagePickerController *)picker
didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    NSURL *mediaURL = info[UIImagePickerControllerMediaURL];
}
```

You will learn about the filesystem in Chapter 18, but what you should know now is that the temporary directory is not a safe place to store the video. It needs to be moved to another location.

```
- (void)imagePickerController:(UIImagePickerController *)picker
didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    NSURL *mediaURL = info[UIImagePickerControllerMediaURL];
    if (mediaURL) {

        // Make sure this device supports videos in its photo album
        if (UIVideoAtPathIsCompatibleWithSavedPhotosAlbum([mediaURL path])) {

            // Save the video to the photos album
            UISaveVideoAtPathToSavedPhotosAlbum([mediaURL path], nil, nil, nil);

            // Remove the video from the temporary directory
            [[NSFileManager defaultManager] removeItemAtPath:[mediaURL path]
                                             error:nil];
        }
    }
}
```

That is really all there is to it. There is just one situation that requires some additional information: suppose you want to restrict the user to choosing *only* videos. Restricting the user to images is simple (leave `mediaTypes` as the default). Allowing the user to choose between images and videos is just as simple (pass the return value from `availableMediaTypesForSourceType:`). However, to allow video only, you have to jump through a few hoops. First, you must make sure the device supports video, and then you must set the `mediaTypes` property to an array containing only the identifier for video.

```
NSArray *availableTypes = [UIImagePickerController
    availableMediaTypesForSourceType:UIImagePickerControllerSourceTypeCamera];
if ([availableTypes containsObject:(__bridge NSString *)kUTTypeMovie]) {
    [ipc setMediaTypes:@[(__bridge NSString *)kUTTypeMovie]];
}
```

Wondering why `kUTTypeMovie` is cast to an `NSString`? This constant is declared as:

```
const CFStringRef kUTTypeMovie;
```

If you build this code, it will fail, and the compiler will complain that it has never heard of `kUTTypeMovie`. Oddly enough, both `kUTTypeMovie` and `kUTTypeImage` are declared and defined in another framework – **MobileCoreServices**. You have to explicitly add this framework and import its header file into your project to use these two constants.

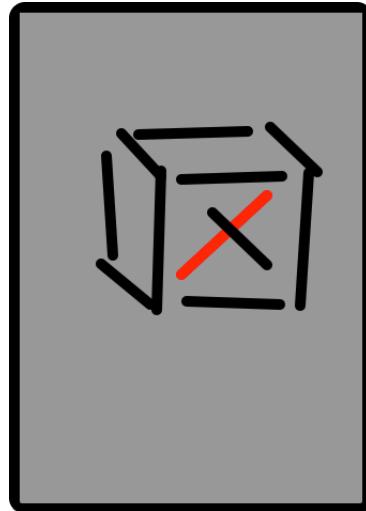
12

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 12.1). Using multi-touch, the user will be able to draw more than one line at a time.

Figure 12.1 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
 - `(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;`
- a finger or fingers moves across the screen (this message is sent repeatedly as a finger moves)
 - `(void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event;`

- a finger or fingers is removed from the screen
 - `(void)touchesEnded:(NSSet *)touches
withEvent:(UIEvent *)event;`
- a system event, like an incoming phone call, interrupts a touch before it ends
 - `(void)touchesCancelled:(NSSet *)touches
withEvent:(UIEvent *)event;`

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.

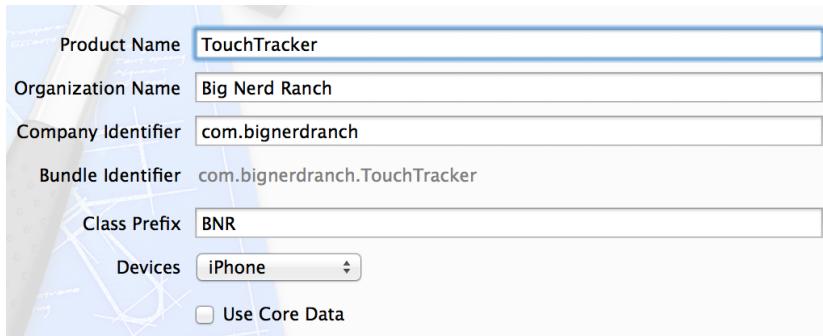
Every time a touch does something, like begins, moves, or ends, a *touch event* is added to a queue of events that the **UIApplication** object manages. In practice, the queue rarely fills up, and events are delivered immediately. The delivery of these touch events involves sending one of the **UIResponder** 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 14 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 Empty Application iPhone project and name it **TouchTracker**. The class prefix should be the same as the other projects, **BNR** (Figure 12.2).

Figure 12.2 Creating TouchTracker



First, you will need a model object that describes a line. Create a new subclass of **NSObject** and name it **BNRLine**. In **BNRLine.h**, declare two **CGPoint** properties:

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

@interface BNRLine : NSObject

@property (nonatomic) CGPoint begin;
@property (nonatomic) CGPoint end;

@end
```

Next, create a new **NSObject** subclass called **BNRDrawView**. In **BNRDrawView.h**, change the superclass to **UIView**.

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

@interface BNRDrawView : NSObject
@interface BNRDrawView : UIView

@end
```

Now you need a view controller to manage an instance of **BNRDrawView** in TouchTracker. Create a new **NSObject** subclass named **BNRDrawViewController**. In **BNRDrawViewController.h**, change the superclass to **UIViewController**.

```
@interface BNRDrawViewController : NSObject
@interface BNRDrawViewController : UIViewController
```

In **BNRDrawViewController.m**, override **loadView** to set up an instance of **BNRDrawView** as **BNRDrawViewController**'s view. Make sure to import the header file for **BNRDrawView** at the top of this file.

```
#import "BNRDrawViewController.h"
#import "BNRDrawView.h"

@implementation BNRDrawViewController

- (void)loadView
{
    self.view = [[BNRDrawView alloc] initWithFrame:CGRectZero];
}

@end
```

In `BNRAppDelegate.m`, create an instance of `BNRDrawViewController` and set it as the `rootViewController` of the window. Do not forget to import the header file for `BNRDrawViewController` in this file.

```
#import "BNRAppDelegate.h"
#import "BNRDrawViewController.h"

@implementation BNRAppDelegate

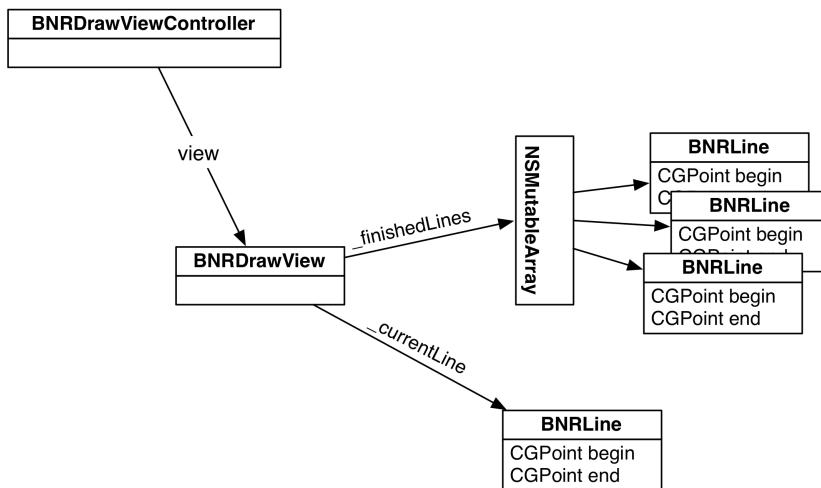
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch

    BNRDrawViewController *dvc = [[BNRDrawViewController alloc] init];
    self.window.rootViewController = dvc;

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}

}
```

Figure 12.3 Object diagram for TouchTracker



The major objects you have just set up for TouchTracker are shown in Figure 12.3.

Drawing with BNRDrawView

`BNRDrawView` will keep track of all of the lines that have been drawn and the line that is currently being drawn. In `BNRDrawView.m`, create two instance variables in the class extension that will hold the lines in their two states. Make sure to import `BNRLine.h` and implement `initWithFrame:`.

```
#import "BNRDrawView.h"
#import "BNRLine.h"

@interface BNRDrawView : NSObject

@property (nonatomic, strong) BNRLine *currentLine;
@property (nonatomic, strong) NSMutableArray *finishedLines;

@end

@implementation BNRDrawView

- (instancetype)initWithFrame:(CGRect)r
{
    self = [super initWithFrame:r];
    if (self) {
        self.finishedLines = [[NSMutableArray alloc] init];
        self.backgroundColor = [UIColor grayColor];
    }
    return self;
}


```

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 **BNRDrawView** to be able to draw lines.

In **BNRDrawView.m**, implement **drawRect:** to draw the current and finished lines.

```
- (void)strokeLine:(BNRLine *)line
{
    UIBezierPath *bp = [UIBezierPath bezierPath];
    bp.lineWidth = 10;
    bp.lineCapStyle = kCGLineCapRound;

    [bp moveToPoint:line.begin];
    [bp addLineToPoint:line.end];
    [bp stroke];
}

- (void)drawRect:(CGRect)rect
{
    // Draw finished lines in black
    [[UIColor blackColor] set];
    for (BNRLine *line in self.finishedLines) {
        [self strokeLine:line];
    }

    if (self.currentLine) {
        // If there is a line currently being drawn, do it in red
        [[UIColor redColor] set];
        [self strokeLine:self.currentLine];
    }
}
```

Turning Touches into Lines

A line is defined by two points. Your **BNRLine** 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 `end`. When the touch ends, you will have your complete line.

In `BNRDrawView.m`, implement `touchesBegan:withEvent:` to create a new line.

```
- (void)touchesBegan:(NSSet *)touches
              withEvent:(UIEvent *)event
{
    UITouch *t = [touches anyObject];

    // Get location of the touch in view's coordinate system
    CGPoint location = [t locationInView:self];

    self.currentLine = [[BNRLine alloc] init];
    self.currentLine.begin = location;
    self.currentLine.end = location;

    [self setNeedsDisplay];
}
```

Then, in `BNRDrawView.m`, implement `touchesMoved:withEvent:` so that it updates the end of the `currentLine`.

```
- (void)touchesMoved:(NSSet *)touches
              withEvent:(UIEvent *)event
{
    UITouch *t = [touches anyObject];
    CGPoint location = [t locationInView:self];

    self.currentLine.end = location;

    [self setNeedsDisplay];
}
```

Finally, in `BNRDrawView.m`, add the `currentLine` to the `finishedLines` when the touch ends.

```
- (void)touchesEnded:(NSSet *)touches
              withEvent:(UIEvent *)event
{
    [self.finishedLines addObject:self.currentLine];

    self.currentLine = nil;

    [self 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 **BNRDrawView** 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 **BNRDrawView** will not be sent **touchesBegan:withEvent:** or any other **UIResponder** messages related to the extra touches.

In **BNRDrawView.m**, enable **BNRDrawView** instances to accept multiple touches.

```
- (instancetype)initWithFrame:(CGRect)r
{
    self = [super initWithFrame:r];
    if (self) {
        self.finishedLines = [[NSMutableArray alloc] init];
        self.backgroundColor = [UIColor grayColor];
        self.multipleTouchEnabled = YES;
    }
    return self;
}
```

Now that **BNRDrawView** 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 sends the message **anyObject** to 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 an **NSMutableDictionary** to hang on to each **BNRLine** 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 **BNRLine** in the dictionary.

In **BNRDrawView.m**, add a new instance variable to replace the **currentLine** and instantiate it in **initWithFrame:**.

```

@interface BNRDrawView ()

@property (nonatomic, strong) BNRLLine *currentLine;
@property (nonatomic, strong) NSMutableDictionary *linesInProgress;
@property (nonatomic, strong) NSMutableArray *finishedLines;

@end

@implementation BNRDrawView

- (instancetype)initWithFrame:(CGRect)r
{
    self = [super initWithFrame:r];

    if (self) {
        self.linesInProgress = [[NSMutableDictionary alloc] init];
        self.finishedLines = [[NSMutableArray alloc] init];
        self.backgroundColor = [UIColor grayColor];
        self.multipleTouchEnabled = YES;
    }

    return self;
}

```

Now you need to update the **UIResponder** methods to add lines that are currently being drawn to this dictionary. In `BNRDrawView.m`, update the code in `touchesBegan:withEvent:`:

```

- (void)touchesBegan:(NSSet *)touches
    withEvent:(UIEvent *)event
{
    // Let's put in a log statement to see the order of events
    NSLog(@"%@", NSStringFromSelector(_cmd));

    for (UITouch *t in touches) {
        CGPoint location = [t locationInView:self];

        BNRLLine *line = [[BNRLLine alloc] init];
        line.begin = location;
        line.end = location;

        NSValue *key = [NSValue valueWithNonretainedObject:t];
        self.linesInProgress[key] = line;
    }

    UITouch *t = [touches anyObject];
    CGPoint location = [t locationInView:self];

    self.currentLine = [[BNRLLine alloc] init];
    self.currentLine.begin = location;
    self.currentLine.end = location;

    [self setNeedsDisplay];
}

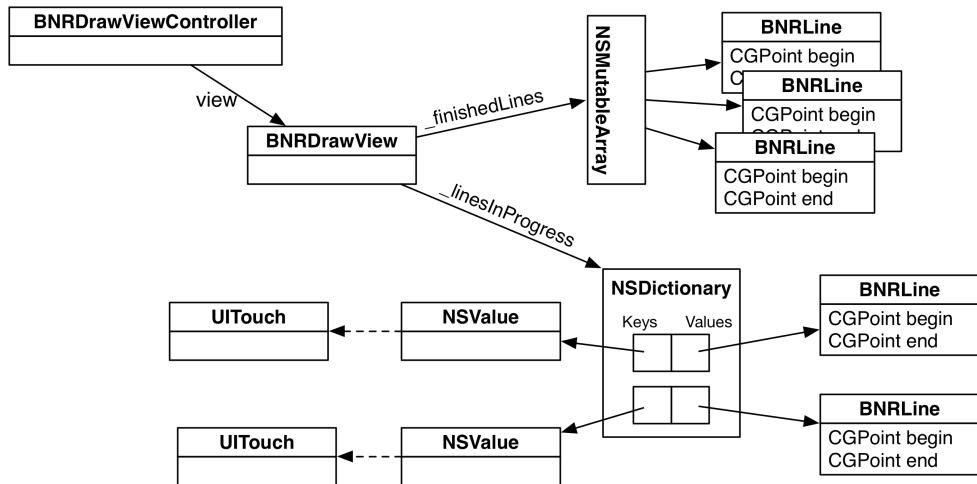
```

First, 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 **BNRDrawView** will receive multiple **touchesBegan:withEvent:** messages containing each touch.)

Next, notice the use of **valueWithNonretainedObject:** to derive the key to store the **BNRLine**. 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 12.4 Object diagram for Multitouch TouchTracker



Update **touchesMoved:withEvent:** in **BNRDrawView.m** so that it can look up the right **BNRLine**.

```

- (void)touchesMoved:(NSSet *)touches
                  withEvent:(UIEvent *)event
{
    // Let's put in a log statement to see the order of events
    NSLog(@"%@", NSStringFromSelector(_cmd));

    for (UITouch *t in touches) {
        NSValue *key = [NSValue valueWithNonretainedObject:t];
        BNRLine *line = self.linesInProgress[key];

        line.end = [t locationInView:self];
    }

    UITouch *t = [touches anyObject];
    CGPoint location = [t locationInView:self];
    self.currentLine.end = location;
    [self setNeedsDisplay];
}
    
```

Then, update **touchesEnded:withEvent:** to move any finished lines into the **_finishedLines** array.

```
- (void)touchesEnded:(NSSet *)touches
              withEvent:(UIEvent *)event
{
    // Let's put in a log statement to see the order of events
    NSLog(@"%@", NSStringFromSelector(_cmd));

    for (UITouch *t in touches) {
        NSValue *key = [NSValue valueWithNonretainedObject:t];
        BNRLLine *line = self.linesInProgress[key];

        [self.finishedLines addObject:line];
        [self.linesInProgress removeObjectForKey:key];
    }

    [self.finishedLines addObject:self.currentLine];
    self.currentLine = nil;
    [self setNeedsDisplay];
}
```

Finally, update **drawRect:** to draw each line in `_linesInProgress`.

```
// Draw finished lines in black
[[UIColor blackColor] set];
for (BNRLLine *line in self.finishedLines) {
    [self strokeLine:line];
}

[[UIColor redColor] set];
for (NSValue *key in self.linesInProgress) {
    [self strokeLine:self.linesInProgress[key]];
}

if (self.currentLine) {
    // Draw line in progress in red
    [[UIColor redColor] set];
    [self strokeLine:self.currentLine];
}
}
```

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.)

You may be wondering: why not use the **UITouch** itself as the key? Why go through the hoop of creating an **NSValue**? Objects used as keys in an **NSDictionary** must conform to the **NSCopying** protocol, which allows them to be copied by sending the message **copy**. **UITouch** instances do not conform to this protocol because it does not make sense for them to be copied. Thus, the **NSValue** instances hold the address of the **UITouch** so that equal **NSValue** instances can be later created with the same **UITouch**.

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 `BNRDrawView.m`, implement `touchesCancelled:withEvent:`:

```
- (void)touchesCancelled:(NSSet *)touches
                  withEvent:(UIEvent *)event
{
    // Let's put in a log statement to see the order of events
    NSLog(@"%@", NSStringFromSelector(_cmd));

    for (UITouch *t in touches) {
        NSValue *key = [NSValue valueWithNonretainedObject:t];
        [self.linesInProgress removeObjectForKey:key];
    }

    [self 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 `_finishedLines`.

Gold Challenge: Circles

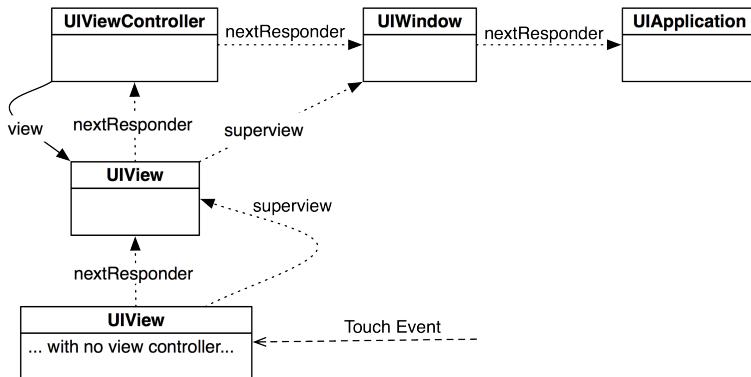
Use two fingers to draw circles. Try having each finger represent one corner of the bounding box around the circle. You can simulate two fingers on the simulator by holding down the Option button. (Hint: This is much easier if you track touches that are working on a circle in a separate dictionary.)

For the More Curious: The Responder Chain

In Chapter 7, 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 pointer called `nextResponder`, and together these objects make up the responder chain (Figure 12.5). 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 12.5 Responder chain



How does a **UIResponder** *not* handle an event? It forwards the same message to its `nextResponder`. 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:

```

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    if (touch.tapCount == 2) {
        [[self nextResponder] touchesBegan:touches withEvent:event];
        return;
    }
    ...
    ... 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 `UIControlEventTouchUpInside` 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:

```
[rButton addTarget:tempController
            action:@selector(resetTemperature:)
            forControlEvents:UIControlEventTouchUpInside | UIControlEventTouchUpOutside];
```

Now consider how **UIControl** handles **UIControlEventTouchUpInside**.

```
// Not the exact code. There is a bit more going on!
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    // Reference to the touch that is ending
    UITouch *touch = [touches anyObject];

    // Location of that point in this control's coordinate system
    CGPoint 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!
        [self sendActionsForControlEvents:UIControlEventTouchUpInside];
    } else {
        // The touch ended outside the bounds, different control event
        [self sendActionsForControlEvents:UIControlEventTouchUpInsideOutside];
    }
}
```

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.

This page intentionally left blank

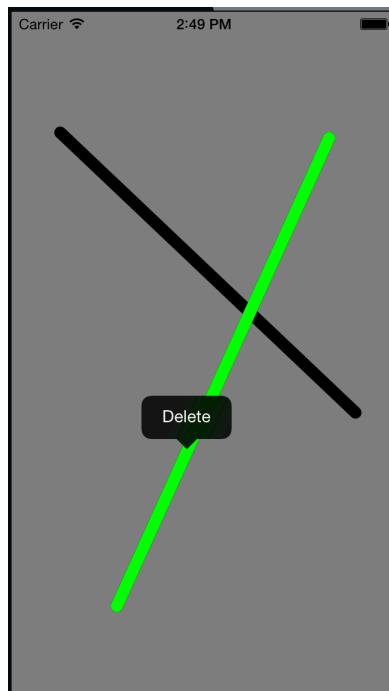
13

UIGestureRecognizer and UIMenuController

In Chapter 12, 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 13.1). You will also see how to use another interesting iOS class, **UIMenuController**.

Figure 13.1 TouchTracker by the end of the chapter



UIGestureRecognizer Subclasses

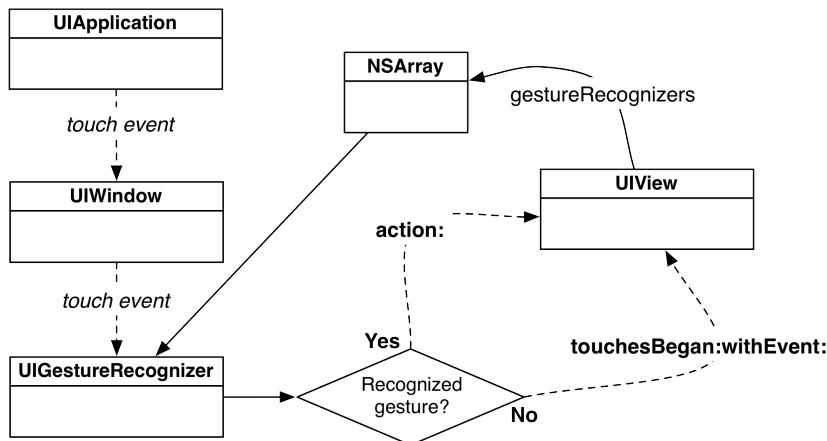
You do not instantiate **UIGestureRecognizer** itself. Instead, there are a number of subclasses of **UIGestureRecognizer**, and each one is responsible for recognizing a particular gesture.

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:

```
- (void)action:(UIGestureRecognizer *)gestureRecognizer;
```

When recognizing a gesture, the gesture recognizer intercepts the touches destined for the view (Figure 13.2). Thus, a view with gesture recognizers may not receive the typical **UIResponder** messages like **touchesBegan:withEvent:**.

Figure 13.2 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 12.

In **BNRDrawView.m**, instantiate a **UITapGestureRecognizer** that requires two taps to fire in **initWithFrame:**.

```

- (instancetype)initWithFrame:(CGRect)r
{
    self = [super initWithFrame:r];

    if (self) {
        self.linesInProgress = [[NSMutableDictionary alloc] init];
        self.finishedLines = [[NSMutableArray alloc] init];
        self.backgroundColor = [UIColor grayColor];
        self.multipleTouchEnabled = YES;

        UITapGestureRecognizer *doubleTapRecognizer =
            [[UITapGestureRecognizer alloc] initWithTarget:self
                                              action:@selector(doubleTap:)];
        doubleTapRecognizer.numberOfTapsRequired = 2;

        [self addGestureRecognizer:doubleTapRecognizer];
    }
    return self;
}

```

When a double tap occurs on an instance of **BNRDrawView**, the message **doubleTap:** will be sent to that instance. Implement this method in **BNRDrawView.m**.

```

- (void)doubleTap:(UIGestureRecognizer *)gr
{
    NSLog(@"Recognized Double Tap");

    [self.linesInProgress removeAllObjects];
    [self.finishedLines removeAllObjects];
    [self setNeedsDisplay];
}

```

Notice that the argument to the action method for a gesture recognizer is the instance of **UIGestureRecognizer** that sent the message. 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 sent to the **BNRDrawView** on the first tap, creating a small line. Check the console and you will see the following sequence of events:

```

touchesBegan:withEvent:
Recognized 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 **BNRDrawView.m**, modify **initWithFrame:** to do just this.

```
UITapGestureRecognizer *doubleTapRecognizer =
    [[UITapGestureRecognizer alloc] initWithTarget:self
                                            action:@selector(doubleTap:)];
doubleTapRecognizer.numberOfTapsRequired = 2;
doubleTapRecognizer.delaysTouchesBegan = YES;
[self addGestureRecognizer:doubleTapRecognizer];
}
return self;
}
```

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 **BNRDrawView** that only requires one tap.

In **BNRDrawView.m**, modify **initWithFrame:**.

```
[self addGestureRecognizer:doubleTapRecognizer];

UITapGestureRecognizer *tapRecognizer =
    [[UITapGestureRecognizer alloc] initWithTarget:self
                                            action:@selector(tap:)];
tapRecognizer.delaysTouchesBegan = YES;
[self addGestureRecognizer:tapRecognizer];
}
return self;
}
```

Now, implement **tap:** to log the tap to the console in **BNRDrawView.m**.

```
- (void)tap:(UIGestureRecognizer *)gr
{
    NSLog(@"Recognized 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 `initWithFrame:`, 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.

```
UITapGestureRecognizer *tapRecognizer =
    [[UITapGestureRecognizer alloc] initWithTarget:self
                                            action:@selector(tap:)];
tapRecognizer.delaysTouchesBegan = YES;
[tapRecognizer requireGestureRecognizerToFail:doubleTapRecognizer];
[self 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 `BNRDrawView` so that the user can select lines when they are tapped. First, add a property to hold onto a selected line to the class extension in `BNRDrawView.m`.

```
@interface BNRDrawView ()
@property (nonatomic, strong) NSMutableDictionary *linesInProgress;
@property (nonatomic, strong) NSMutableArray *finishedLines;

@property (nonatomic, weak) BNRLLine *selectedLine;
@end
```

(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.

```
[[UIColor redColor] set];
for (NSValue *key in self.linesInProgress) {
    [self strokeLine:self.linesInProgress[key]];
}

if (self.selectedLine) {
    [[UIColor greenColor] set];
    [self strokeLine:self.selectedLine];
}
```

Implement `lineAtPoint:` in `BNRDrawView.m` to get a `BNRLLine` close to the given point.

```
- (BNRLine *)lineAtPoint:(CGPoint)p
{
    // Find a line close to p
    for (BNRLine *l in self.finishedLines) {
        CGPoint start = l.begin;
        CGPoint end = l.end;

        // Check a few points on the line
        for (float t = 0.0; t <= 1.0; t += 0.05) {
            float x = start.x + t * (end.x - start.x);
            float y = start.y + t * (end.y - start.y);

            // If the tapped point is within 20 points, let's return this line
            if (hypot(x - p.x, y - p.y) < 20.0) {
                return l;
            }
        }
    }

    // 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. Sending this message to 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 `BNRDrawView.m`, send the `locationInView:` message to the gesture recognizer, pass the result to `lineAtPoint:`, and make the returned line the `selectedLine`.

```
- (void)tap:(UIGestureRecognizer *)gr
{
    NSLog(@"Recognized tap");

    CGPoint point = [gr locationInView:self];
    self.selectedLine = [self lineAtPoint:point];

    [self 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 13.3). 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 13.3 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 `BNRDrawView.m`'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.

```
- (void)tap:(UIGestureRecognizer *)gr
{
    NSLog(@"Recognized tap");

    CGPoint point = [gr locationInView:self];
    self.selectedLine = [self lineAtPoint:point];

    if (self.selectedLine) {

        // Make ourselves the target of menu item action messages
        [self becomeFirstResponder];

        // Grab the menu controller
        UIMenuController *menu = [UIMenuController sharedMenuController];

        // Create a new "Delete" UIMenuItem
        UIMenuItem *deleteItem = [[UIMenuItem alloc] initWithTitle:@"Delete"
                                                       action:@selector(deleteLine:)];
        menu.menuItems = @[deleteItem];

        // Tell the menu where it should come from and show it
        [menu setTargetRect:CGRectMake(point.x, point.y, 2, 2) inView:self];
        [menu setMenuVisible:YES animated:YES];
    } else {
        // Hide the menu if no line is selected
        [[UIMenuController sharedMenuController] setMenuVisible:NO animated:YES];
    }
    [self 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 sent the message `becomeFirstResponder` to the `BNRDrawView` 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 `BNRDrawView.m`, override this method to return YES.

```
- (BOOL)canBecomeFirstResponder
{
    return YES;
}
```

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 **BNRDrawView.m**.

```
- (void)deleteLine:(id)sender
{
    // Remove the selected line from the list of _finishedLines
    [self.finishedLines removeObject:self.selectedLine];

    // Redraw everything
    [self 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 **BNRDrawView.m**, instantiate a **UILongPressGestureRecognizer** in **initWithFrame:** and add it to the **BNRDrawView**.

```
[self addGestureRecognizer:tapRecognizer];

UILongPressGestureRecognizer *pressRecognizer =
    [[UILongPressGestureRecognizer alloc] initWithTarget:self
                                                action:@selector(longPress:)];
[self addGestureRecognizer:pressRecognizer];
```

Now when the user holds down on the **BNRDrawView**, the message **longPress:** will be sent to 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 *began*. 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 **UIGestureRecognizerStatePossible**, then **UIGestureRecognizerStateBegan**, and finally **UIGestureRecognizerStateEnded**.

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 `BNRDrawView.m`, implement `longPress:`.

```
- (void)longPress:(UIGestureRecognizer *)gr
{
    if (gr.state == UIGestureRecognizerStateBegan) {
        CGPoint point = [gr locationInView:self];
        self.selectedLine = [self lineAtPoint:point];

        if (self.selectedLine) {
            [self.linesInProgress removeAllObjects];
        }
    } else if (gr.state == UIGestureRecognizerStateChanged) {
        self.selectedLine = nil;
    }
    [self 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 the class extension in `BNRDrawView.m`, declare that `BNRDrawView` conforms to the `UIGestureRecognizerDelegate` protocol. Then, declare a **UIPanGestureRecognizer** as a property so that you have access to it in all of your methods.

```
@interface BNRDrawView () <UIGestureRecognizerDelegate>

@property (nonatomic, strong) UIPanGestureRecognizer *moveRecognizer;
@property (nonatomic, strong) NSMutableDictionary *linesInProgress;
@property (nonatomic, strong) NSMutableArray *finishedLines;

@property (nonatomic, weak) BNRLLine *selectedLine;

@end
```

In BNRDrawView.m, add code to **initWithFrame:** to instantiate a **UIPanGestureRecognizer**, set two of its properties, and attach it to the **BNRDrawView**.

```
[self addGestureRecognizer:pressRecognizer];  
  
self.moveRecognizer = [[UIPanGestureRecognizer alloc] initWithTarget:self  
                                         action:@selector(moveLine:)];  
self.moveRecognizer.delegate = self;  
self.moveRecognizer.cancelsTouchesInView = NO;  
[self addGestureRecognizer:self.moveRecognizer];
```

There are a number of methods in the **UIGestureRecognizerDelegate** protocol, but you are only interested in one – **gestureRecognizer:shouldRecognizeSimultaneouslyWithGestureRecognizer:**. A gesture recognizer will send this message to its delegate when it recognizes its gesture but realizes that another gesture recognizer has recognized its gesture, too. If this method returns YES, the recognizer will share its touches with other gesture recognizers.

In BNRDrawView.m, return YES when the **_moveRecognizer** sends the message to its delegate.

```
- (BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer  
    shouldRecognizeSimultaneouslyWithGestureRecognizer:(UIGestureRecognizer *)other  
{  
    if (gestureRecognizer == self.moveRecognizer) {  
        return YES;  
    }  
    return NO;  
}
```

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 BNRDrawView.m, 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 pointer to an instance of **UIPanGestureRecognizer** rather than **UIGestureRecognizer**.

```

- (void)moveLine:(UIPanGestureRecognizer *)gr
{
    // If we have not selected a line, we do not do anything here
    if (!self.selectedLine) {
        return;
    }

    // When the pan recognizer changes its position...
    if (gr.state == UIGestureRecognizerStateChanged) {
        // How far has the pan moved?
        CGPoint translation = [gr translationInView:self];

        // Add the translation to the current beginning and end points of the line
        CGPoint begin = self.selectedLine.begin;
        CGPoint end = self.selectedLine.end;
        begin.x += translation.x;
        begin.y += translation.y;
        end.x += translation.x;
        end.y += translation.y;

        // Set the new beginning and end points of the line
        self.selectedLine.begin = begin;
        self.selectedLine.end = end;

        // Redraw the screen
        [self setNeedsDisplay];
    }
}

```

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 `BNRDrawView.m`, add the following line of code.

```

    [self setNeedsDisplay];

    [gr setTranslation:CGPointZero inView:self];
}
}

```

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 YES. 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 NO, 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 NO 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 YES or NO 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 NO.

```
- (BOOL)canPerformAction:(SEL)action withSender:(id)sender
{
    if (action == @selector(copy:))
        return NO;

    // The superclass's implementation will return YES if the method is in the .m file
    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:

- `UIGestureRecognizerStatePossible`
- `UIGestureRecognizerStateBegan`
- `UIGestureRecognizerStateChanged`
- `UIGestureRecognizerStateEnded`
- `UIGestureRecognizerStateFailed`
- `UIGestureRecognizerStateCancelled`
- `UIGestureRecognizerStateRecognized`

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 three built-in recognizers you did not implement in this chapter are `UIPinchGestureRecognizer`, `UISwipeGestureRecognizer`, 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.

This page intentionally left blank

14

Debugging Tools

In Chapter 7, 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 14.1 Gauges

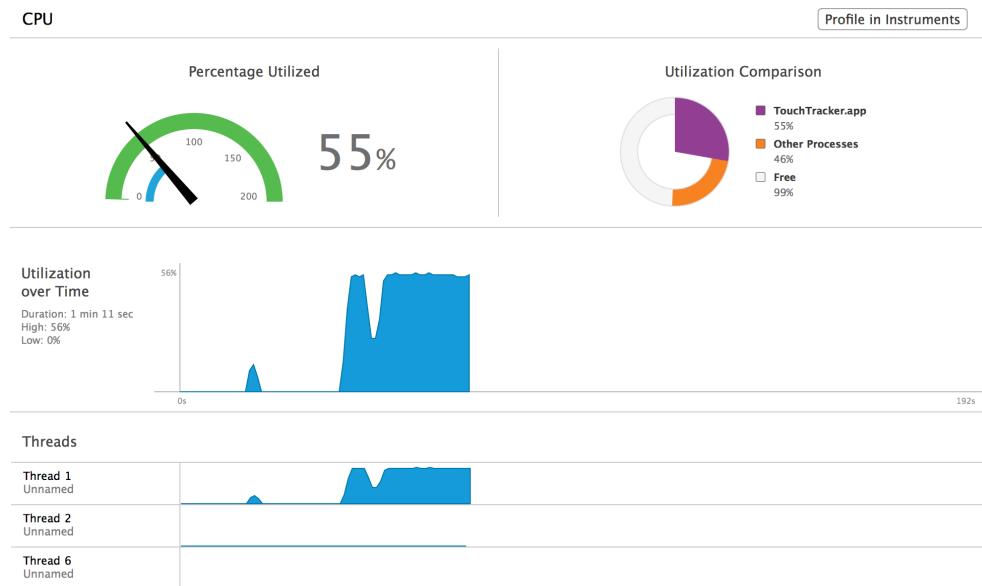


While the application is running (not paused or crashed), the debug navigator shows CPU and memory gauges (Figure 14.1). Each of these shows a live-updating graph of resource usage over time, as well as a numerical figure describing that resource's current usage.

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 14.2).

Figure 14.2 CPU report



The report contains four basic sections:

Percentage Utilized

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%.

Utilization 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.

Utilization 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

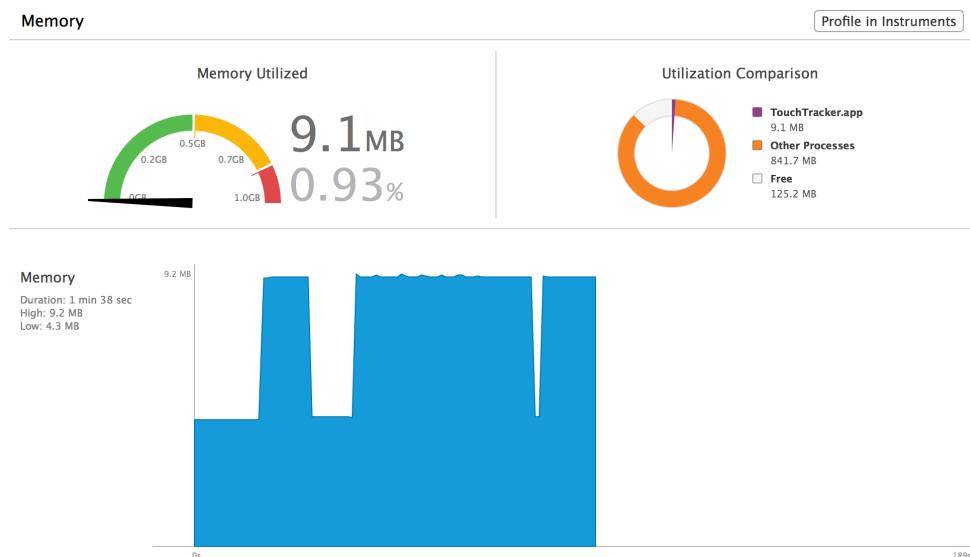
shows a the breakdown of the Utilization over Time graph on a per-thread basis. Multithreading is outside the scope of this book, but this information will become useful to you as you continue your iOS development education and career.

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 sent to your `BNRDrawView` 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 14.3).

Figure 14.3 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.

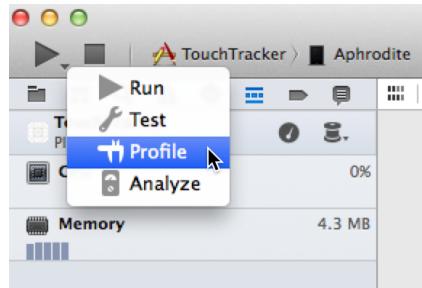
Instruments is an application bundled with Xcode that you can use to monitor your application while it is running and gather fine-grained statistics about your application's performance. Instruments is made up of several plug-ins that enable you to inspect object allocations, CPU utilization per function 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.

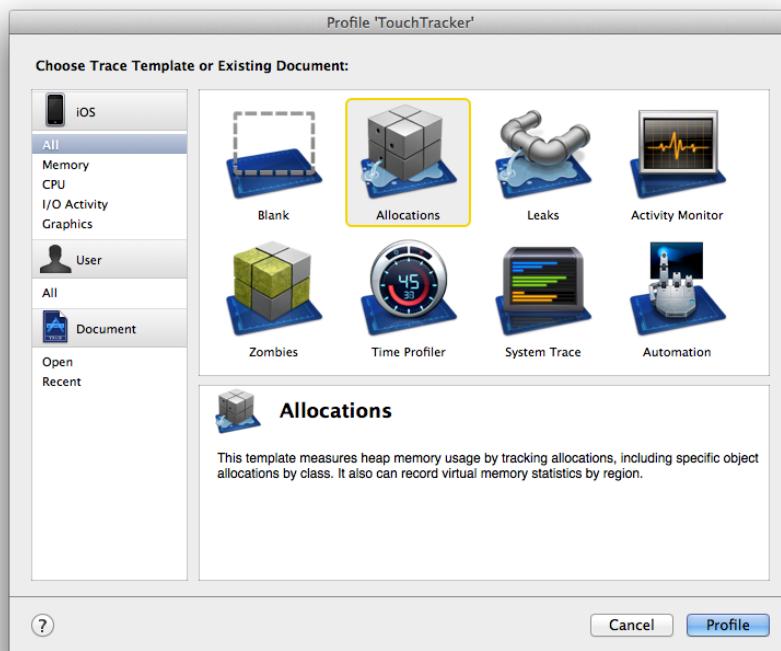
To profile an application, click and hold the Run button in the top left corner of the workspace. In the pop-up menu that appears, select Profile (Figure 14.4).

Figure 14.4 Profiling an application



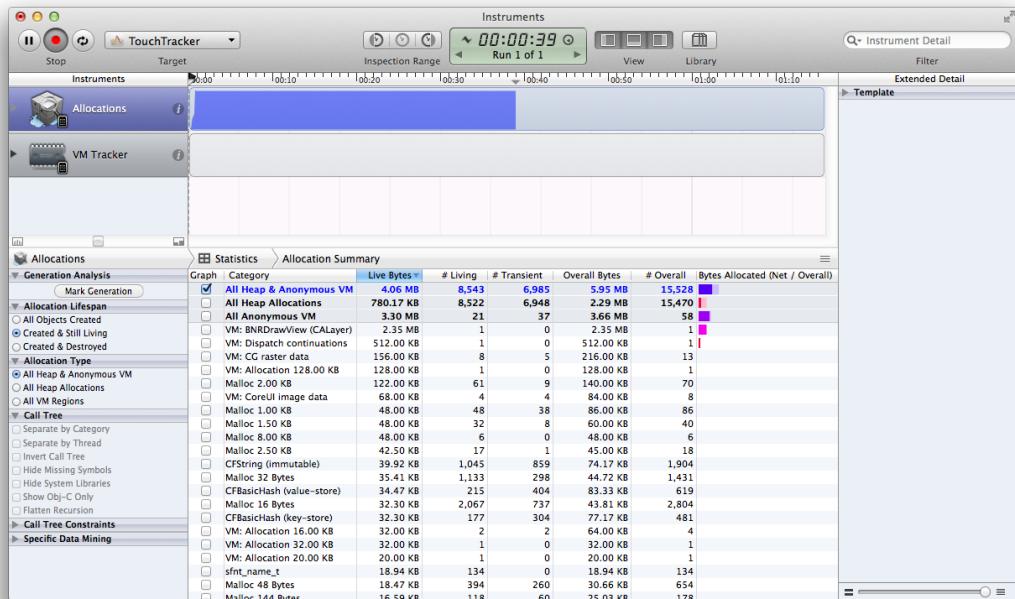
Instruments will launch and ask which instrument template to use. Note that there are more than eight choices; you will see more if you scroll down. Choose Allocations and click Profile (Figure 14.5).

Figure 14.5 Choosing an instrument



TouchTracker will launch, and a window will open in Instruments (Figure 14.6). 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 all three buttons to reveal the three main areas. The window should look like Figure 14.6.

Figure 14.6 Allocations instrument



This 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 **BNRLLine** 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 **BNRLLine** (Figure 14.7).

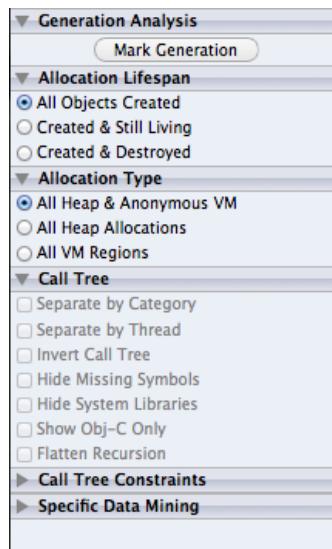
Figure 14.7 Allocated lines

| Graph | Category | Live Bytes | # Living | # Transient | Overall Bytes | # Overall | Bytes Allocated (Net / Overall) |
|--------------------------|----------|------------|----------|-------------|---------------|-----------|---------------------------------|
| <input type="checkbox"/> | BNRLLine | 128 Bytes | 4 | 0 | 128 Bytes | 4 | |

The # Living column shows you how many line objects are currently allocated. Live Bytes shows how much memory these living instances take up. The # Overall 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 **BNRLine** 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 Objects Created from the Allocation Lifespan section of the lefthand panel (Figure 14.8).

Figure 14.8 Allocations options



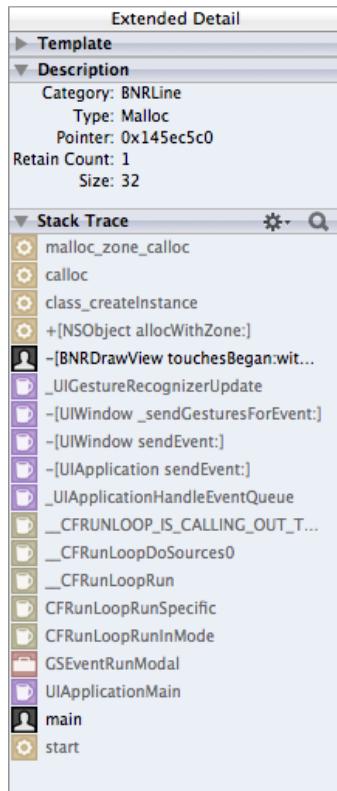
Let's see what else the Allocations instrument can tell you about your lines. First, draw a few more lines in TouchTracker. Then, in the table, select the row that says **BNRLine**. An arrow will appear in the Category column; click that arrow to see more details about these allocations (Figure 14.9).

Figure 14.9 **BNRLine** summary

| BNRLine | | | | | | | |
|---------|------------|----------|---------------|------|----------|---------------------|----------------------------|
| # | Address | Category | Timestamp | Live | Size | Responsible Library | Responsible Caller |
| 0 | 0x145ec5c0 | BNRLine | 02:57.539.569 | • | 32 Bytes | TouchTracker | -[BNRDrawView touchesBe... |
| 1 | 0x145cf0c0 | BNRLine | 02:58.552.178 | • | 32 Bytes | TouchTracker | -[BNRDrawView touchesBe... |
| 2 | 0x1468a070 | BNRLine | 02:59.152.489 | • | 32 Bytes | TouchTracker | -[BNRDrawView touchesBe... |
| 3 | 0x145cf0a0 | BNRLine | 03:00.954.103 | • | 32 Bytes | TouchTracker | -[BNRDrawView touchesBe... |

Each row in this table shows a single instance of **BNRLine** 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 14.10). This stack trace shows you where that instance of **BNRLine** was created. Grayed-out items in the trace are system library calls. Items in black text are your code. Find the top-most item that is your code (`-[BNRDrawView touchesBegan:withEvent:]`) and double-click it.

Figure 14.10 Stack trace



The source code for this implementation will replace the table of **BNRLLine** instances (Figure 14.11). The percentages you see are the amount of memory these method calls allocate compared to the other calls in **touchesBegan:withEvent:**. For example, the **BNRLLine** instance makes up 0.2 percent of the memory allocated by **touchesBegan:withEvent:**, whereas the **NSLog** allocates a considerable amount of memory relative to the creation of the line and the two **NSValue** objects (the one you create and the copy made by using it as the key in `self.linesInProgress`).

Figure 14.11 Source code in Instruments

```

159 - (void)touchesBegan:(NSSet *)touches
160     withEvent:(UIEvent *)event
161 {
162     // Let's put in a log statement to see the order of
163     NSLog(@"%@", NSStringFromSelector(_cmd)); ⚡ 99.0%
164
165     for(UITouch *t in touches) {
166         CGPoint location = [t locationInView:self];
167         BNRLLine *line = [[BNRLLine alloc] init]; ⚡ 0.2%
168         line.begin = location; ⚡ 0.1%
169         line.end = location;
170         NSValue *key = [NSValue valueWithNonretainedObject:t]; ⚡ 0.5%
171         ;
172         self.linesInProgress[key] = line; ⚡ 0.1%
173     }
174
175     [self setNeedsDisplay];
176 }

```

Notice that above the summary area is a breadcrumb navigation bar (Figure 14.12). You can click on an item in this bar to return to a previous set of information.

Figure 14.12 Navigation for summary area



Click on the **BNRLLine** item in the breadcrumb bar to get back to the list of all **BNRLLine** 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 **BNRLLine** 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 14.13).

Figure 14.13 Generation analysis

| Generation Analysis | Snapshot | Timestamp | Growth | # Persistent |
|-------------------------|---------------------------------|---------------|-----------|--------------|
| Mark Generation | Generation A | 00:06.670.257 | 4.08 MB | 8,736 |
| Allocation Lifespan | Generation B | 00:15.766.416 | 1.48 KB | 30 |
| All Objects Created | ▶ non-object > | | 776 Bytes | 13 |
| Created & Still Living | ▶ IOHIDEvent | | 320 Bytes | 2 |
| Created & Destroyed | ▶ CFArray (store-deque) | | 96 Bytes | 3 |
| All Heap & Anonymous VM | ▶ CFArray (mutable-variable) | | 96 Bytes | 3 |
| All Heap Allocations | ▶ UIGestureRecognizerFailureMap | | 64 Bytes | 2 |
| All VM Regions | ▶ CFDictionary (mutable) | | 48 Bytes | 1 |
| Call Tree | ▶ CFBasicHash (key-store) | | 32 Bytes | 2 |
| Separate by Category | ▶ CFBasicHash (value-store) | | 32 Bytes | 2 |
| Separate by Thread | ▶ BNRLLine | | 32 Bytes | 1 |
| Invert Call Tree | ▶ CGRegion | | 16 Bytes | 1 |

Every allocation that took place after the first generation is in this category. You can see the **BNRLine** instances 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

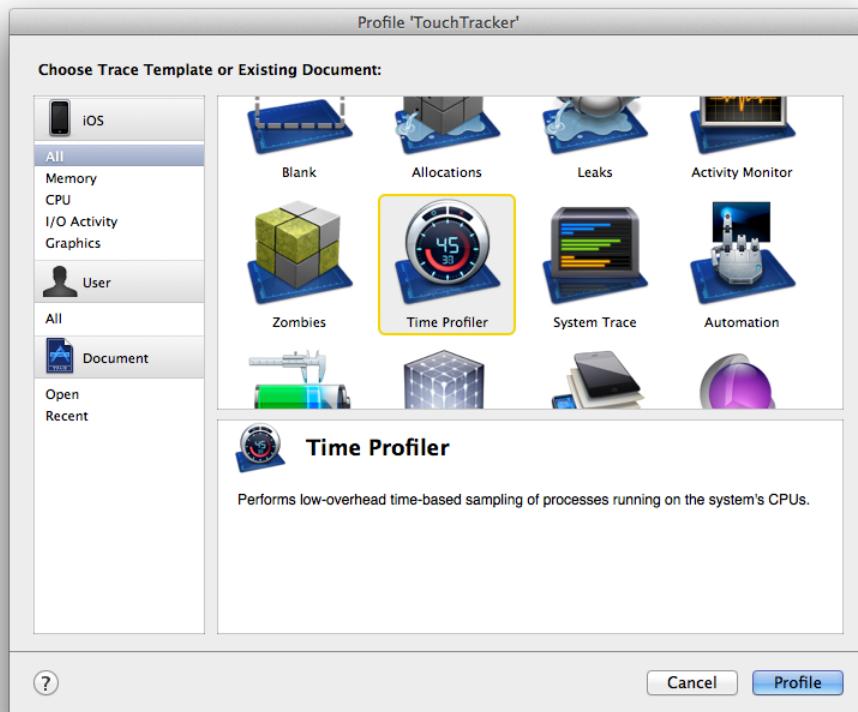
The Time Profiler instrument provides exhaustive statistics about the CPU utilization of your application. Right now, TouchTracker does not abuse the CPU enough to provide meaningful results.

In `BNRDrawView.m`, make things more interesting by adding the following CPU cycle-wasting code to the end of your **drawRect:** method:

```
float f = 0.0;
for (int i = 0; i < 1000000; i++) {
    f = f + sin(sin(sin(time(NULL) + i)));
}
NSLog(@"%@", f);
```

Build and profile the application. When Instruments asks which instrument to use, choose Time Profiler (Figure 14.14). 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.

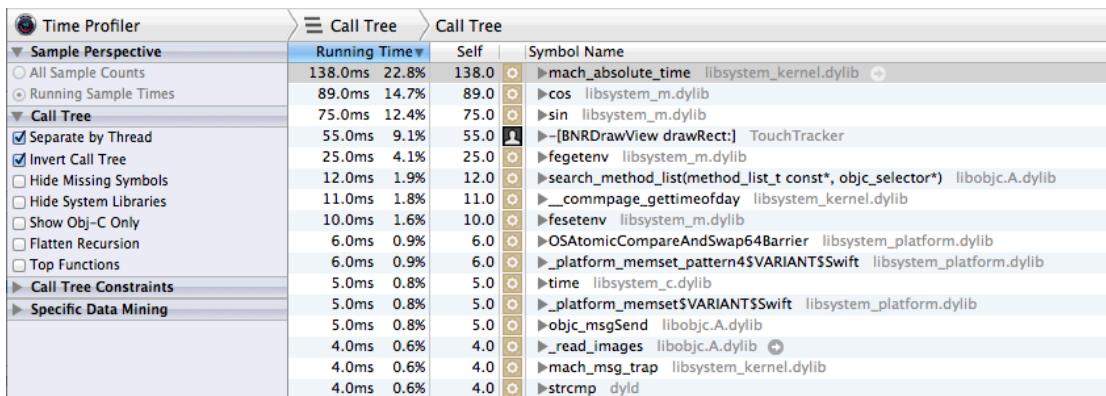
Figure 14.14 Time Profiler instrument



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 `BNRDrawView`. Each `touchesMoved:withEvent:` message causes `drawRect:` to be sent, 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 lefthand 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 14.15). This gives you an idea of where your application is spending its execution time.

Figure 14.15 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.

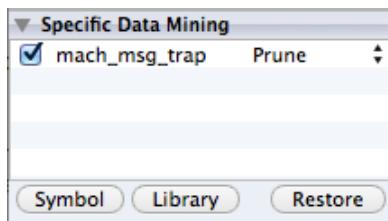
You know that when drawing a line, two things are happening: `touchesMoved:withEvent:` and `drawRect:` are being sent to the `BNRDrawView` view. In Time Profiler, you can check to see how much time is spent in these two methods relative to the rest of the application. If an inordinate amount of time 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 left side of the screen, click the Symbol button under Specific Data Mining. The `mach_msg_trap` function appears in the table under Specific 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 Specific Data Mining table to add it back to the total time.

Figure 14.16 Pruning a symbol



Other options for reducing the list of symbols in Time Profiler include showing only Objective-C calls, hiding system libraries, 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 Specific 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, `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. It 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.

As an example, an overzealous Objective-C developer might be tempted to create classes for things like vectors, points, and rectangles. These classes would likely have methods to add, subtract, or multiply instances as well as accessor methods to get and set instance variables. When these classes are used for drawing, the code has to send a lot of messages to do something simple, like creating two vectors and adding them together. The messages add excessive overhead considering the simplicity of the operation. Therefore, the better alternative is to create data types like these as structures and access their memory directly. (This is why `CGRect` and `CGPoint` are structures and not Objective-C classes.)

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 `BNRLine` needs to know what array of lines it belongs to. Add a new property to `BNRLine.h`:

```
@property (nonatomic, strong) NSMutableArray *containingArray;
```

In `BNRDrawView.m`, set every completed line's `containingArray` property in `touchesEnded:withEvent:`.

```
- (void)touchesEnded:(NSSet *)touches
              withEvent:(UIEvent *)event
{
    // Remove ending touches from dictionary
    for (UITouch *t in touches) {
        NSValue *key = [NSValue valueWithNonretainedObject:t];
        BNRLLine *line = self.linesInProgress[key];

        [self.finishedLines addObject:line];
        [self.linesInProgress removeObjectForKey:key];

        line.containingArray = self.finishedLines;
    }

    // Redraw
    [self setNeedsDisplay];
}
```

Finally, in `doubleTap:` of `BNRDrawView.m`, comment out the code that removes all of the objects from `self.finishedLines` and create a new instance of `NSMutableArray` instead.

```
- (void)doubleTap:(UIGestureRecognizer *)gr
{
    NSLog(@"Recognized Double Tap");

    [self.linesInProgress removeAllObjects];
    // [self.finishedLines removeAllObjects];

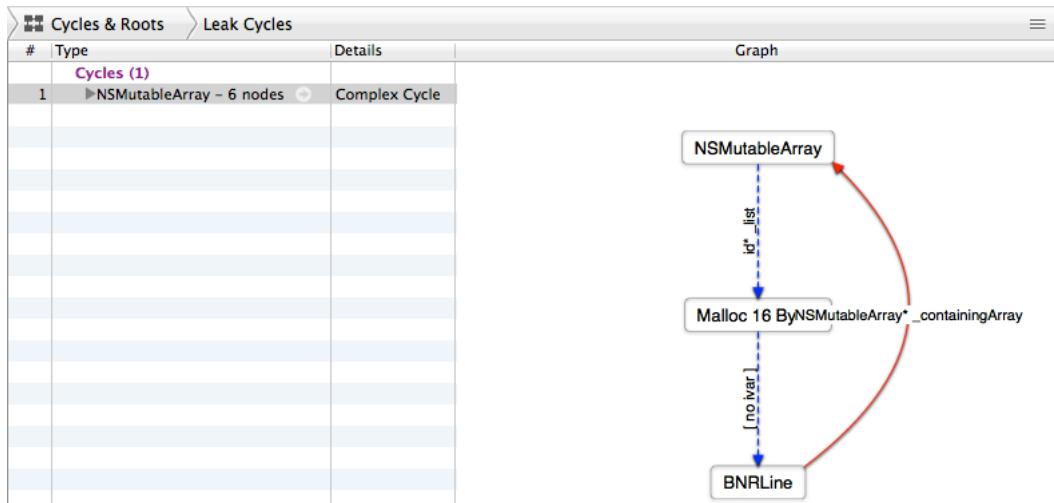
    self.finishedLines = [[NSMutableArray alloc] init];
    [self setNeedsDisplay];
}
```

Build and profile the application. Choose Leaks as the instrument to use.

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. Three items will appear in the summary table: an `NSMutableArray`, a few `BNRLLine` instances, and a Malloc 16 Bytes block. This memory has been leaked.

Select the Leaks pop-up button in the breadcrumb bar and change it to *Cycles & Roots* (Figure 14.17). This view gives you a lovely graphical representation of the strong reference cycle: an `NSMutableArray` (your `self.finishedLines` array) has a reference to a list of `BNRLLine` objects, and each `BNRLLine` has a reference back to its `containingArray`.

Figure 14.17 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.

Static Analyzer

Instruments can be helpful when tracking down a problem in a running application. You can also ask Xcode to analyze your code without running it. The static analyzer is a tool that can make educated guesses about what would happen if your code were to be executed and inform you of potential problems.

When the static analyzer checks the code, it examines each function and method individually by iterating over every possible *code path*. A method can have a number of control statements (`if`, `for`, `switch`, etc.). The conditions of these statements will dictate which code is actually executed. A code path is one of the possible paths the code will take given these control statements. For example, a method that has a single `if` statement has two code paths: one if the condition fails and one if the condition succeeds.

Right now, TouchTracker does not have any code that offends the static analyzer. Add some: In `BNRDrawView.m`, implement the following method:

```

- (int)numberOfLines
{
    int count;

    // Check that they are non-nil before we add their counts...
    if (self.linesInProgress && self.finishedLines)
        count = [self.linesInProgress count] + [self.finishedLines count];

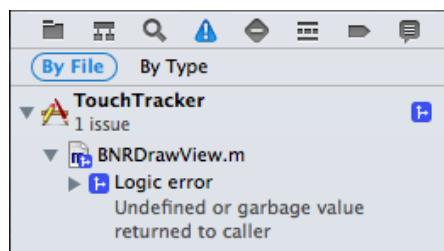
    return count;
}

```

To run the static analyzer, click and hold the Run button (like you did when you profiled your application). This time, choose Analyze. Alternatively, you can use the keyboard shortcut Command-Shift-B.

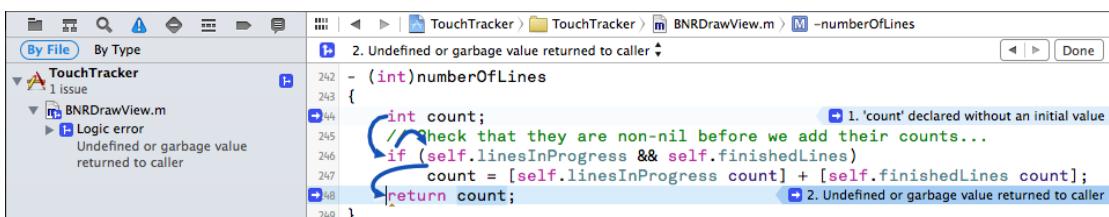
Analysis results appear in the issue navigator (Figure 14.18). You will see one Logic error in your code at the return point of `numberOfLines`. The analyzer believes there is a code path that will result in an undefined or garbage value being returned to the caller. In English, that means it is possible that the variable `count` will not be given a value before it is returned from `numberOfLines`.

Figure 14.18 Analyzer results



The analyzer can show you how it came to this conclusion. Click the disclosure button next to the analyzer result to reveal the detailed information underneath it. Click the item underneath the disclosure button. In the editor area, curvy blue lines will appear inside the `numberOfLines` method (Figure 14.19). (If you do not see line numbers in the gutter, you can turn them on by selecting Preferences from the Xcode menu. Choose the Text Editing tab and click the checkbox Show Line Numbers.)

Figure 14.19 Expanded analysis



The code path shown by the analyzer lines is as follows:

1. The variable `count` is created and not initialized.
2. The `if` statement fails, so `count` does not get a value.
3. The variable `count` is returned without being assigned a value.

You can fix this issue by initializing count to zero.

```
{  
    int count;  
    int count = 0;  
  
    // Check that they are non-nil before we add their counts...  
    if (self.linesInProgress && self.finishedLines)  
        count = [self.linesInProgress count] + [self.finishedLines count];  
  
    return count;  
}
```

Analyze this code again, and no issues will be reported now that count is always initialized with a value.

When you analyze your code (which smart programmers do on a regular basis), you will see issues other than the one described here. Many times, we see novice programmers shy away from analyzer issues because of the technical language. Do not do this. Take the time to expand the analysis and understand what the analyzer is trying to tell you. It will be worth it for the development of your application and for your development as a programmer.

Projects, Targets, and Build Settings

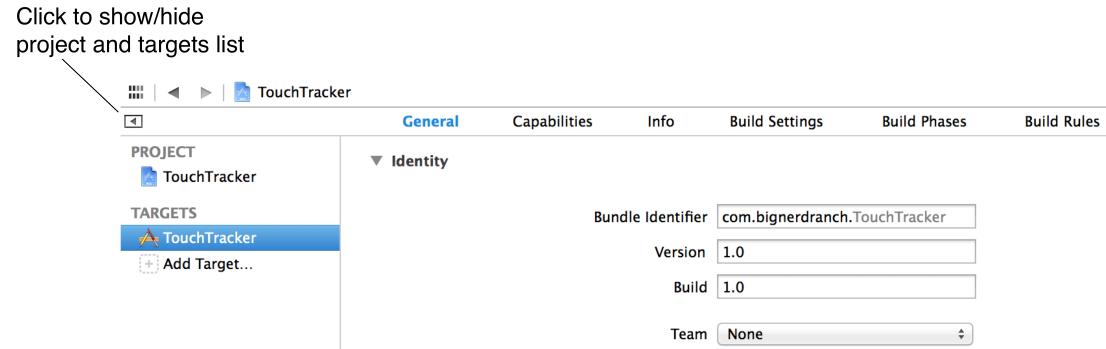
A Xcode project is a file that contains a list of references to other files (source code, resources, frameworks, and libraries) as well as a number of settings that lay out the rules for items within the project. Projects end in .xcodeproj, as in TouchTracker.xcodeproj.

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 14.20). Click this button to show the project and targets list for TouchTracker.

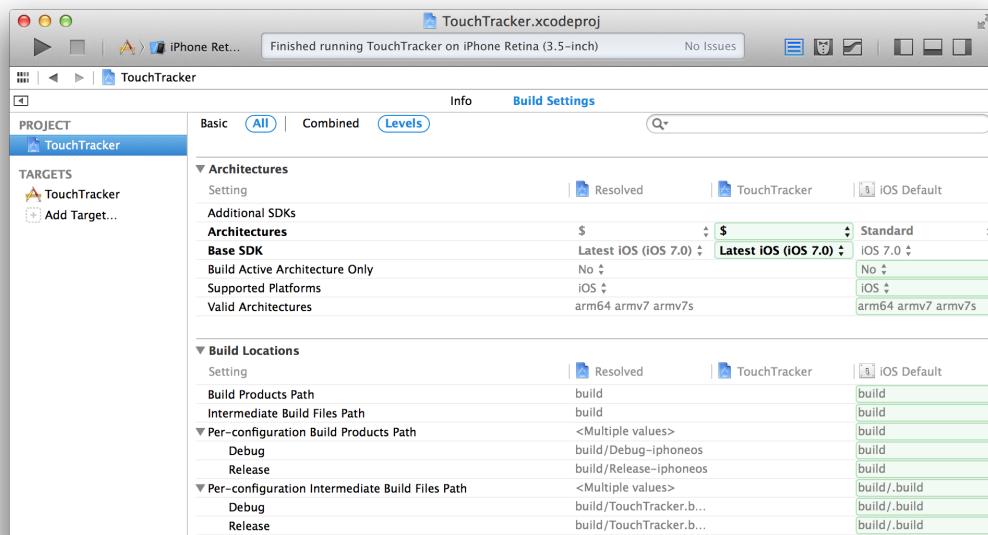
Figure 14.20 TouchTracker project and targets list



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 (Figure 14.21).

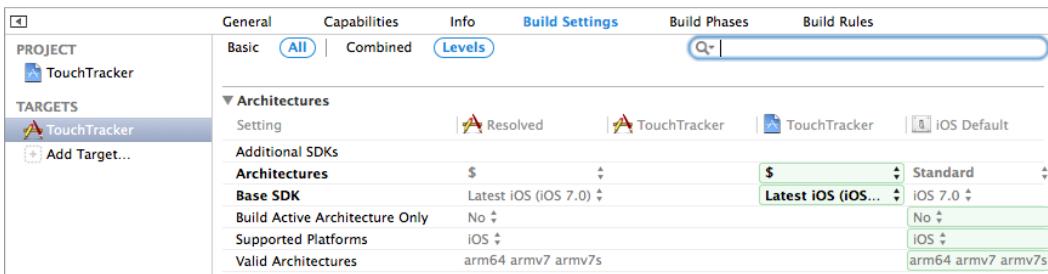
Figure 14.21 TouchTracker project build settings



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. Above the list of settings, find and click the Levels option (Figure 14.22).

Figure 14.22 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.

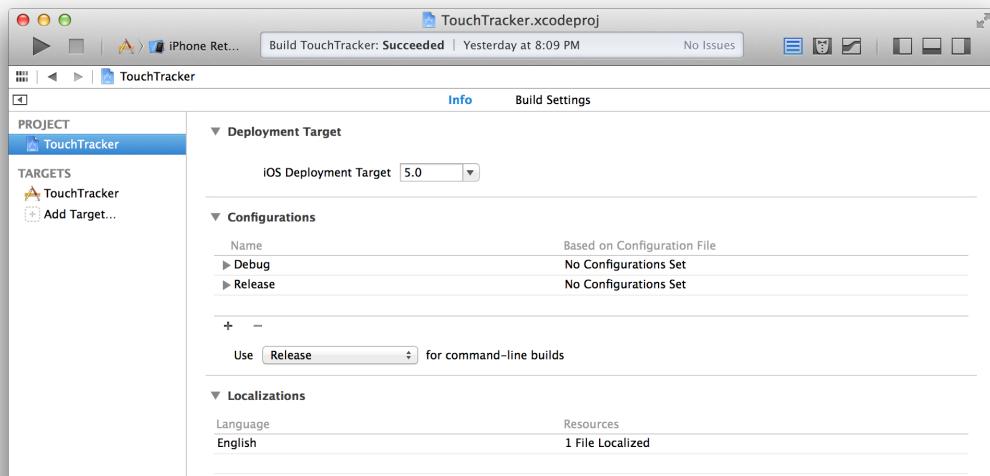
While you are here, search for "static analyzer" in the build settings list. You can set the **Analyze During 'Build'** setting to Yes for Xcode to automatically run the static analyzer every time you build your application. This will slightly increase the amount of time that it takes to build, but is generally a good idea.

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 14.23).

Figure 14.23 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.

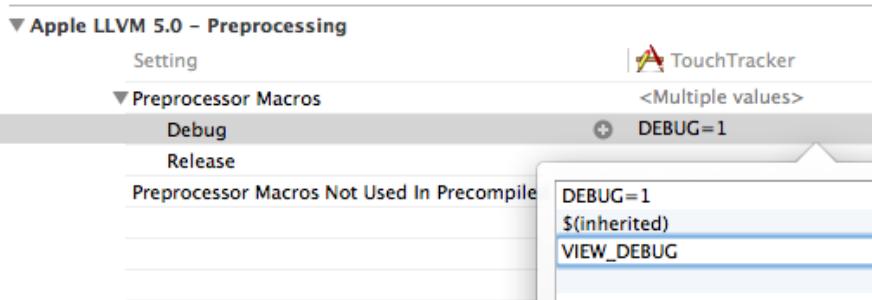
When performing a scheme action, the scheme will use one of these configurations when building its targets. You can specify the build configuration that the scheme uses in the scheme editor in the option for Build Configuration in the Info tab.

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 Preprocessor Macros build setting. Double-click on the value column for the Debug configuration under Preprocessor Macros. In the table that appears, add a new item: VIEW_DEBUG, as shown in Figure 14.24.

Figure 14.24 Changing a build setting



Adding this value to this setting says, “When you are building the TouchTracker target with the debug configuration, a preprocessor macro `VIEW_DEBUG` is defined.”

Let’s add some debugging code to TouchTracker that will only be compiled when the target is built with the debug configuration. `UIView` has a private method `recursiveDescription` that prints out the entire view hierarchy of an application. However, you cannot call this method in an application that you deploy to the App Store, so you will only allow it to be called if `VIEW_DEBUG` is defined.

In `BNRAppDelegate.m`, add the following code to `application:didFinishLaunchingWithOptions:`.

```
[self.window makeKeyAndVisible];
#ifndef VIEW_DEBUG
    NSLog(@"%@", [self.window performSelector:@selector(recursiveDescription)]);
#endif
    return YES;
}
```

This code will send the message `recursiveDescription` to the window. (Notice the use of `performSelector::`. `recursiveDescription` is a private method, so you have to dispatch it in this way.) `recursiveDescription` will print a view’s description, then all of its subviews, and its subviews’ subviews and so on. You can leave this code in for all builds. Because the preprocessor macro will not be defined for a release build, the code will not be compiled when you build for the App Store.

Now build and run the application. Check out the console and you will see the view hierarchy of your application, starting at the window.

This page intentionally left blank

15

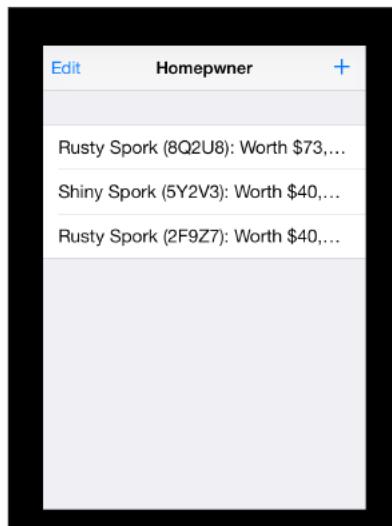
Introduction to Auto Layout

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 15.1 An iPhone application running on a simulated iPad

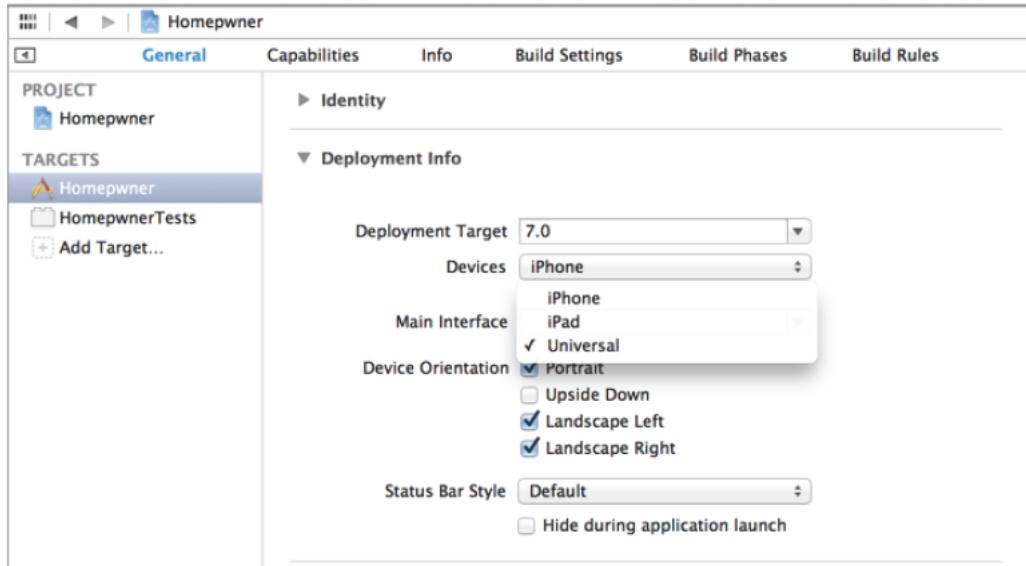


This is not what you want for your iPad users. You want `Homepwner` to run *natively* on the iPad so that it will look like an iPad app. A single application that runs natively on both the iPad and the iPhone is called a *universal application*.

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.

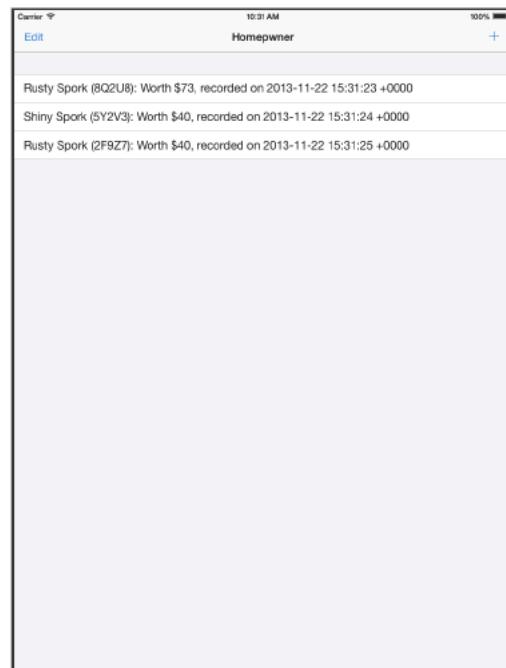
Locate the Deployment Info section and change the Devices pop-up from iPhone to Universal (Figure 15.2).

Figure 15.2 Universalizing Homepwner



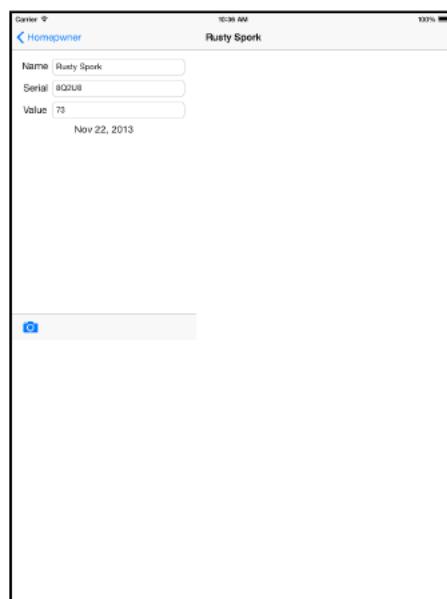
Homepwner is now a universal application. To see the difference, select an iPad simulator from the scheme pop-up menu and then build and run the application. Homepwner looks much better. The table view and its rows are sized appropriately for the iPad screen.

Figure 15.3 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 15.4 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 4, 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 15.1. (Remember that points are used when laying out your interface and map to physical pixels on the device's screen. A retina device has the same size screen in points as a non-retina device, even though it has twice as many pixels.)

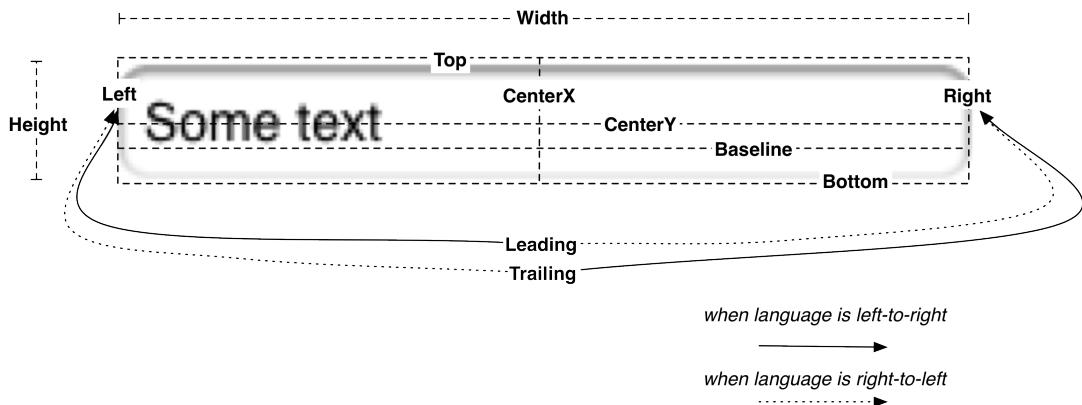
Table 15.1 Device Screen Sizes

| Device | Width x Height (points) |
|------------------------------|-------------------------|
| iPhone/iPod (4S and earlier) | 320 x 480 |
| iPhone/iPod (5 and later) | 320 x 568 |
| 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 15.5).

Figure 15.5 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 (left edge + width = 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 **BNRDetailViewController**'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 15.6).

Figure 15.6 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 `BNRDetailViewController`).
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.

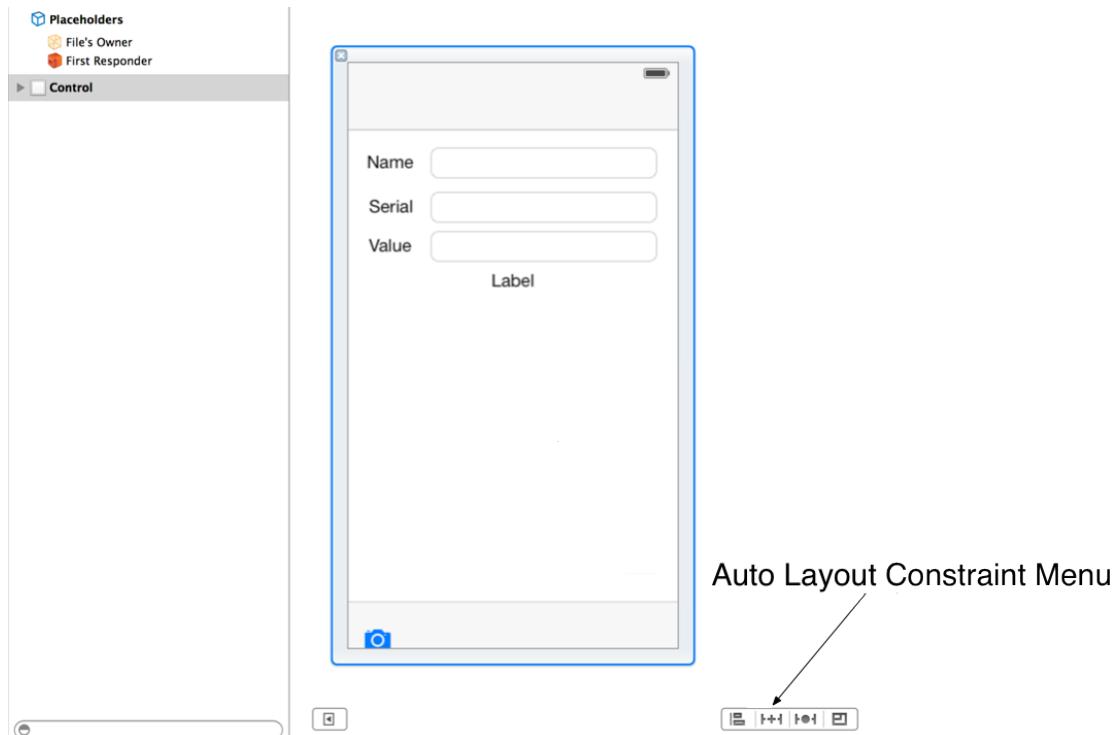
Apple recommends that you add constraints using Interface Builder whenever possible. This is what you will do in this chapter. However, if your views are created and configured programmatically, then you can add constraints in code. In Chapter 16, you will get a chance to practice that approach.

Adding Constraints in Interface Builder

Open `BNRDetailViewController.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 16.

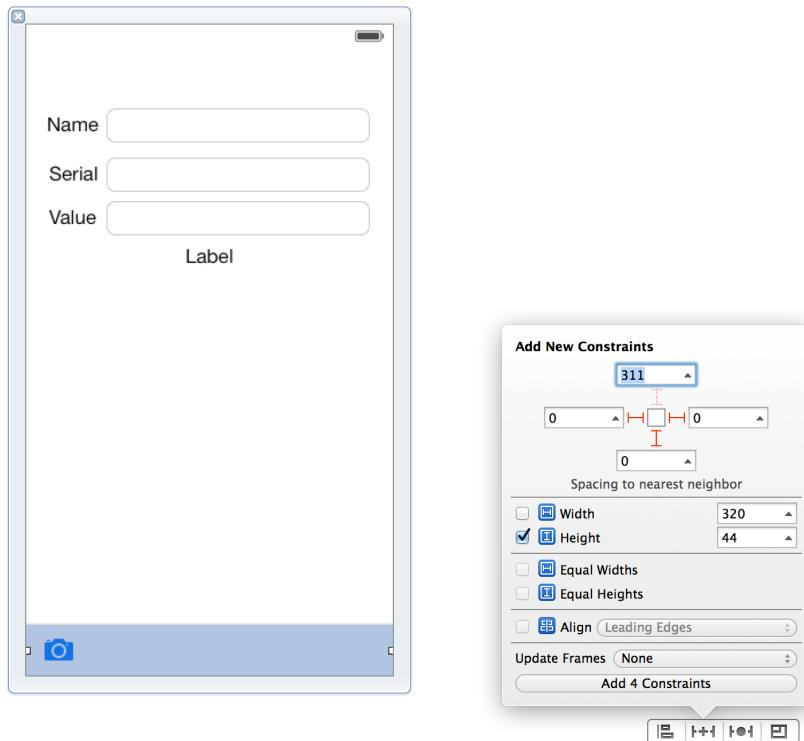
Select the toolbar on the canvas. At the bottom righthand corner of the canvas, find the Auto Layout constraint menu (Figure 15.7).

Figure 15.7 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 15.8).

Figure 15.8 Adding 4 constraints to the toolbar



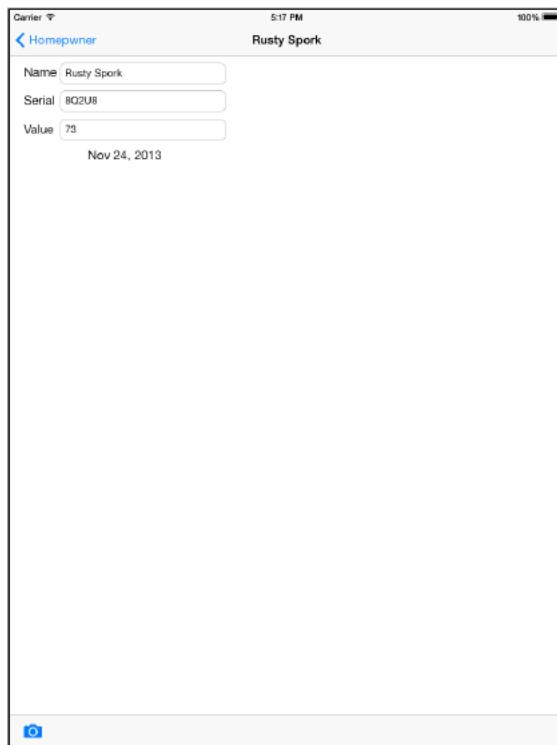
At the top of the Pin menu are four values that describe the toolbar's current spacing from its nearest neighbor on the canvas. For the toolbar, you are only interested in the bottom, left, and right values. They are all 0, meaning that these edges of the toolbar are currently 0 points away from the 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 **BNRDetailViewController**.

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.

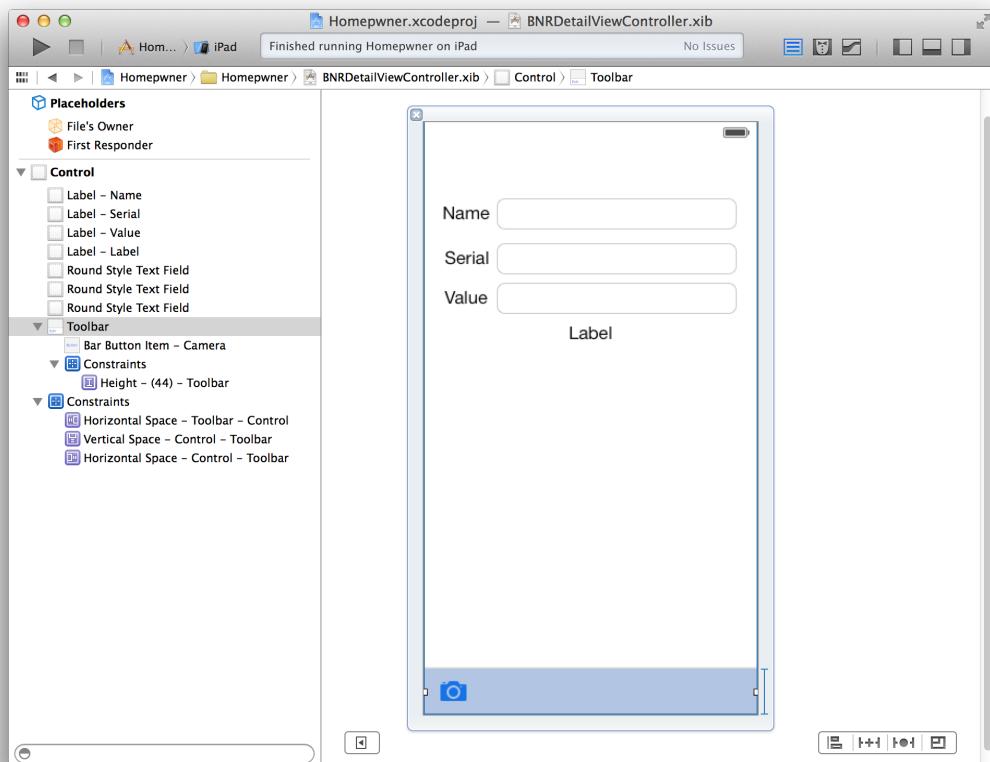
Build and run the application on the iPad simulator. Create an item and select it to navigate to the detail interface. The toolbar will appear at the bottom of the screen and be as wide as the screen (Figure 15.9).

Figure 15.9 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 15.10).

Figure 15.10 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 **BNRDetailViewController**. (Recall that you changed the class of this view object from **UIView** to **UIControl** at the end of Chapter 11 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

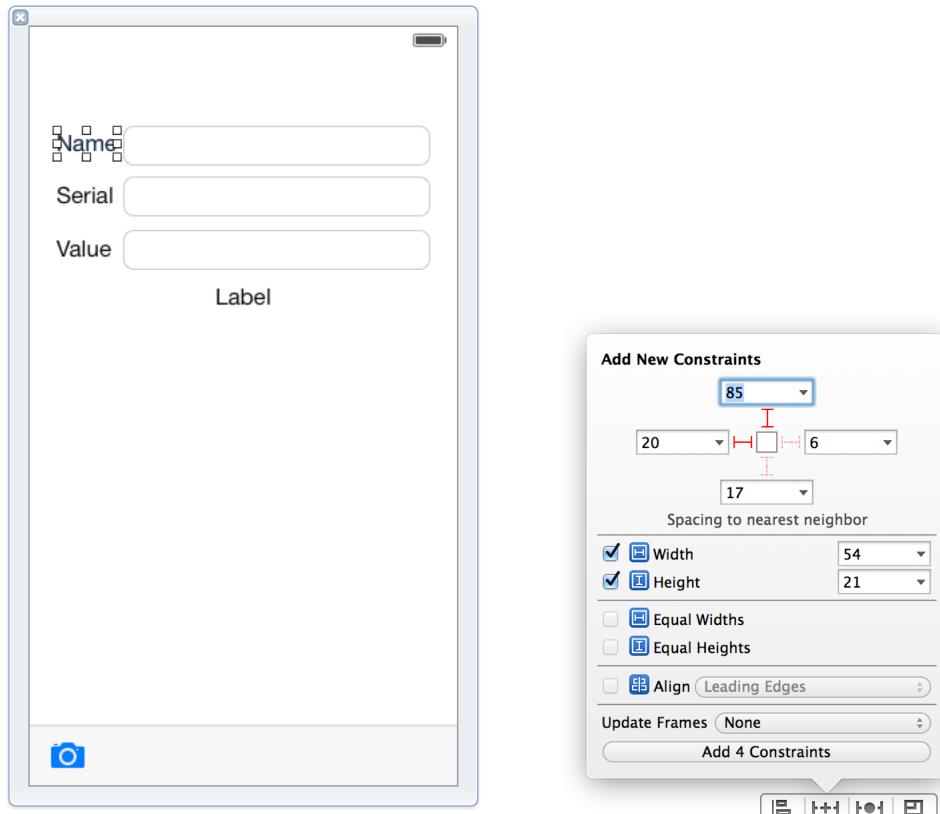
Let's turn now to the Name label. This label's size and position are fine right now when run on the iPad. However, you still need to constrain this view so that it will appear appropriately if it is presented in a different language or font size.

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 `BNRDetailViewController`). 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 15.11. 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 15.11, 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 15.11 Example constraints for the Name label



Click Add 4 Constraints at the bottom of the menu.

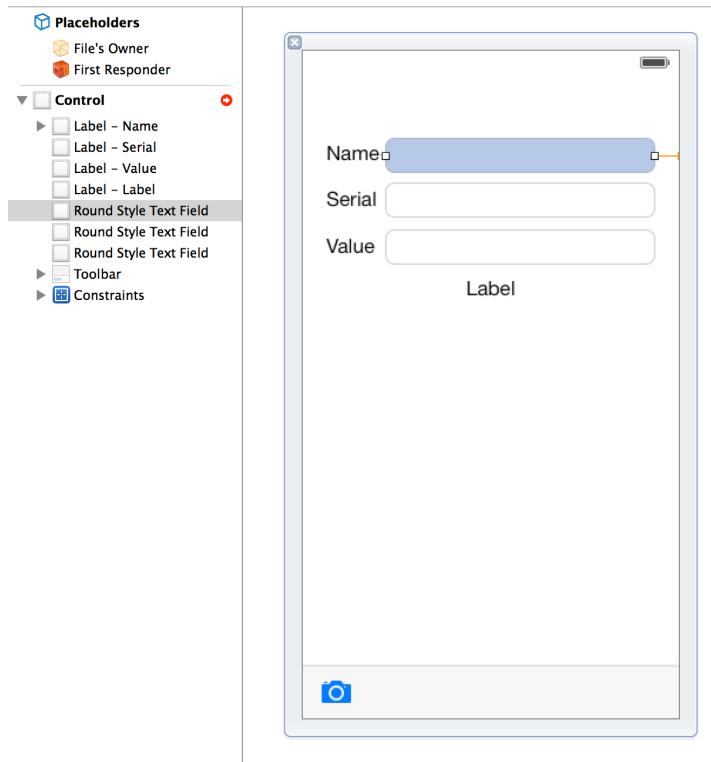
Now consider the text field to the right of the Name label. Regardless of screen size, the text field's width should stretch from its position to the left of the Name label to fill most of the screen.

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 15.12 Insufficient constraints for text field

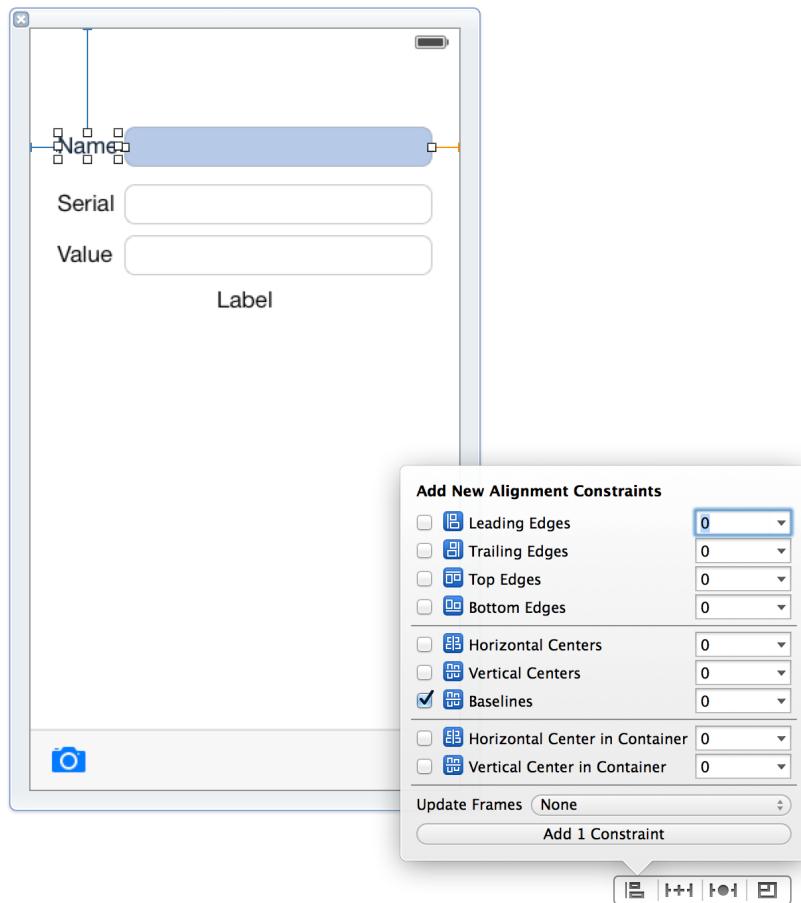


According to Interface Builder, you are missing a constraint. The text field needs its Y (vertical) position constrained. You could fix this problem by opening the Pin menu and selecting the top strut. This would pin the text field some distance from its container.

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 15.13 Aligning baselines of label and text field

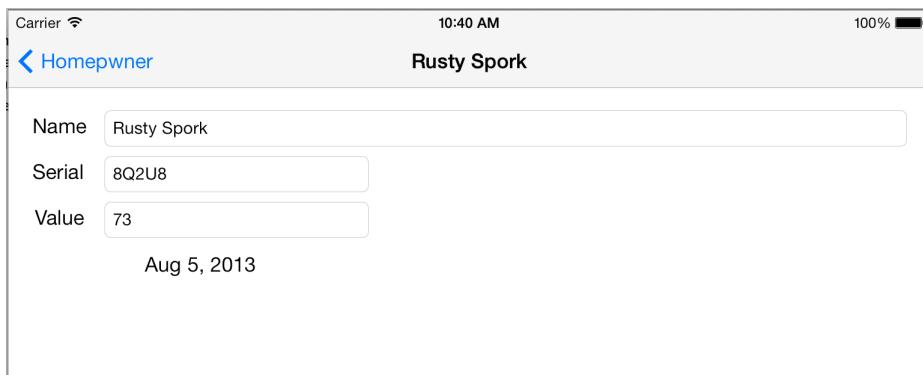


Now the lines representing the constraints are blue again, and the red icon has disappeared; the text field has enough constraints to size and position it unambiguously.

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 15.14).

Figure 15.14 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.

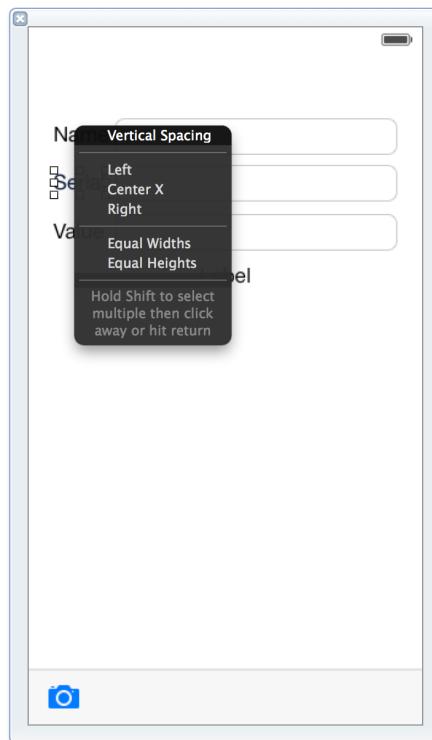
Here are the constraints that you need to add for the `Serial` label:

- 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 15.15).

Figure 15.15 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 edge should be pinned at its current distance from its container

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 text field right to the superview. Select **Trailing Space to Container**.

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 pinned at its current distance from its container

For the date label...

- top edge should be pinned at its current distance from the text field that displays the item's value
- leading and trailing edges should be pinned at their current distances from its container
- height should be fixed at its current value

Build and run the application on iPad. The `BNRDetailViewController` 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 `BNRDetailViewController.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.

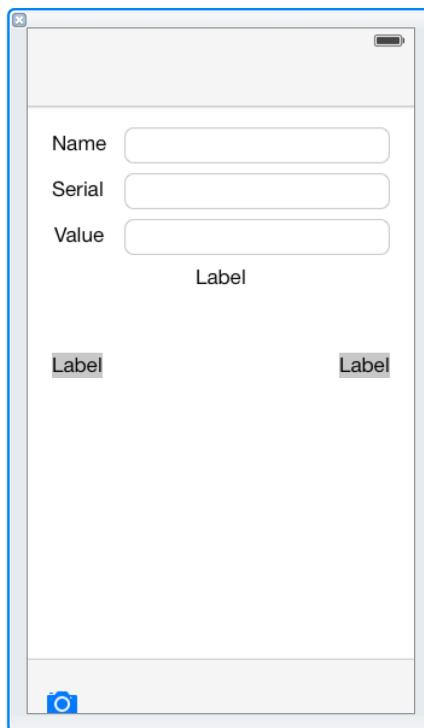
Fortunately, there are several tools you can use to debug Auto Layout constraints. Let's take a look at each of these in turn and see ways to fix them.

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.

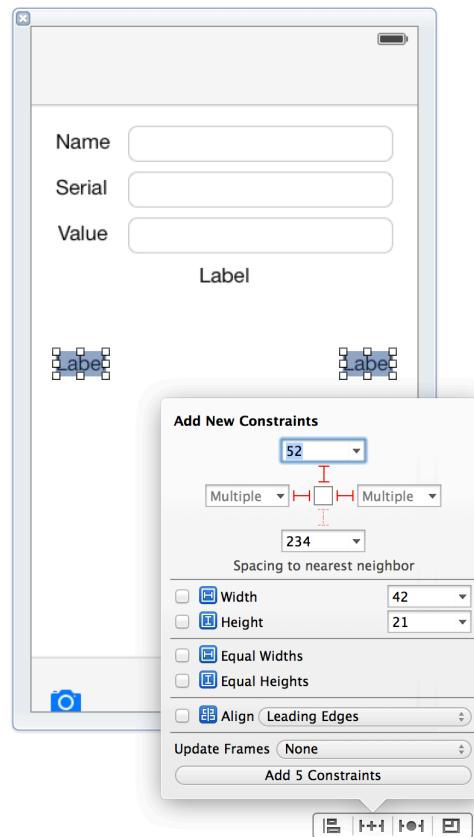
Currently, there are no ambiguous layouts, so let's introduce some. Add two labels to the **UIControl** under the `dateLabel` and position them next to one another. In the attributes inspector, change the background color of the labels to light gray so that you can see their frames. The interface should look like Figure 15.16.

Figure 15.16 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 15.17).

Figure 15.17 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 15.18).

Figure 15.18 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 **BNRDetailViewController.m**. Override the method **viewDidLayoutSubviews** to check if any of its subviews has an ambiguous layout.

```
- (void)viewDidLayoutSubviews
{
    for (UIView *Subview in self.view.subviews) {
        if ([Subview hasAmbiguousLayout])
            NSLog(@"%@", 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 **BNRDetailViewController**. 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 **BNRDetailViewController.m**, edit the **backgroundTapped:** method to send the message **exerciseAmbiguityInLayout** to any ambiguous views.

```
- (IBAction)backgroundTapped:(id)sender
{
    [self.view endEditing:YES];

    for (UIView *Subview in self.view.subviews) {
        if ([Subview hasAmbiguousLayout]) {
            [Subview exerciseAmbiguityInLayout];
        }
    }
}
```

Build and run the application again. Once on the **BNRDetailViewController**, tap the background view anywhere. The width of the two labels will swap.

Figure 15.19 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 **BNRDetailViewController.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 `BNRDetailViewController.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 `BNRDetailViewController.m`, delete `viewDidLayoutSubviews` and delete the code that calls `exerciseAmbiguityInLayout` in `backgroundTapped:`.

```
- (void)viewDidLayoutSubviews
{
    for (UIView *Subview in self.view.subviews) {
        if ([Subview hasAmbiguousLayout])
            NSLog(@"%@", Subview);
    }
}
- (IBAction)backgroundTapped:(id)sender
{
    [self.view endEditing:YES];

    for (UIView *Subview in self.view.subviews) {
        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 `BNRDetailViewController`.

In `BNRDetailViewController.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 `NSAutoresizingMaskLayoutConstraint` 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/UIKit.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.

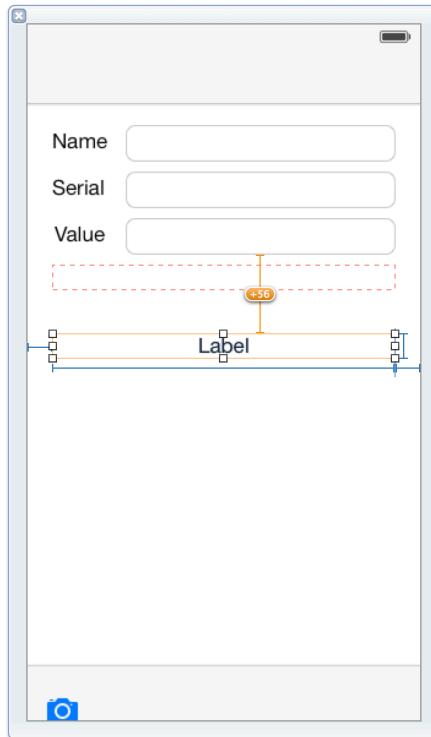
In `BNRDetailViewController.xib`, delete the width constraint that you just added to the label and set its background color back to clear.

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 15.20.

Figure 15.20 A misplaced view

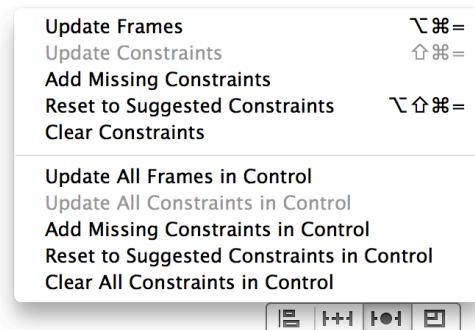


A rectangle with an orange, dashed border will appear where the label used to be. This is the runtime frame; at runtime, the existing constraints will position the label where this rectangle is and not where you just dragged it to.

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 15.21).

Figure 15.21 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 15.4 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 15.22 Universalized Quiz running on the iPad



Decide how the interface should look on the iPad and then add constraints in `BNRQuizViewController.xib` to ensure that it will appear as you want on any device.

For the More Curious: Debugging Using the Auto Layout Trace

In this chapter, to check for an ambiguous layout, you iterated over the subviews of `BNRDetailViewController`'s view, and then asked each subview if it `hasAmbiguousLayout`. This worked well since all of the views you were working with were subviews of the main view. What if your view hierarchy was much more complex? There is another way to find ambiguous layouts using some private methods that Apple has made known.

`UIWindow` has a private instance method named `_autoLayoutTrace`. This will return a string with a graphical representation of that window's entire view hierarchy, and will tag views with an ambiguous layout with `AMBIGUOUS LAYOUT`.

The best way to use this is to place a breakpoint somewhere in your code that will get triggered after the affected view is on screen. Once this breakpoint has been hit, type the following into the debugger, and press Enter.

```
(lldb) po [[UIWindow keyWindow] _autoLayoutTrace]
```

This can be very helpful when your UI does not look as you expect but you are not sure where the problem is originating from.

For the More Curious: Multiple XIB Files

It is possible that you may have a view controller that needs completely different views depending on the type of device that the application is running on. If this is the case, you can create two XIB files – one for each device type.

To get a view controller to load the appropriate XIB file for each device, you simply add a suffix to the filename:

```
BNRDetailViewController~iphone.xib  
BNRDetailViewController~ipad.xib
```

By naming XIB files in this way, a view controller will automatically find and load the right file at runtime.

Note that using distinct XIB files is not an alternative to using Auto Layout. You will still need to add constraints to both files. Auto Layout also enables your interface to respond appropriately to other differences, like in the user's language or preferred font size or in the orientation of the device.

16

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 **BNRDetailViewController**'s interface has been loaded.

Recall that in Chapter 6, 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 **BNRDetailViewController.m**, implement **viewDidLoad** to create an instance of **UIImageView**.

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    UIImageView *iv = [[UIImageView alloc] initWithImage:nil];
    // The contentMode of the image view in the XIB was Aspect Fit:
    iv.contentMode = UIViewContentModeScaleAspectFit;
    // Do not produce a translated constraint for this view
    iv.translatesAutoresizingMaskIntoConstraints = NO;
    // The image view was a subview of the view
    [self.view addSubview:iv];
    // The image view was pointed to by the imageView property
    self.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:

```
+ (NSArray *)constraintsWithVisualFormat:(NSString *)format
                                    options:(NSLayoutFormatOptions)opts
                                      metrics:(NSDictionary *)metrics
                                         views:(NSDictionary *)views
```

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 an **NSDictionary** 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 `BNRDetailViewController.m`, create a dictionary of names for the views at the end of `viewDidLoad`.

```
[self.view addSubview:iv];
self.imageView = iv;

NSDictionary *nameMap = @{@"imageView" : self.imageView,
                         @"dateLabel" : self.dateLabel,
                         @"toolbar" : self.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 `BNRDetailViewController.m`, create the horizontal and vertical constraints for the image view:

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    ...

    NSDictionary *nameMap = @{@"imageView" : self.imageView,
                               @"dateLabel" : self.dateLabel,
                               @"toolbar" : self.toolbar};

    // imageView is 0 pts from superview at left and right edges
    NSArray *horizontalConstraints =
        [NSLayoutConstraint constraintsWithVisualFormat:@"H:|-0-[imageView]-0-"
                                                options:0
                                                metrics:nil
                                                views:nameMap];

    // imageView is 8 pts from dateLabel at its top edge...
    // ... and 8 pts from toolbar at its bottom edge
    NSArray *verticalConstraints =
        [NSLayoutConstraint constraintsWithVisualFormat:
            @"V:[dateLabel]-[imageView]-[toolbar]"
            options:0
            metrics:nil
            views:nameMap];
}
```

Adding Constraints

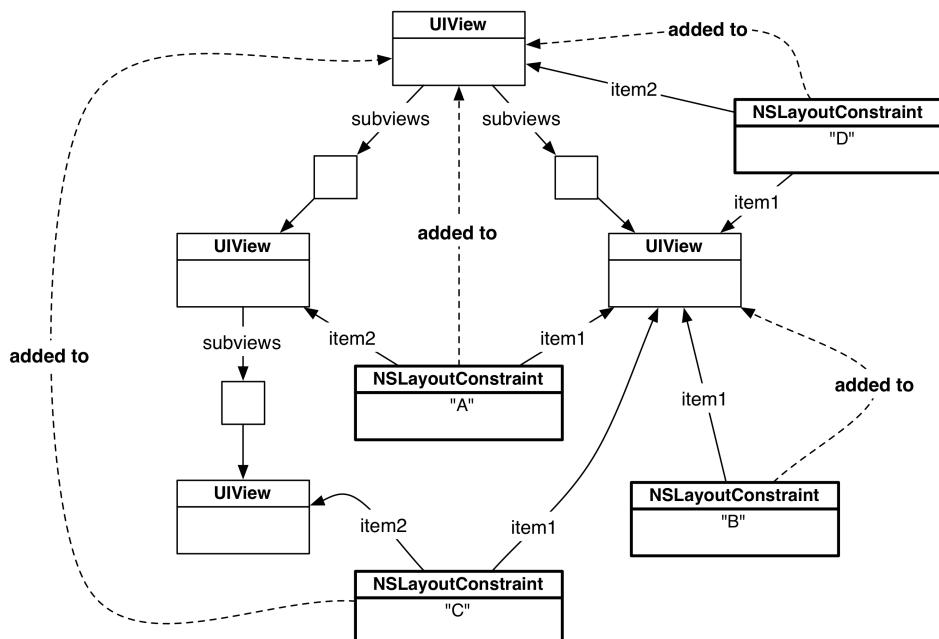
You now have two arrays of `NSLayoutConstraint` objects. However, these constraints will have no effect on the layout until you explicitly add them using the `UIView` method

```
- (void)addConstraints:(NSArray *)constraints
```

Which view should receive the **addConstraints:** message? Usually, the closest common ancestor of the views that are affected by the constraint. Here is a list of rules you can follow to determine which view you should add constraints to:

- If a constraint affects two views that have the same superview (such as the constraint labeled “A” in Figure 16.1), then the constraint should be added to their superview.
- If a constraint affects just one view (the constraint labeled “B”), then the constraint should be added to the view being affected.
- If a constraint affects two views that do not have the same superview but do share a common ancestor much higher up on the view hierarchy (the constraint labeled “C”), then the first common ancestor gets the constraint.
- If a constraint affects a view and its superview (the constraint labeled “D”), then this constraint will be added to the superview.

Figure 16.1 Constraint hierarchy



For the image view’s horizontal constraints, this determination is easy. These constraints affect only the `imageView` and its superview, so you add them to the superview – the view of the **BNRDetailViewController**.

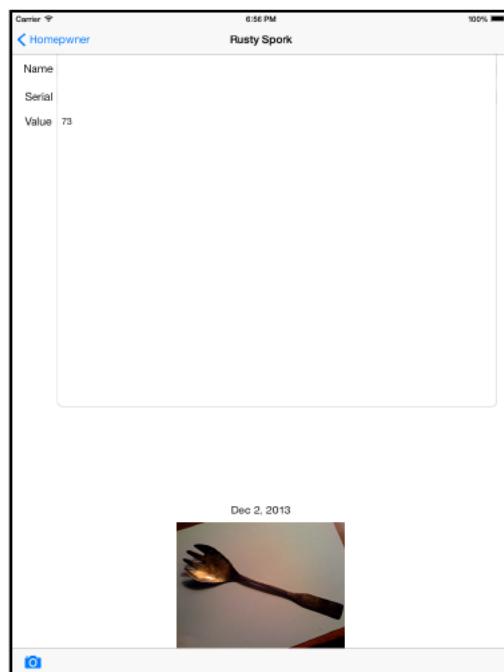
For the vertical constraints, the `imageView`, `dateLabel`, and `toolbar` are the affected views. They all share the same superview (the view of the **BNRDetailViewController**), so you also add these constraints to the superview.

In `BNRDetailViewController.m`, add both sets of constraints to the **BNRDetailViewController**’s view at the end of `viewDidLoad`.

```
...  
  
NSArray *verticalConstraints =  
    [NSLayoutConstraint constraintsWithVisualFormat:  
        @"V:[dateLabel]-[imageView]-[toolbar]"  
        options:0  
        metrics:nil  
        views:nameMap];  
  
[self.view addConstraints:horizontalConstraints];  
[self.view addConstraints:verticalConstraints];  
}
```

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 16.2 Small-size image gives unexpected results



To understand what is going on, 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.

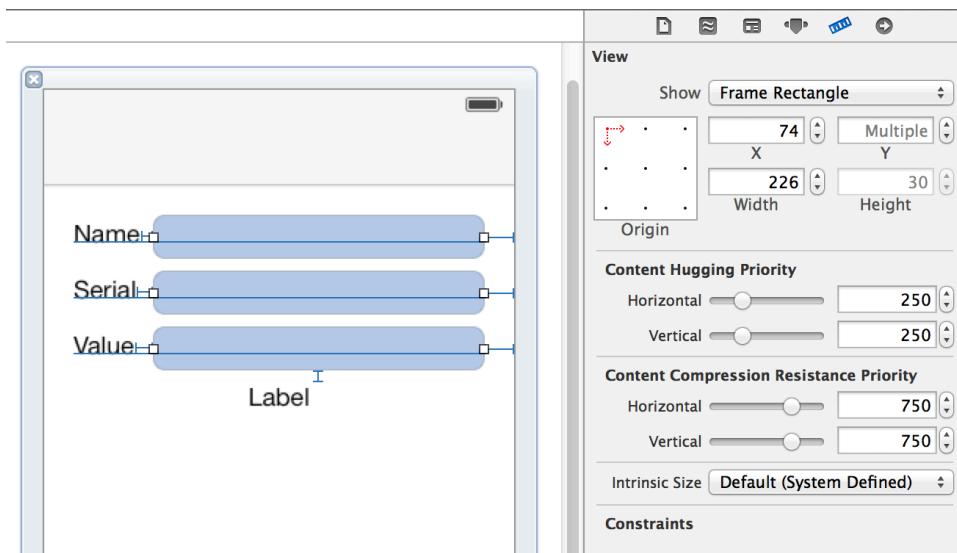
Content compression resistance priority

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 Interface Builder. Reopen `BNRDetailViewController.xib`. Shift-click to select all three text fields in the canvas. Head to the inspector and select the  tab to reveal the *size inspector*. Find the Content Hugging Priority and Content Compression Resistance Priority sections.

Figure 16.3 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.

It would be better if the image view had a smaller vertical content hugging and compression resistance priority than the other subviews. Open `BNRDetailViewController.m` and update `viewDidLoad` to lower these priorities.

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    UIImageView *iv = [[UIImageView alloc] initWithImage:nil];
    // The contentMode of the image view in the XIB was Aspect Fit:
    iv.contentMode = UIViewContentModeScaleAspectFit;

    // Do not produce a translated constraint for this view
    iv.translatesAutoresizingMaskIntoConstraints = NO;

    // The image view was a subview of the view
    [self.view addSubview:iv];

    // The image view was pointed to by the imageView property
    self.imageView = iv;

    // Set the vertical priorities to be less than
    // those of the other subviews
    [self.imageView setContentHuggingPriority:200
                                         forAxis:UILayoutConstraintAxisVertical];
    [self.imageView setContentCompressionResistancePriority:700
                                         forAxis:UILayoutConstraintAxisVertical];

    ...
}
```

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

```
+ (id)constraintWithItem:(id)view1
                     attribute:(NSLayoutAttribute)attr1
                     relatedBy:(NSLayoutRelation)relation
                        toItem:(id)view2
                     attribute:(NSLayoutAttribute)attr2
                    multiplier:(CGFloat)multiplier
                      constant:(CGFloat)c
```

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:

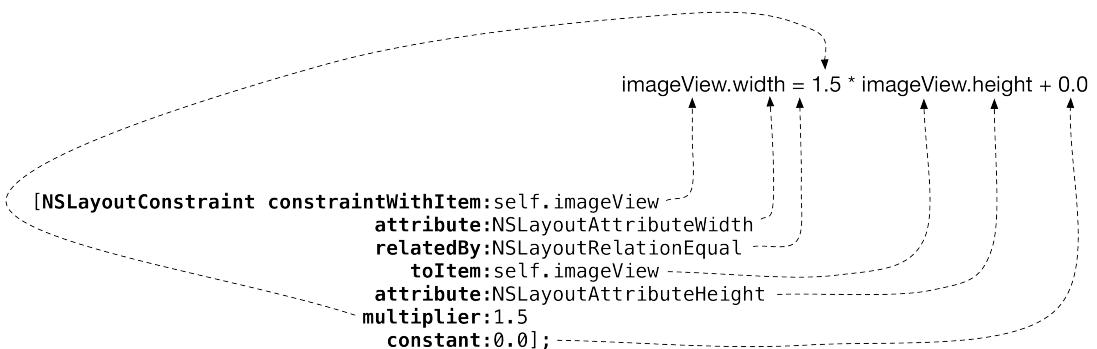
- `NSLayoutAttributeLeft`
- `NSLayoutAttributeRight`
- `NSLayoutAttributeTop`
- `NSLayoutAttributeBottom`
- `NSLayoutAttributeWidth`
- `NSLayoutAttributeHeight`
- `NSLayoutAttributeBaseline`
- `NSLayoutAttributeCenterX`
- `NSLayoutAttributeCenterY`
- `NSLayoutAttributeLeading`
- `NSLayoutAttributeTrailing`

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.)

```
NSLayoutConstraint *aspectConstraint =
    [NSLayoutConstraint constraintWithItem:self.imageView
        attribute:NSLayoutAttributeWidth
        relatedBy:NSLayoutRelationEqual
            toItem:self.imageView
        attribute:NSLayoutAttributeHeight
        multiplier:1.5
        constant:0.0];
```

To understand how this method works, think of this constraint as the equation shown in Figure 16.4.

Figure 16.4 `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.

To add a single constraint to a view, you use the method

- `(void)addConstraint:(NSLayoutConstraint *)constraint`

The same logic applies to decide which view should receive this message. When using this method, the determination is even easier to make because the affected view objects are the first and fourth arguments. In this case, the only affected object is the image view, so you would add `aspectConstraint` to that view:

```
[self.imageView addConstraint:aspectConstraint];
```

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 NSAutoresizingMaskLayoutConstraints 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 = UIViewContentModeScaleAspectFit;

// Do not produce a translated constraint for this view
iv.translatesAutoresizingMaskIntoConstraints = NO;

// The image view was a subview of the view
[self.view addSubview:iv];

```

This page intentionally left blank

17

Autorotation, Popover Controllers, and Modal View Controllers

In the last two chapters, you used Auto Layout to ensure that Homeowner 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 Homeowner's behavior that will tailor the app's behavior to whatever device it is running on.

- On iPads only, allow the interface to rotate when the device is upside down.
- On iPads only, show the image picker in a popover controller when the user presses the camera button.
- On iPads only, present the detail interface modally when the user creates a new item.
- On iPhones only, disable the camera button in the detail interface when the device is in landscape orientation.

Thus, you are going to learn how to test for the type of the device and write device-specific code. You will also learn about rotation, popover controllers, and more about modal view 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.

Chapter 17 Autorotation, Popover Controllers, and Modal View Controllers

The interface orientation, by contrast, is a property of the running application. The following list shows all of the possible interface orientations:

| | |
|---|--|
| <code>UIInterfaceOrientationPortrait</code> | The Home button is below the screen. |
| <code>UIInterfaceOrientationPortraitUpsideDown</code> | The Home button is above the screen. |
| <code>UIInterfaceOrientationLandscapeLeft</code> | The device is on its side and the Home button is to the right of the screen. |
| <code>UIInterfaceOrientationLandscapeRight</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.

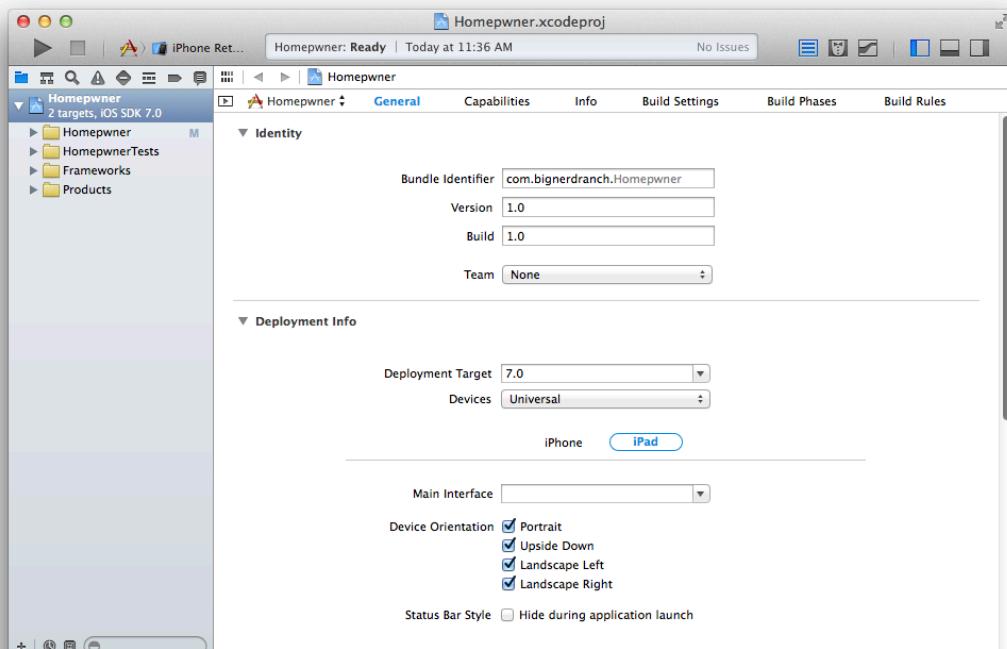
While `Homepwner` is running on the simulator, you can simulate a rotation. Navigate to the **BNRDetailViewController**. From the simulator's Hardware menu, select Rotate Left option. The simulator window will rotate, causing your "device" to rotate its window and contents. Because you have configured 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 `UIInterfaceOrientationPortraitUpsideDown`. 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 for iPad. 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 17.1).

Figure 17.1 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.

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.

In addition to the application choosing which interface orientations are acceptable, the view controller that is occupying the screen also gets a say. (In Homepwner, the **UINavigationController** occupies the screen, except for when the **BNRDetailViewController** is presented modally.) Each view controller implements a method that returns all of the interface orientations it supports. For the interface orientation to change, both the `rootViewController` of the application and the application itself (per the Supported Interface Orientations section of the info property list) must agree that the new orientation is OK.

By default, a view controller running on the iPad will allow all orientations. A view controller on the iPhone application will allow all but the upside-down orientation. If you would like to change this, you must override **supportedInterfaceOrientations** in that view controller. The default implementation of this method looks like this:

Chapter 17 Autorotation,Popover Controllers, and Modal View Controllers

```
- (NSUInteger)supportedInterfaceOrientations
{
    if ([UIDevice currentDevice].userInterfaceIdiom == UIUserInterfaceIdiomPad) {
        return UIInterfaceOrientationMaskAll;
    } else {
        return UIInterfaceOrientationMaskAllButUpsideDown;
    }
}
```

This code checks to see whether the application is running on an iPad or an iPhone. You get the current device and then test its `userInterfaceIdiom` property. The two possible values (as of this writing) are `UIUserInterfaceIdiomPhone` and `UIUserInterfaceIdiomPad`.

If, for some reason, your root view controller wanted to appear in landscape left or landscape right only, it could implement the method as follows:

```
- (NSUInteger)supportedInterfaceOrientations
{
    // On all devices, return left and right
    return UIInterfaceOrientationMaskLandscapeLeft
        | UIInterfaceOrientationMaskLandscapeRight;
}
```

(If the bitwise-OR (`|`) operator is unfamiliar to you, check out the section called “For the More Curious: Bitmasks” at the end of this chapter.)

In many apps, the screen is occupied by a `UINavigationController` or a `UITabViewController`. `UINavigationController` uses the `supportedInterfaceOrientations` inherited from `UIViewController`. If you want the view controller being displayed by the `UINavigationController` to determine the autorotation mask, subclass `UINavigationController` and override it:

```
@implementation MyNavController
- (NSUInteger)supportedInterfaceOrientations
{
    return self.topViewController.supportedInterfaceOrientations;
}
@end
```

`UITabViewController` asks the view controller for each of its tabs for its supported interface orientations and returns the intersection: That is, the `UITabViewController` only supports an orientation if all its tabs support it.

Rotation Notification

There will be times when you want to do something special in a view controller when the device orientation changes. In `Homewner`, one issue is that on an iPhone, the `UIImageView` on the `BNRDetailViewController` becomes too small in landscape orientation. It would make more sense to limit the user to taking and viewing the picture in portrait orientation. To make this happen, you need to hide the image view and disable the camera button when the application is in landscape orientation.

First, you need a pointer to the camera button so that you can send it a message to disable it. Navigate to `BNRDetailViewController.m`. Then, Option-Click on `BNRDetailViewController.xib` to open it in the assistant editor.

Now, Control-drag from the camera button on the toolbar to the class extension area of `BNRDetailViewController.m` to create a weak property outlet named `cameraButton`. This will create and connect a new property:

```
@property (weak, nonatomic) IBOutlet UIBarButtonItem *cameraButton;
```

Now back to your goal: hiding the image view and disabling the camera button only in landscape and only if the device is an iPhone.

When writing code to respond to a change in orientation, you override the `UIViewController` method `willAnimateRotationToInterfaceOrientation:duration:`. The message `willAnimateRotationToInterfaceOrientation:duration:` is sent to a view controller when the interface orientation successfully changes. The new interface orientation value is in the first argument to this method.

In `BNRDetailViewController.m`, create a new method called `prepareViewsForOrientation:` to check for the device and then check the interface orientation. If the device is an iPhone and the new orientation is landscape, hide the image view and disable the button. Call this method when the view first comes on screen and again whenever the orientation changes:

```
- (void)prepareViewsForOrientation:(UIInterfaceOrientation)orientation
{
    // Is it an iPad? No preparation necessary
    if ([UIDevice currentDevice].userInterfaceIdiom == UIUserInterfaceIdiomPad) {
        return;
    }

    // Is it landscape?
    if (UIInterfaceOrientationIsLandscape(orientation)) {
        self.imageView.hidden = YES;
        self.cameraButton.enabled = NO;
    } else {
        self.imageView.hidden = NO;
        self.cameraButton.enabled = YES;
    }
}

- (void)willAnimateRotationToInterfaceOrientation:
    (UIInterfaceOrientation)toInterfaceOrientation
    duration:(NSTimeInterval)duration
{
    [self prepareViewsForOrientation:toInterfaceOrientation];
}

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];

    UIInterfaceOrientation io =
        [[UIApplication sharedApplication] statusBarOrientation];
    [self prepareViewsForOrientation:io];
    ...
}
```

Build and run the application on the iPhone simulator. On the **BNRDetailViewController**, add an image to the **BNRItem** and then rotate to landscape. The image will disappear and the camera button will be grayed out. Upon rotating to portrait again, the image will reappear and the camera button will be enabled again. If you build and run on the iPad, the image view and camera button will always be available and enabled.

When you write code that changes something about a view (like its `frame` or whether it is hidden) in this method, those changes are animated. The `duration` argument tells you how long that animation will take. If you are doing some other work on rotation that does not involve views, or you just do not want the views to animate their changes, you can override the `willRotateToInterfaceOrientation:duration:` method in your view controller. This method gives you the same information at the same time, but your views are not automatically animated.

Additionally, if you want to do something after the rotation is completed, you can override `didRotateFromInterfaceOrientation:` in your view controller. This method's argument is the previous interface orientation before the rotation occurred. You can always ask a view controller for its current orientation by sending it the message `interfaceOrientation`.

You have now written some iPhone-specific code. In the next section, you will add another device-specific feature – showing the image picker in a popover when running on an iPad.

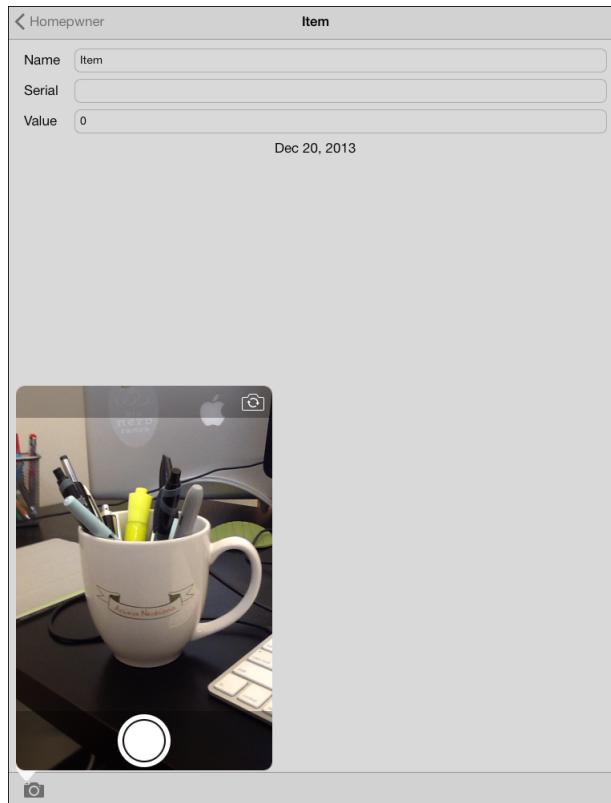
UIPopoverController

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 **UIPopoverController** when the user taps the camera button in the detail interface.

A popover controller 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 **UIPopoverController** when the user taps the camera bar button item in the **BNRDetailViewController**'s view (Figure 17.2).

Figure 17.2 **UIPopoverController**



In the class extension in `BNRDetailViewController.m`, declare that `BNRDetailViewController` conforms to the `UIPopoverControllerDelegate` protocol.

```
@interface BNRDetailViewController ()  
    <UINavigationControllerDelegate, UIImagePickerControllerDelegate,  
    UITextFieldDelegate, UIPopoverControllerDelegate>
```

Additionally, add a property to hold the popover controller.

```
@interface BNRDetailViewController ()  
    <UINavigationControllerDelegate, UIImagePickerControllerDelegate,  
    UITextFieldDelegate, UIPopoverControllerDelegate>  
  
@property (strong, nonatomic) UIPopoverController *imagePickerPopover;  
@property (weak, nonatomic) IBOutlet UITextField *nameField;
```

In `BNRDetailViewController.m`, add the following code to the end of `takePicture:`:

```
imagePickerController.delegate = self;

[self presentViewController:imagePickerController animated:YES completion:nil];

// Place image picker on the screen
// Check for iPad device before instantiating the popover controller
if ([UIDevice currentDevice].userInterfaceIdiom == UIUserInterfaceIdiomPad) {
    // Create a new popover controller that will display the imagePickerController
    self.imagePickerController = [[UIPopoverController alloc]
        initWithContentViewController:imagePickerController];

    self.imagePickerController.delegate = self;

    // Display the popover controller; sender
    // is the camera bar button item
    [self.imagePickerController
        presentPopoverFromBarButtonItem:sender
        permittedArrowDirections:UIPopoverArrowDirectionAny
        animated:YES];
} else {
    [self presentViewController:imagePickerController animated:YES completion:nil];
}
}
```

Notice that you check the device before creating the `UIPopoverController`. It is critical to do this. You can only instantiate popover controllers on the iPad family of devices, and trying to create one on an iPhone will throw an exception.

Build and run the application on the iPad simulator or on an iPad. Navigate to the `BNRDetailViewController` and tap the camera icon. The popover will appear and show the `UIImagePickerController`'s view.

Dismiss the popover by tapping anywhere on the screen. When a popover is dismissed in this way, it sends the message `popoverControllerDidDismissPopover:` to its delegate.

In `BNRDetailViewController.m`, implement `popoverControllerDidDismissPopover:` to set `imagePickerController` to `nil` to destroy the popover. You will create a new popover each time the camera button is tapped.

```
- (void)popoverControllerDidDismissPopover:(UIPopoverController *)popoverController
{
    NSLog(@"User dismissed popover");
    self.imagePickerController = nil;
}
```

The popover should also be dismissed when you select an image from the image picker. In `BNRDetailViewController.m`, at the end of `imagePickerController:didFinishPickingMediaWithInfo:`, dismiss the popover when an image is selected.

```

self.imageView.image = image;
[self dismissViewControllerAnimated:YES completion:nil];

// Do I have a popover?
if (self.imagePopover) {

    // Dismiss it
    [self.imagePopover dismissPopoverAnimated:YES];
    self.imagePopover = nil;
} else {

    // Dismiss the modal image picker
    [self dismissViewControllerAnimated:YES completion:nil];
}
}

```

When you explicitly send the message `dismissPopoverAnimated:` to dismiss the popover controller, it does not send `popoverControllerDidDismissPopover:` to its delegate, so you must set `imagePopover` to `nil` in `dismissPopoverAnimated:` after explicitly dismissing the popover.

There is a small issue with this code. If the `UIPopoverController` is visible and the user taps on the camera button again, the application will crash. This crash occurs because the `UIPopoverController` that is on the screen is destroyed when `imagePopover` is set to point at the new `UIPopoverController` in `takePicture:`. You can ensure that the destroyed `UIPopoverController` is not visible and cannot be tapped by adding the following code to the top of `takePicture:` in `BNRDetailViewController.m`.

```

- (IBAction)takePicture:(id)sender
{
    if ([self.imagePopover isPopoverVisible]) {
        // If the popover is already up, get rid of it
        [self.imagePopover dismissPopoverAnimated:YES];
        self.imagePopover = nil;
        return;
    }

    UIImagePickerController *imagePicker =
        [[UIImagePickerController alloc] init];
}

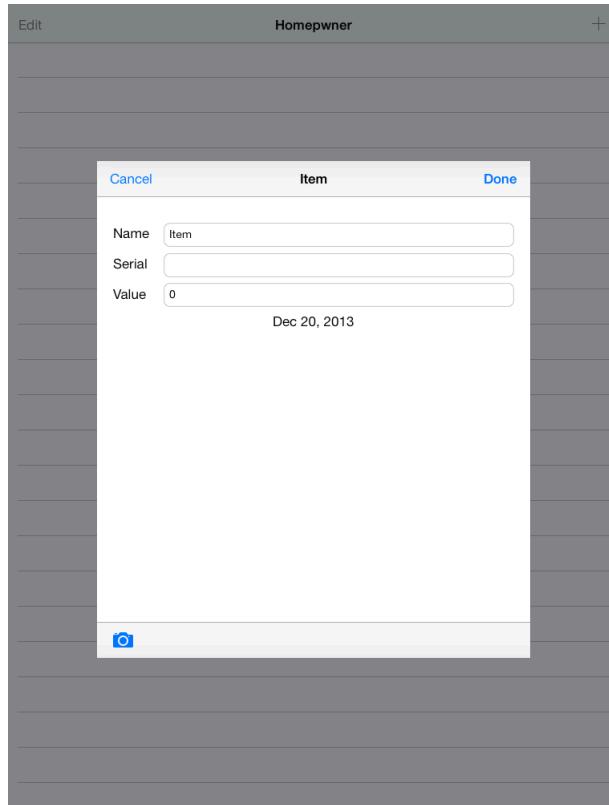
```

Build and run the application. Tap the camera button to show the popover and then tap it again – the popover will disappear.

More Modal View Controllers

In this part of the chapter, you will update `Homepwner` to present the `BNRDetailViewController` modally when the user creates a new `BNRItem` (Figure 17.3). When the user selects an existing `BNRItem`, the `BNRDetailViewController` will be pushed onto the `UINavigationController`'s stack as before.

Figure 17.3 New item



To implement this dual usage of **BNRDetailViewController**, you will give it a new designated initializer, **initWithNewItem:**. This initializer will check whether the instance is being used to create a new **BNRItem** or to show an existing one. Then it will configure the interface accordingly.

In **BNRDetailViewController.h**, declare this initializer.

```
- (instancetype)initWithNewItem:(BOOL)isNew;  
@property (nonatomic, strong) BNRItem *item;
```

If the **BNRDetailViewController** is being used to create a new **BNRItem**, you want it to show a Done button and a Cancel button on its navigation item. Implement this method in **BNRDetailViewController.m**.

```

- (instancetype)initForNewItem:(BOOL)isNew
{
    self = [super initWithNibName:nil bundle:nil];

    if (self) {
        if (isNew) {
            UIBarButtonItem *doneItem = [[UIBarButtonItem alloc]
                initWithBarButtonSystemItem:UIBarButtonItemSystemItemDone
                target:self
                action:@selector(save:)];
            self.navigationItem.rightBarButtonItem = doneItem;

            UIBarButtonItem *cancelItem = [[UIBarButtonItem alloc]
                initWithBarButtonSystemItem:UIBarButtonItemSystemItemCancel
                target:self
                action:@selector(cancel:)];
            self.navigationItem.leftBarButtonItem = cancelItem;
        }
    }

    return self;
}

```

In the past, when you have changed the designated initializer of a class from its superclass's designated initializer, you have overridden the superclass's initializer to call the new one. In this case, you are just going to make it illegal to use the superclass's designated initializer by throwing an exception when anyone calls it.

In `BNRDetailViewController.m`, override `UIViewController`'s designated initializer.

```

- (instancetype)initWithNibName:(NSString *)NibNameOrNil
                           bundle:(NSBundle *)NibNameOrNil
{
    @throw [NSError exceptionWithName:@"Wrong initializer"
                                reason:@"Use initForNewItem:"
                                userInfo:nil];
    return nil;
}

```

This code creates an instance of `NSError` with a name and a reason and then throws the exception. This halts the application and shows the exception in the console.

To confirm that this exception will be thrown, let's return to where this `initWithNibName:bundle:` method is currently called – the `tableView:didSelectRowAtIndexPath:` method of `BNRItemsViewController`. In this method, `BNRItemsViewController` creates an instance of `BNRDetailViewController` and sends it the message `init`, which eventually calls `initWithNibName:bundle:`. Therefore, selecting a row in the table view will result in the “Wrong initializer” exception being thrown.

Build and run the application. (You will get warnings that `save:` and `cancel:` are not implemented. Ignore them for now.) Tap a row. Your application will halt, and you will see an exception in the console. Notice that the name and the reason are part of the console message.

You do not want to see this exception again, so in `BNRItemsViewController.m`, update `tableView:didSelectRowAtIndexPath:` to use the new initializer.

```
- (void)tableView:(UITableView *)tableView  
didSelectRowAtIndexPath:(NSIndexPath *)indexPath  
{  
    BNRDetailViewController *detailViewController =  
        [[BNRDetailViewController alloc] init];  
  
    BNRDetailViewController *detailViewController =  
        [[BNRDetailViewController alloc] initForNewItem:NO];  
  
    NSArray *items = [[BNRItemStore sharedStore] allItems];
```

Build and run the application again. Nothing new and exciting will happen, but your application will no longer crash when you select a row in the table.

Now that you have your new initializer in place, let's change what happens when the user adds a new item.

In `BNRItemsViewController.m`, edit the `addNewItem:` method to create an instance of `BNRDetailViewController` in a `UINavigationController` and present the navigation controller modally.

```
- (IBAction)addNewItem:(id)sender  
{  
    BNRItem *newItem = [[BNRItemStore sharedStore] createItem];  
    NSInteger lastRow = [[[BNRItemStore sharedStore] allItems] indexOfObject:newItem];  
  
    NSIndexPath *indexPath = [NSIndexPath indexPathForRow:lastRow inSection:0];  
  
    [self.tableView insertRowsAtIndexPaths:@[indexPath]  
        withRowAnimation:UITableViewRowAnimationTop];  
  
    BNRDetailViewController *detailViewController =  
        [[BNRDetailViewController alloc] initForNewItem:YES];  
  
    detailViewController.item = newItem;  
  
    UINavigationController *navController = [[UINavigationController alloc]  
        initWithRootViewController:detailViewController];  
  
    [self presentViewController:navController animated:YES completion:nil];  
}
```

Build and run the application and tap the New button to create a new item. An instance of `BNRDetailViewController` 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 `BNRDetailViewController`

presented it, and when the image picker told the **BNRDetailViewController** it was done, the **BNRDetailViewController** dismissed it.

Now, you have a slightly different situation. When a new item is created, the **BNRItemsViewController** presents the **BNRDetailViewController** modally. The **BNRDetailViewController** 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 **BNRDetailViewController**, but it is the responsibility of the **BNRItemsViewController** to do the dismissing. The **BNRDetailViewController** needs a way to tell the view controller that presented it, “Hey, I’m done, you can dismiss me now.”

Fortunately, every **UIViewController** has a `presentingViewController` property that points to the view controller that presented it. The **BNRDetailViewController** will grab a pointer to its `presentingViewController` and send it the message `dismissViewControllerAnimated:completion:`.

In `BNRDetailViewController.m`, implement the action method for the Done button.

```
- (void)save:(id)sender
{
    [self.presentingViewController dismissViewControllerAnimated:YES
                                                    completion:nil];
}
```

The Cancel button has a little bit more going on. When the user taps the button on the **BNRItemsViewController** to add a new item to the list, a new instance of **BNRItem** is created and added to the store, and then the **BNRDetailViewController** slides up to edit this new item. If the user cancels the item’s creation, then that **BNRItem** needs to be removed from the store.

At the top of `BNRDetailViewController.m`, import the header for **BNRItemStore**.

```
#import "BNRDetailViewController.h"
#import "BNRItem.h"
#import "BNRImageStore.h"
#import "BNRItemStore.h"

@implementation BNRDetailViewController
```

Now implement the action method for the Cancel button in `BNRDetailViewController.m`.

```
- (void)cancel:(id)sender
{
    // If the user cancelled, then remove the BNRItem from the store
    [[BNRItemStore sharedStore] removeItem:self.item];

    [self.presentingViewController dismissViewControllerAnimated:YES
                                                    completion:nil];
}
```

Build and run the application. Create a new item and tap the Cancel button. The instance of **BNRDetailViewController** 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 **BNRDetailViewController** will slide off the screen, and your new **BNRItem** will appear in the table view.

There is one final note to make. We said that the **BNRItemsViewController** presents the **BNRDetailViewController** modally. This is true in spirit, but the actual relationships are more complicated than that.

The **BNRDetailViewController**'s `presentingViewController` is really the **UINavigationController** that has the **BNRItemsViewController** on its stack. You can tell this is the case because when the **BNRDetailViewController** is presented modally, it covers up the navigation bar. If the **BNRItemsViewController** was handling the modal presentation, then the **BNRDetailViewController**'s view would fit within the view of the **BNRItemsViewController**, and the navigation bar would not be obscured.

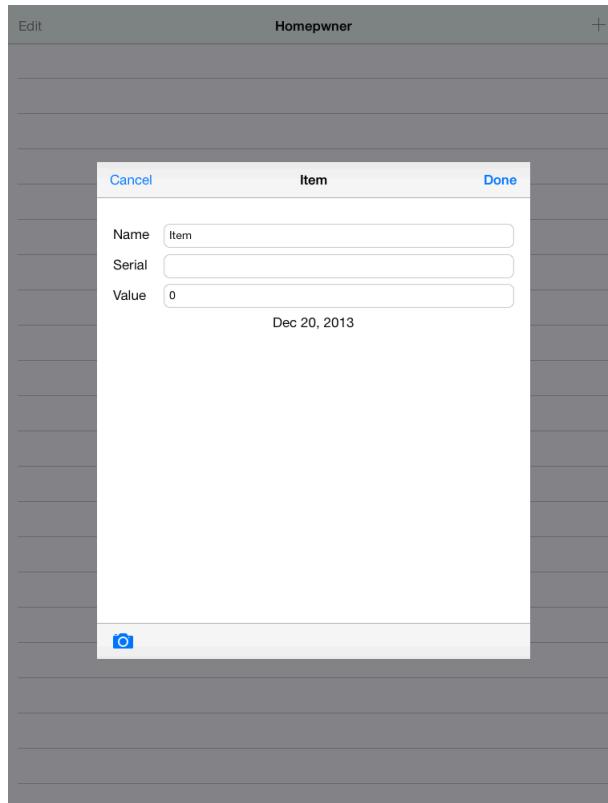
For the purposes of presenting and dismissing modal view controllers, this does not matter; the modal view controller does not care who its `presentingViewController` is as long as it can send it a message and get dismissed. We will address the more complicated truths about view controller relationships at the end of this chapter.

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 – `UIModalPresentationFormSheet` or `UIModalPresentationPageSheet`.

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 17.4).

Figure 17.4 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 `BNRItemsViewController.m`, modify the `addNewItem:` method to change the presentation style of the `UINavigationController` that is being presented.

```
UINavigationController *navController = [[UINavigationController alloc]
                                         initWithRootViewController:detailViewController];
navController.modalPresentationStyle = UIModalPresentationFormSheet;
[self presentViewController:navController animated:YES completion:nil];
```

Notice that you change the presentation style of the `UINavigationController`, not the `BNRDetailViewController`, 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 `BNRItem` 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 `BNRItemsViewController` to disappear. When the modal view controller was dismissed, the `BNRItemsViewController` was sent the messages `viewWillAppear:` and `viewDidAppear:` and took this opportunity to reload its table to catch any updates to the `BNRItemStore`.

With the new presentation style, the `BNRItemsViewController`'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 `BNRItemsViewController` to reload its table view is simple. It looks like this:

```
[self.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.

Completion blocks

In both `dismissViewControllerAnimated:completion:` and `presentViewController:animated:completion:`, you have been passing `nil` as the last argument. Take a look at the type of that argument in the declaration for `dismissViewControllerAnimated:completion:`.

```
- (void)dismissViewControllerAnimated:(BOOL)flag
                           completion:(void (^)(void))completion;
```

Looks strange, huh? This method expects a *block* as an argument, and passing a block here is the solution to your problem. So we need to talk about blocks. However, the concepts and syntax of blocks

can take a while to get used to, so we are just going to introduce them briefly. We will return to blocks as we progress through the book and address different features of them as needed.

A block is a chunk of code to be executed at a later time. You can put the code to reload the table view into a block and pass it to `dismissViewControllerAnimated:completion:`. Then that code will be executed right after the modal view controller is dismissed.

In `BNRDetailViewController.h`, add a new property for a pointer to a block.

```
@property (nonatomic, copy) void (^dismissBlock)(void);
```

This says `BNRDetailViewController` has a property named `dismissBlock` that points to a block. Like a C function, a block has a return value and a list of arguments. These function-like characteristics are included in the declaration of a block. This particular block returns `void` and takes no arguments.

You will not create the block object in `BNRDetailViewController`, though. You have to create it in `BNRItemsViewController` because the `BNRItemsViewController` is the only object that knows about its `tableView`.

In `BNRItemsViewController.m`, create a block that reloads the `BNRItemsViewController`'s table and pass the block to the `BNRDetailViewController`. Do this in the `addNewItem:` method in `BNRItemsViewController.m`.

```
- (IBAction)addItem:(id)sender
{
    // Create a new BNRItem and add it to the store
    BNRItem *newItem = [[BNRItemStore sharedStore] createItem];

    BNRDetailViewController *detailViewController =
        [[BNRDetailViewController alloc] initForNewItem:YES];

    detailViewController.item = newItem;

    detailViewController.dismissBlock = ^{
        [self.tableView reloadData];
    };

    UINavigationController *navController = [[UINavigationController alloc]
                                             initWithRootViewController:detailViewController];
```

Now when the user taps a button to add a new item, a block that reloads the `BNRItemsViewController`'s table is created and set as the `dismissBlock` of the `BNRDetailViewController`. The `BNRDetailViewController` will hold on to this block until the `BNRDetailViewController` needs to be dismissed.

At that point, the `BNRDetailViewController` will pass this block to `dismissViewControllerAnimated:completion:`.

In `BNRDetailViewController.m`, modify the implementations of `save:` and `cancel:` to send the message `dismissViewControllerAnimated:completion:` with `dismissBlock` as an argument.

```

- (IBAction)save:(id)sender
{
    [self.presentingViewController dismissViewControllerAnimated:YES
                                                       completion:nil];
    [self.presentingViewController dismissViewControllerAnimated:YES
                                                       completion:self.dismissBlock];
}

- (IBAction)cancel:(id)sender
{
    [[BNRItemStore sharedStore] removeItem:self.item];

    [self.presentingViewController dismissViewControllerAnimated:YES
                                                       completion:nil];
    [self.presentingViewController dismissViewControllerAnimated:YES
                                                       completion:self.dismissBlock];
}

```

Build and run the application. Tap the button to create a new item and then tap Done. The new **BNRItem** will appear in the table.

Once again, do not worry if the syntax or the general idea of blocks does not make sense at this point. Hold on until Chapter 19, and we will go into more detail there.

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>UIModalTransitionStyleCoverVertical</code> | slides up from the bottom |
| <code>UIModalTransitionStyleCrossDissolve</code> | fades in |
| <code>UIModalTransitionStyleFlipHorizontal</code> | flips in with a 3D effect |
| <code>UIModalTransitionStylePartialCurl</code> | presenting view controller is peeled up revealing the modal view controller |

Thread-Safe Singletons

Thus far, we have only talked about *single-threaded* apps. A single-threaded app only uses one core and is executing only one function at a time. *Multithreaded* apps can execute multiple functions simultaneously on different cores.

In Chapter 11, you created a singleton like this:

```
+ (instancetype)sharedStore
{
    static BNRIImageStore *sharedStore = nil;

    if (!sharedStore) {
        sharedStore = [[self alloc] initPrivate];
    }
    return sharedStore;
}

// No one should call init
- (instancetype)init
{
    @throw [NSError exceptionWithName:@"Singleton"
                    reason:@"Use +[BNRIImageStore sharedStore]"
                    userInfo:nil];
    return nil;
}

- (instancetype)initPrivate
{
    self = [super init];
    if (self) {
        _dictionary = [[NSMutableDictionary alloc] init];
    }
    return self;
}
```

This singleton technique is sufficient in a single-threaded app. However, if your app is multithreaded, you could end up creating two instances of **BNRIImageStore**. Or, you might return an instance for use before it gets properly initialized.

You can create a singleton that is thread-safe by using the function **dispatch_once** to ensure that code is run exactly once.

Open **BNRIImageStore.m** and alter your **sharedStore** method to make **BNRIImageStore** a thread-safe singleton:

```
+ (instancetype)sharedStore
{
    static BNRIImageStore *sharedStore = nil;
    if (!sharedStore) {
        sharedStore = [[self alloc] initPrivate];
    }

    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        sharedStore = [[self alloc] initPrivate];
    });
    return sharedStore;
}
```

Build and run it. You should see no change in behavior, but your **sharedStore** method is now thread-safe.

Bronze Challenge: Another Thread-Safe Singleton

Update the `BNRItemStore` class singleton to also use `dispatch_once()`.

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 `UIPopoverController`.)

For the More Curious: Bitmasks

Earlier in this chapter, you saw the method `supportedInterfaceOrientations` that returned all of the acceptable interface orientations for a view controller. While the return value for this method was a single `int`, this `int` was somehow capable of indicating every combination of the four possible interface orientation values. This is possible because of something called a *bitmask*.

To understand the need for a bitmask, consider an alternative solution for a view controller to advertise which interface orientations it supports. A naive approach would be to give each view controller four properties like this:

```
@property (nonatomic, assign) BOOL canRotateToLandscapeLeft;
@property (nonatomic, assign) BOOL canRotateToLandscapeRight;
@property (nonatomic, assign) BOOL canRotateToPortrait;
@property (nonatomic, assign) BOOL canRotateToPortraitUpsideDown;
```

With this approach, each time the device rotated and the root view controller was checked to see if the interface orientation should comply, the appropriate property would be checked. A bitmask exists to minimize the amount of code and storage needed to represent a series of on and off switches in a single integer variable.

A bitmask is possible because a computer stores values in *binary*. Binary numbers are a string of 1s and 0s. Here are a few examples of numbers in base 10 (decimal; the way we think about numbers) and base 2 (binary; the way a computer thinks about numbers):

| |
|------------------------|
| $1_{10} = 00000001_2$ |
| $2_{10} = 00000010_2$ |
| $16_{10} = 00010000_2$ |
| $27_{10} = 00011011_2$ |
| $34_{10} = 00100010_2$ |

When talking about binary numbers, we call each digit a *bit*. You can think of each bit as an on-off switch, where 1 is “on” and 0 is “off.” When thinking in these terms, we can use an `int` (which has space for at least 32 bits) as a set of on-off switches. Each position in the number represents one switch – a value of 1 means true, 0 means false. Essentially, we are shoving a ton of `BOOLs` into a single value.

Notice that in the examples above, numbers like 1, 2 and 16 – which are powers of two – have all zeroes and a single one. Numbers that are not a power of two, like 27 and 34, have multiple ones in their binary representation. Therefore, we can use numbers that are powers of two to represent a single switch in a bitmask. Each one of these switches are known as a *mask*.

There exists a mask for each possible interface orientation:

```
UIInterfaceOrientationMaskPortrait = 210 = 000000102
UIInterfaceOrientationMaskPortraitUpsideDown = 410 = 000001002
UIInterfaceOrientationMaskLandscapeRight = 810 = 000010002
UIInterfaceOrientationMaskLandscapeLeft = 1610 = 000100002
```

We can turn on a switch in a bitmask using the bitwise-OR operation. This operation takes two numbers and produces a result where a bit is set to 1 if either of the original numbers had a 1 in the same position. When you bitwise-OR a number with 2^n , it flips on the switch at the n th position. For example, if you bitwise-OR 1 and 16, you get the following:

```
00000010 ( 210, UIInterfaceOrientationMaskPortrait)
| 00010000 (1610, UIInterfaceOrientationMaskLandscapeLeft)
-----
00010010 (1810, both UIInterfaceOrientationMaskPortrait
            and UIInterfaceOrientationMaskLandscapeLeft)
```

The complement to the bitwise-OR operator is the bitwise-AND (&) operator. When you bitwise-AND two numbers, the result is a number that has a 1 in each bit where there is a 1 in the same position as *both* of the original numbers.

```
00010010 (1810, Portrait and Landscape Left)
& 00010000 (1610, Landscape Left)
-----
00010000 (1610, YES)

00010010 (1810, Portrait and Landscape Left)
& 00000100 (410, Upside Down)
-----
00000000 (010, NO)
```

Since any non-zero number means YES (and zero is NO), we use the bitwise-AND operator to check whether a switch is on or not. Thus, when a view controller's **supportedInterfaceOrientations** mask is checked, the code looks like this:

```
if ([viewController supportedInterfaceOrientations]
    & UIInterfaceOrientationMaskLandscapeLeft)
{
    // Allow interface orientation to change to landscape left
}
```

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.

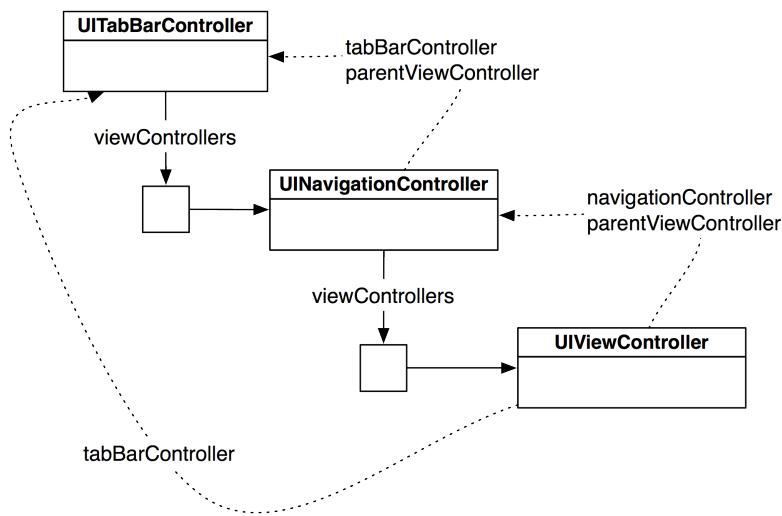
Parent-child relationships

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 22). 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 17.5). 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 17.5 A view controller family



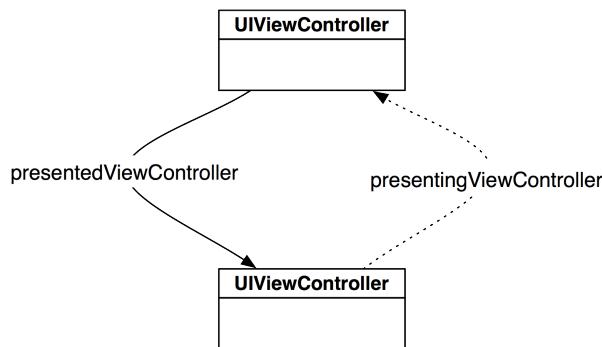
Every **UIViewController** has a `parentViewController` property. This property holds the closest view controller ancestor in the family. Thus, it could return a **UINavigationController**, **UITabBarController**, or a **UISplitViewController** depending on the makeup of the family tree.

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 17.6 Presenting-presenter relationship



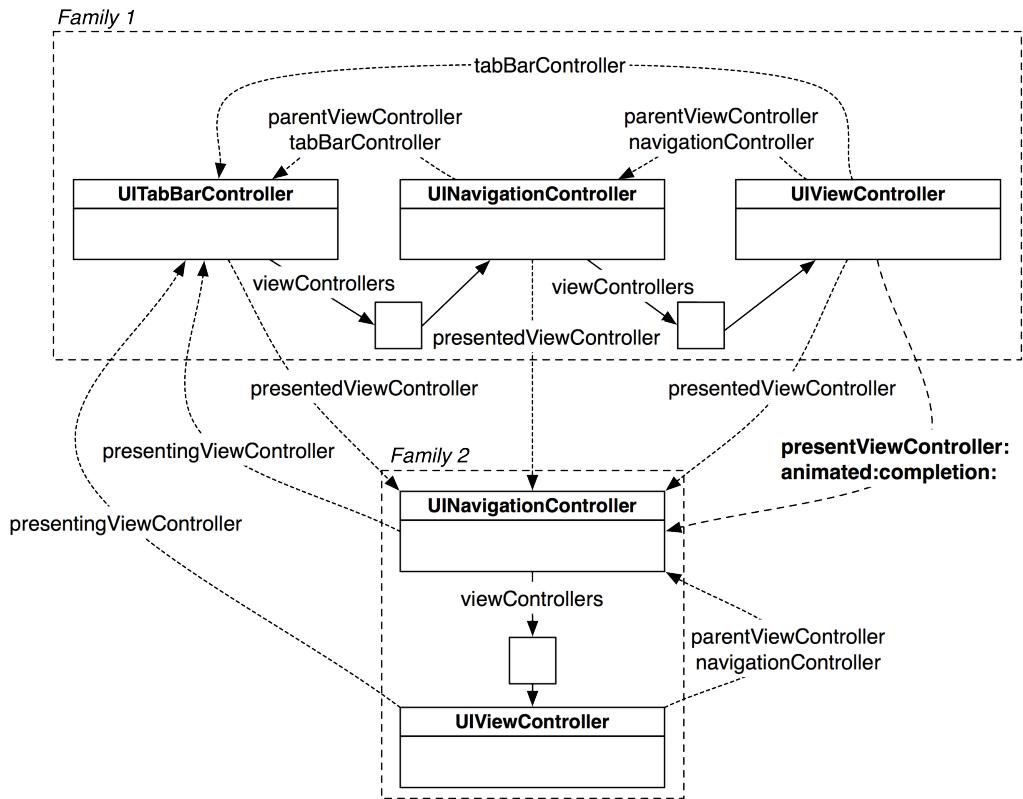
There are two built-in properties for managing the relationship between presenter and presentee. A modally-presented view controller's `presentingViewController` will point back to the view controller that presented it, while the presenter will keep a pointer to the presentee in its `presentedViewController` property (Figure 17.6).

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 17.7. There are two families, each with multiple view controllers. This diagram shows the values of the view controller relationship properties.

Figure 17.7 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 17.7, 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.

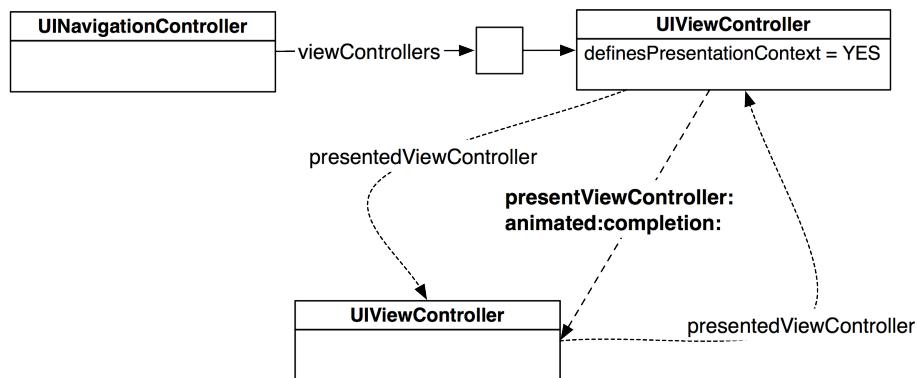
This behavior explains why the **BNRDetailViewController** obscures the **UINavigationBar** when presented modally but not when presented normally in the **UINavigationController**'s stack. Even though the **BNRItemsViewController** is told to do the modal presenting, its oldest ancestor, the **UINavigationController**, actually carries out the task. The **BNRDetailViewController** 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.

You can actually override this oldest-ancestor behavior (but only on the iPad). By doing so, you can specify where the views of the presented view controller family appear on the screen. For example, you could present the **BNRDetailViewController** and its **navigationController** so that it only obscures the **UITableView** but not the **UINavigationBar**.

Every **UIViewController** has a **definesPresentationContext** property for this purpose. By default, this property is NO, which means the view controller will always pass presentation off to its next ancestor, until there are no more ancestors left. Setting this property to YES interrupts the search for the oldest ancestor, allowing a view controller to present the modal view controller in its own view (Figure 17.8). Additionally, you must set the **modalPresentationStyle** for the presented view controller to **UIModalPresentationCurrentContext**.

Figure 17.8 Presentation context



You can test this out by changing the code in **BNRItemsViewController.m**'s **addNewItem:** method.

```
UINavigationController *navController = [[UINavigationController alloc]
                                         initWithRootViewController:detailViewController];

navController.modalPresentationStyle = UIModalPresentationFormSheet;
navController.modalPresentationStyle = UIModalPresentationCurrentContext;
self.definesPresentationContext = YES;

navController.modalTransitionStyle = UIModalTransitionStyleFlipHorizontal;
[self presentViewController:navController animated:YES completion:nil];
}
```

After building and running on the iPad, tap the + icon. Notice that the **BNRDetailViewController** does not obscure the **UINavigationBar**. Make sure you undo this code before moving on to the next chapter.

18

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 **BNRItem**. Homepwner would actually be a useful application if instances of **BNRItem** persisted between runs of the application, and in this chapter, you will use *archiving* to save and load **BNRItem** 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 **initWithCoder:**.

```
@protocol NSCoding  
- (void)encodeWithCoder:(NSCoder *)aCoder;  
- (instancetype)initWithCoder:(NSCoder *)aDecoder;  
@end
```

Make **BNRItem** conform to **NSCoding**. Open **Homepwner.xcodeproj** and add this protocol declaration in **BNRItem.h**.

```
@interface BNRItem : NSObject <NSCoding>
```

Now you need to implement the required methods. Let's start with **encodeWithCoder:**. When a **BNRItem** 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 `BNRItem.m`, implement **encodeWithCoder:** to add the names and values of the item's properties to the stream.

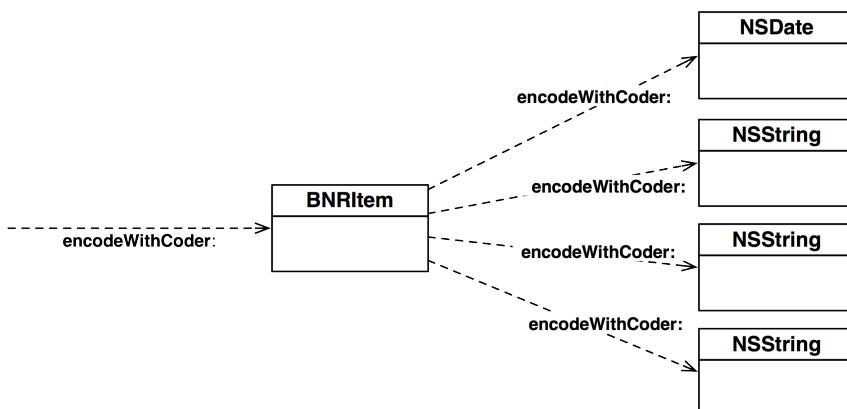
```
- (void)encodeWithCoder:(NSCoder *)aCoder
{
    [aCoder encodeObject:self.itemName forKey:@"itemName"];
    [aCoder encodeObject:self.serialNumber forKey:@"serialNumber"];
    [aCoder encodeObject:self.dateCreated forKey:@"dateCreated"];
    [aCoder encodeObject:self.itemKey forKey:@"itemKey"];

    [aCoder encodeInt:self.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 18.1). Thus, encoding an object is a recursive process where each object encodes its “friends”, and they encode their friends, and so on.

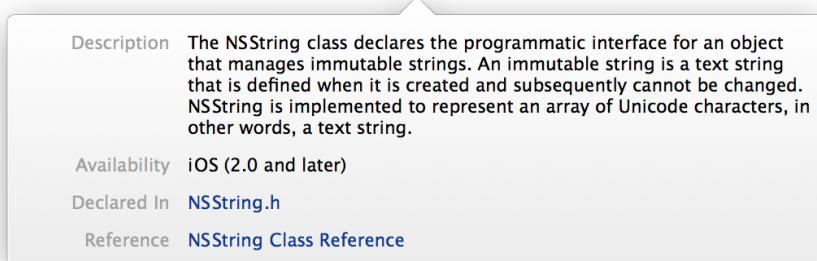
Figure 18.1 Encoding an object



To be encoded, these objects must also conform to **NSCoding**. Let's confirm this for **NSString** 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 `BNRItem.m`, hold down the Option key, mouse over an occurrence of **NSString** 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 18.2 Option-clicking **NSString**



Click the link to be taken to the **NSString** 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 **NSSecureCoding** protocol, you will see that this protocol conforms to **NSCoding**. Thus, **NSString** 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 **BNRItem** is loaded from the filesystem later. Objects being loaded from an archive are sent the message **initWithCoder:**. This method should grab all of the objects that were encoded in **encodeWithCoder:** and assign them to the appropriate instance variable.

In **BNRItem.m**, implement **initWithCoder:**.

```
- (instancetype)initWithCoder:(NSCoder *)aDecoder
{
    self = [super init];
    if (self) {
        _itemName = [aDecoder decodeObjectForKey:@"itemName"];
        _serialNumber = [aDecoder decodeObjectForKey:@"serialNumber"];
        _dateCreated = [aDecoder decodeObjectForKey:@"dateCreated"];
        _itemKey = [aDecoder decodeObjectForKey:@"itemKey"];

        _valueInDollars = [aDecoder decodeIntForKey:@"valueInDollars"];
    }
    return self;
}
```

Notice that this method has an **NSCoder** argument, too. In **initWithCoder:**, the **NSCoder** is full of data to be consumed by the **BNRItem** being initialized. Also notice that you sent **decodeObjectForKey:** to the container to get objects back and **decodeIntForKey:** to get the **valueInDollars**.

In Chapter 2, we talked about the initializer chain and designated initializers. The **initWithCoder:** method is not part of this design pattern; you will keep **BNRItem**'s designated initializer the same, and **initWithCoder:** will not call it.

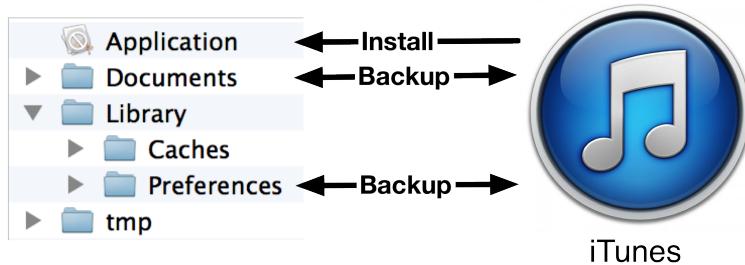
(By the way, archiving is how XIB files are created. `UIView` conforms to `NSCoding`. Instances of `UIView` are created when you drag them onto the canvas area. When the XIB file is saved, these views are archived into the XIB file. When your application launches, it unarchives the views from the XIB file. There are some minor differences between a XIB file and a standard archive, but overall it is the same process.)

Instances of `BNRItem` 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 18.3 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.

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 Chapter 26) 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 **BNRItem** from Homeowner will be saved to a single file in the **Documents** directory. The **BNRItemStore** will handle writing to and reading from that file. To do this, the **BNRItemStore** needs to construct a path to this file.

Implement a new method in **BNRItemStore.m** to do this.

```
- (NSString *)itemArchivePath
{
    // Make sure that the first argument is NSDocumentDirectory
    // and not NSDocumentationDirectory
    NSArray *documentDirectories =
        NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                            NSUserDomainMask, YES);

    // Get the one document directory from that list
    NSString *documentDirectory = [documentDirectories firstObject];

    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 a constant that specifies the directory in the sandbox you want the path to. For example, searching for `NSCachesDirectory` will return the `Caches` directory in the application's sandbox.

You can search the documentation for one of the constants – like `NSDocumentDirectory` – to locate the other options. Remember that these constants 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 `BNRItem` 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 `BNRItem`, you will use the class `NSKeyedArchiver` when the application “exits.”

In `BNRItemStore.h`, declare a new method.

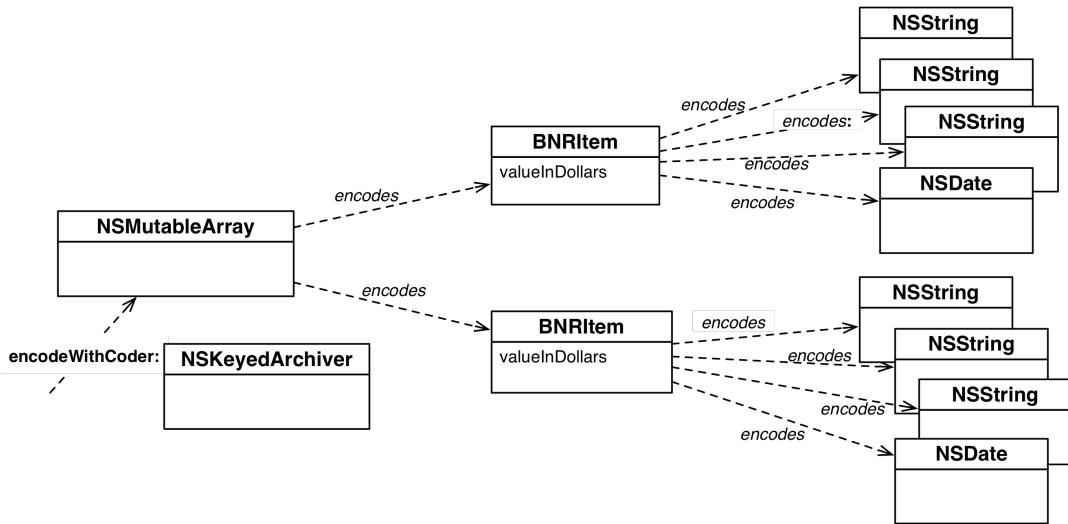
```
- (BOOL)saveChanges;
```

Implement this method in `BNRItemStore.m` to send the message `archiveRootObject:toFile:` to the `NSKeyedArchiver` class.

```
- (BOOL)saveChanges
{
    NSString *path = [self itemArchivePath];
    // Returns YES on success
    return [NSKeyedArchiver archiveRootObject:self.privateItems
                                         toFile:path];
}
```

The `archiveRootObject:toFile:` method takes care of saving every single `BNRItem` in `privateItems` 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`.)
- `privateItems` is sent the message `encodeWithCoder:` and is passed the instance of `NSKeyedArchiver` as an argument.
- The `privateItems` array then sends `encodeWithCoder:` to all of the objects it contains, passing the same `NSKeyedArchiver`. Thus, all your instances of `BNRItem` encode their instance variables into the very same `NSKeyedArchiver` (Figure 18.4).
- The `NSKeyedArchiver` writes the data it collected to the path.

Figure 18.4 Archiving the `privateItems` array

When the user presses the Home button on the device, the message `applicationDidEnterBackground:` is sent to the `BNRAppDelegate`. That is when you want to send `saveChanges` to the `BNRItemStore`.

In `BNRAppDelegate.m`, implement `applicationDidEnterBackground:` to kick off saving the items. Make sure to import the header file for `BNRItemStore` at the top of this file.

```

#import "BNRItemStore.h"

@implementation BNRAppDelegate

- (void)applicationDidEnterBackground:(UIApplication *)application
{
    BOOL success = [[BNRItemStore sharedStore] saveChanges];
    if (success) {
        NSLog(@"Saved all of the BNRItems");
    }
    else {
        NSLog(@"Could not save any of the BNRItems");
    }
}
  
```

(This method may have already been implemented by the template. If so, make sure to add code to the existing method instead of writing a brand new one.)

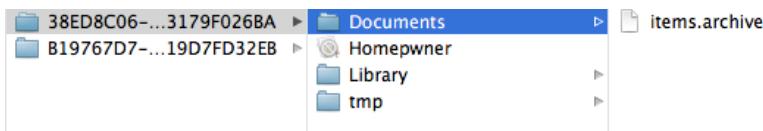
Build and run the application on the simulator. Create a few instances of `BNRItem`. Then, press the Home button to leave the application. Check the console, and you should see a log statement indicating that the items were saved.

While you cannot yet load these instances of `BNRItem` 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 7.0 (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 7.0. Unfortunately, these applications have really unhelpful names. You have to dig into each directory to find the one that contains Homepwner.

Figure 18.5 Homepwner's sandbox



In Homepwner's directory, navigate into the Documents directory (Figure 18.5). 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 `BNRItem` when the application launches, you will use the class `NSKeyedUnarchiver` when the `BNRItemStore` is created.

In `BNRItemStore.m`, add the following code to `initPrivate`.

```
- (instancetype)initPrivate
{
    self = [super init];
    if (self) {
        _privateItems = [[NSMutableArray alloc] init];

        NSString *path = [self itemArchivePath];
        _privateItems = [NSKeyedUnarchiver unarchiveObjectWithFile:path];

        // If the array hadn't been saved previously, create a new empty one
        if (!_privateItems) {
            _privateItems = [[NSMutableArray alloc] init];
        }
    }
    return self;
}
```

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 `NSMutableArray` because you created this archive with a root object of this type. (If the root object was a `BNRItem` instead, `unarchiveObjectWithFile:` would return an instance of `BNRItem`.)

The newly allocated `NSMutableArray` is then sent `initWithCoder:` and, as you may have guessed, the `NSKeyedUnarchiver` is passed as the argument. The array starts decoding its contents (instances of `BNRItem`) from the `NSKeyedUnarchiver` and sends each of these objects the message `initWithCoder:`, 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

Homeowner from Xcode, `applicationDidEnterBackground:`: will not get a chance to be called 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 `BNRItemStore.m`, modify the implementation of `createItem` so that it creates an empty `BNRItem` instead of one with random data.

```
- (BNRItem *)createItem
{
    BNRItem *item = [BNRItem randomItem];
    BNRItem *item = [[BNRItem alloc] init];

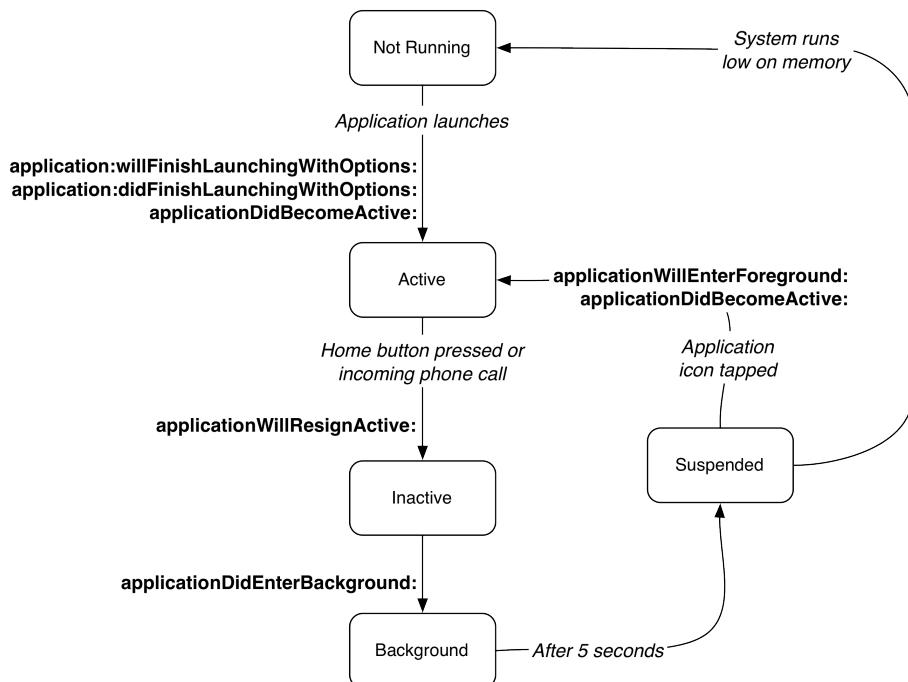
    [self.privateItems addObject:item];

    return item;
}
```

Application States and Transitions

In Homeowner, 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 18.6.

Figure 18.6 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.

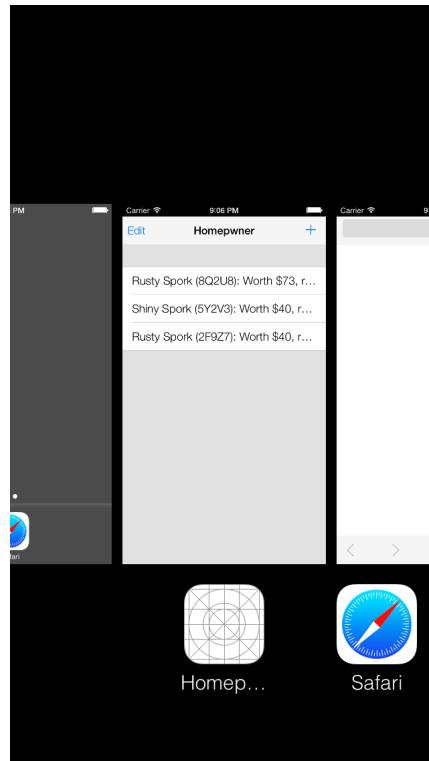
An application in the suspended state cannot execute code, you cannot see its interface, and any resources it does not need while suspended are destroyed. A suspended application is essentially freeze-dried and can be quickly thawed when the user relaunches it. Table 18.1 summarizes the characteristics of the different application states.

Table 18.1 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 |
| Suspended | No | No | No |

You can see what applications are in the background or suspended by double-clicking the Home button to get to the multitasking display. (Recently run applications that have been terminated may also appear in this display.)

Figure 18.7 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 18.6.)

- (BOOL)**application:(UIApplication *)app didFinishLaunchingWithOptions:(NSDictionary *)options**
- (void)**applicationDidBecomeActive:(UIApplication *)app;**
- (void)**applicationWillResignActive:(UIApplication *)app;**
- (void)**applicationDidEnterBackground:(UIApplication *)app;**
- (void)**applicationWillEnterForeground:(UIApplication *)app;**

You can implement code in these methods to take the appropriate actions for your application. 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 `Homeowner` saves and loads the `itemKey` for each `BNRItem`, 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 `BNRItem` 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.

Open `BNRImageStore.m` and add a new method declaration to the class extension.

```
- (NSString *)imagePathForKey:(NSString *)key;
```

Implement `imagePathForKey:` in `BNRImageStore.m` to create a path in the documents directory using a given key.

```
- (NSString *)imagePathForKey:(NSString *)key
{
    NSArray *documentDirectories =
        NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                            NSUserDomainMask,
                                            YES);

    NSString *documentDirectory = [documentDirectories firstObject];

    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 `BNRImageStore.m`, modify `setImage:forKey:` to get a path and save the image.

```
- (void)setImage:(UIImage *)image forKey:(NSString *)key
{
    self.dictionary[key] = image;

    // Create full path for image
    NSString *imagePath = [self imagePathForKey:key];

    // Turn image into JPEG data
    NSData *data = UIImageJPEGRepresentation(image, 0.5);

    // Write it to full path
    [data writeToFile:imagePath atomically:YES];
}
```

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 YES, 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 **BNRImageStore.m**, make sure that when an image is deleted from the store, it is also deleted from the filesystem:

```
- (void)deleteImageForKey:(NSString *)key
{
    if (!key) {
        return;
    }
    [self.dictionary removeObjectForKey:key];

    NSString *imagePath = [self imagePathForKey:key];
    [[NSFileManager defaultManager] removeItemAtPath:imagePath
                                              error:nil];
}
```

Now that the image is stored in the filesystem, the **BNRImageStore** 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 **BNRImageStore.m**, replace the method **imageForKey:** so that the **BNRImageStore** will load the image from the filesystem if it does not already have it.

```
- (UIImage *)imageForKey:(NSString *)key
{
    //return self.dictionary[key];

    // If possible, get it from the dictionary
    UIImage *result = self.dictionary[key];

    if (!result) {
        NSString *imagePath = [self imagePathForKey:key];

        // Create UIImage object from file
        result = [UIImage imageWithContentsOfFile:imagePath];

        // If we found an image on the file system, place it into the cache
        if (result) {
            self.dictionary[key] = result;
        }
        else {
            NSLog(@"Error: unable to find %@", [self imagePathForKey:key]);
        }
    }
    return result;
}
```

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 **BNRItem** 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.

NSNotificationCenter and 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, are sent the message **didReceiveMemoryWarning**.

Objects other than view controllers may have data that they are not using and can recreate later. The **BNRImageStore** is such an object – when a low-memory warning occurs, it can release its ownership of the images by emptying its **dictionary**. 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. 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 a low-memory warning occurs, **UIApplicationDidReceiveMemoryWarningNotification** is posted to the notification center. Objects that want to implement their own low-memroy warning handlers can register for this notification.

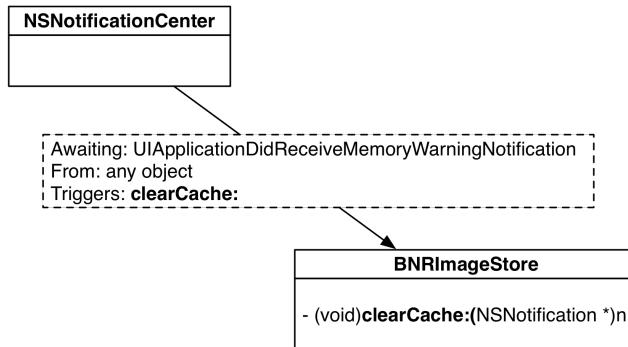
In **BNRImageStore.m**, edit the **initPrivate** method to register the image store as an observer of this notification.

```
- (instancetype)initPrivate
{
    self = [super init];
    if (self) {
        _dictionary = [[NSMutableDictionary alloc] init];

        NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
        [nc addObserver:self
                  selector:@selector(clearCache:)
                    name:UIApplicationDidReceiveMemoryWarningNotification
                  object:nil];
    }
    return self;
}
```

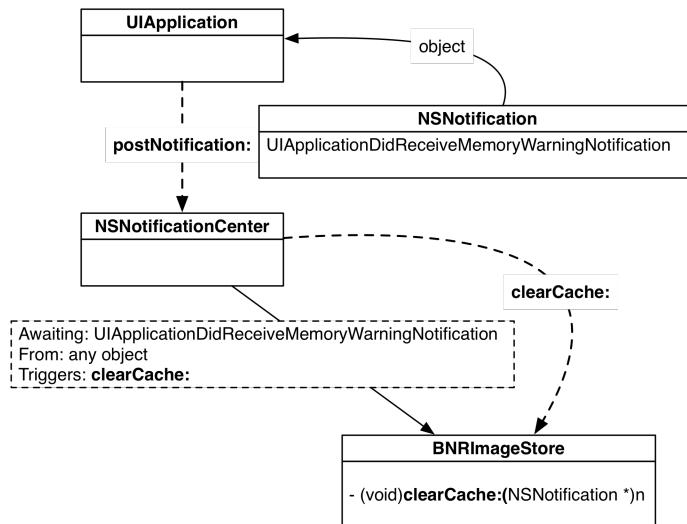
Now your image store is registered as an observer with the notification center (Figure 18.8).

Figure 18.8 Registered as an observer with notification center



Now, when a low-memory warning is posted, the notification center will send the message **clearCache:** to the **BNRIImageStore** instance (Figure 18.9).

Figure 18.9 Receiving the notification



In **BNRIImageStore.m**, implement **clearCache:** to remove all the instances of **UIImage** from the **BNRIImageStore**'s dictionary.

```

- (void)clearCache:(NSNotification *)note
{
    NSLog(@"flushing %d images out of the cache", [self.dictionary count]);
    [self.dictionary removeAllObjects];
}
  
```

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 **BNRDetailViewController**'s `imageView`, then it will not be destroyed since it is owned by the `imageView`. When the **BNRDetailViewController**'s `imageView` loses ownership of that image (either because the **BNRDetailViewController** 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 in Objective-C 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:

```
- (void)statusBarMovedOrResized:(NSNotification *)note
{
    NSDictionary *userInfo = [note userInfo];
    NSValue *wrappedRect = userInfo[UIApplicationStatusBarFrameUserInfoKey];
    CGRect newFrame = [wrappedRect CGRectValue];

    ...use frame here...
}
```

Notice that the `CGRect` had to be wrapped in an **NSValue** because only objects can go into dictionaries.

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 pointer to an 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 **BNRItemStore** to allow it to save and load **BNRItem** instances from the filesystem. The controller object asks the **BNRItemStore** 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 **BNRItemStore** 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.

You already know about `self`, an implicit variable that points to the instance that is executing the current method. There is another implicit variable called `_cmd`, which is the selector for the current method. You can get the **NSString** representation of a selector with the function **NSStringFromSelector**.

In `BNRAppDelegate.m`, 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.)

```
- (void)applicationWillResignActive:(UIApplication *)application
{
    NSLog(@"%@", NSStringFromSelector(_cmd));
}

- (void)applicationDidEnterForeground:(UIApplication *)application
{
    NSLog(@"%@", NSStringFromSelector(_cmd));
}

- (void)applicationDidBecomeActive:(UIApplication *)application
{
    NSLog(@"%@", NSStringFromSelector(_cmd));
}

- (void)applicationWillTerminate:(UIApplication *)application
{
    NSLog(@"%@", NSStringFromSelector(_cmd));
}
```

Now, add the following `NSLog` statements to the top of `application:didFinishLaunchingWithOptions:` and `applicationDidEnterBackground:`.

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    NSLog(@"%@", NSStringFromSelector(_cmd));
    ...
}

- (void)applicationDidEnterBackground:(UIApplication *)application
{
    NSLog(@"%@", NSStringFromSelector(_cmd));
    [[BNRItemStore sharedStore] saveChanges];
}
```

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-click 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 23. A couple of others are worth mentioning here.

You have access to the standard file I/O functions from the C library. These functions look like this:

```
FILE *inFile = fopen("textfile", "rt");
char *buffer = malloc(someSize);
fread(buffer, byteCount, 1, inFile);

FILE *outFile = fopen("binaryfile", "w");
fwrite(buffer, byteCount, 1, outFile);
```

However, you will not see these functions used much because there are more convenient ways of reading and writing binary and text data. Using **NSData** works well for binary data. For text data, **NSString** has two instance methods **writeToFile:atomically:encoding:error:** and **initWithContentsOfFile:.** They are used as follows:

```
// A local variable to store an error object if one comes back
NSError *err;

NSString *someString = @"Text Data";
BOOL success = [someString writeToFile:@"/some/path/file"
    atomically:YES
    encoding:NSUTF8StringEncoding
    error:&err];
if (!success) {
    NSLog(@"Error writing file: %@", [err localizedDescription]);
}

NSString *myEssay = [[NSString alloc] initWithContentsOfFile:@"/some/path/file"
    encoding:NSUTF8StringEncoding
    error:&err];
if (!myEssay) {
    NSLog(@"Error reading file: %@", [err 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 send the message **localizedDescription** to an instance 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 (**&err**) 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**.

Sometimes you want to show the error to the user. This is typically done with a **UIAlertView** (Figure 18.10).

Figure 18.10 Example of **UIAlertView**

Creating a **UIAlertView** looks like this:

```
NSString *x = [[NSString alloc] initWithContentsOfFile:@"/some/path/file"
                                                encoding:NSUTF8StringEncoding
                                                error:&err];
if (!x) {
    UIAlertView *a = [[UIAlertView alloc] initWithTitle:@"Read Failed"
                                                message:[err localizedDescription]
                                                delegate:nil
                                                cancelButtonTitle:@"OK"
                                                otherButtonTitles:nil];
    [a show];
}
```

Note that in many languages, anything unexpected results in an exception being thrown. Among Objective-C 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.

Like **NSString**, the classes **NSDictionary** and **NSArray** have **writeToFile:** and **initWithContentsOfFile:** methods. To write collection objects to the filesystem with these methods, the collection objects must contain only *property list serializable* objects. The only objects that are property list serializable are **NSString**, **NSNumber**, **NSDate**, **NSData**, **NSArray**, and **NSDictionary**. When an **NSArray** or **NSDictionary** 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>Joe</string>
    <key>lastName</key>
    <string>Conway</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:

```

NSMutableDictionary *d = [NSMutableDictionary dictionary];
d[@"String"] = @"A string";
[d writeToFile:@"/some/path/file" atomically:YES];

NSMutableDictionary *anotherD = [[NSMutableDictionary alloc]
    initWithContentsOfFile:@"/some/path/file"];

```

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.

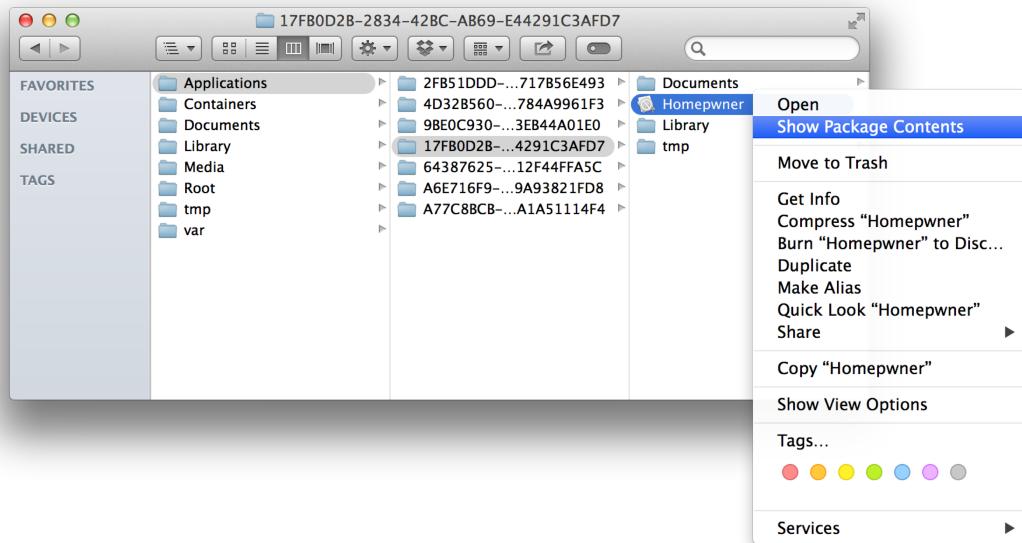
How can you tell which files are being bundled with your application? Select the Homepwner project from the project navigator. Check out the Build Phases pane in the Homepwner target. Everything under Copy Bundle Resources will be added to the application bundle when it is built.

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

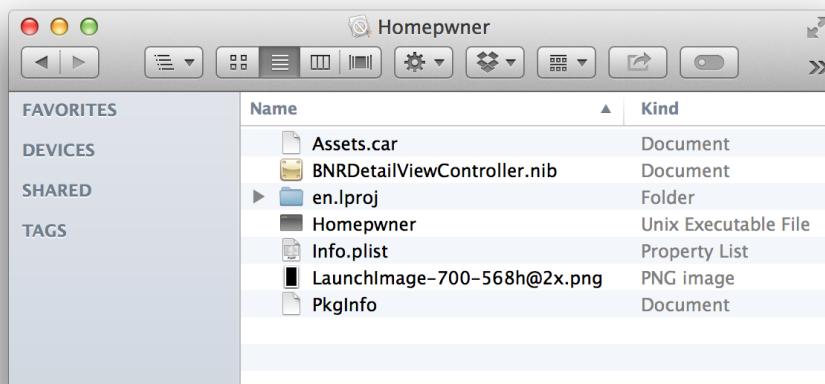
for applications installed on your computer’s iOS simulator. Opening one of these directories will show you what you expect in an application sandbox: an application bundle and the Documents, tmp, and Library directories. Right- or Command-click the application bundle and choose Show Package Contents from the contextual menu (Figure 18.11).

Figure 18.11 Viewing an application bundle



A Finder window will appear showing you the contents of the application bundle (Figure 18.12). When a user downloads your application from the App Store, these files are copied to their device.

Figure 18.12 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 pointer to the application bundle
NSBundle *applicationBundle = [NSBundle mainBundle];

// Ask for the path to a resource named myImage.png in the bundle
NSString *path = [applicationBundle pathForResource:@"myImage"
                                             ofType:@"png"];
```

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.

This page intentionally left blank

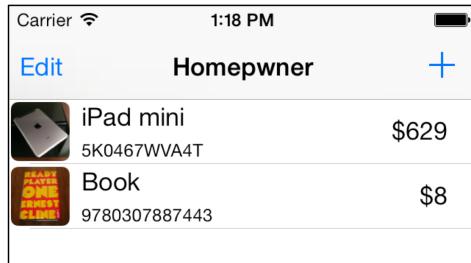
19

Subclassing UITableViewCell

A **UITableView** displays a list of **UITableViewCell** objects. 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 chapter, you will create a custom subclass of **UITableViewCell** named **BNRItemCell** that will display **BNRItem** instances more effectively. Each of these cells will show a **BNRItem**'s name, its value in dollars, its serial number, and a thumbnail of its image (Figure 19.1).

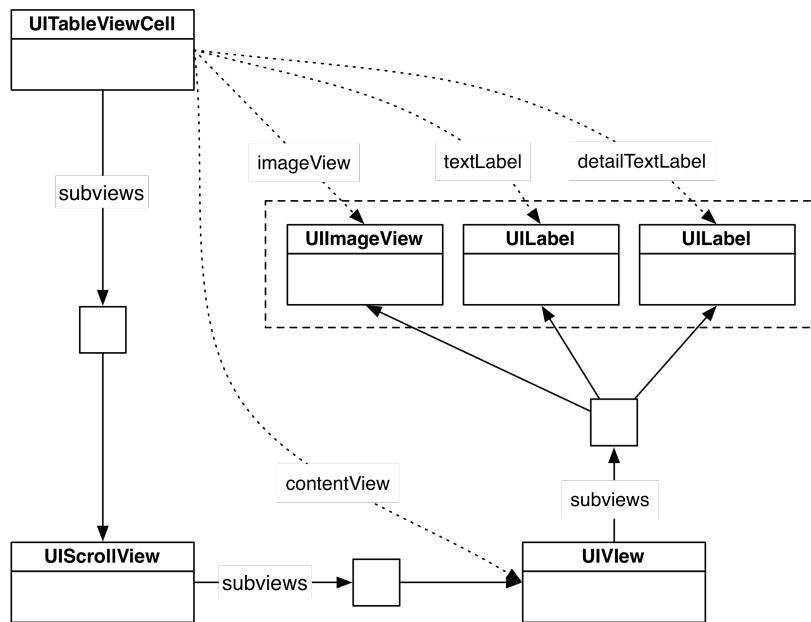
Figure 19.1 Homepwner with subclassed table view cells



Creating BNRItemCell

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*.

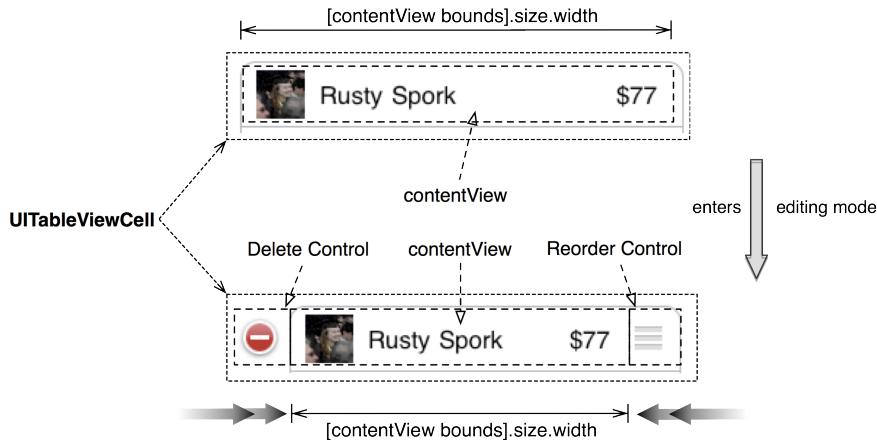
Each cell has a subview named `contentView`, which is a container for the view objects that make up the layout of a cell subclass (Figure 19.2). When you subclass **UITableViewCell**, you often change its look and behavior by changing the subviews of the cell's `contentView`. For instance, you could create instances of the classes **UITextField**, **UILabel**, and **UIButton** and add them to the `contentView`.

Figure 19.2 **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 19.3). 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 19.3 Table view cell layout in standard and editing mode



Open Homeowner.xcodeproj. Create a new **NSObject** subclass and name it **BNRItemCell**.

In **BNRItemCell.h**, change the superclass to **UITableViewCell**.

```
@interface BNRItemCell : NSObject
@interface BNRItemCell : UITableViewCell
```

Configuring a UITableViewCell subclass's interface

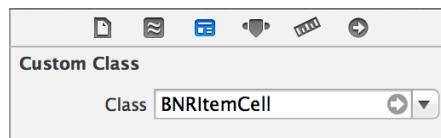
The easiest way to configure a **UITableViewCell** subclass is with a XIB file. Create a new Empty XIB file and name this file **BNRItemCell.xib**. (The Device Family is irrelevant for this file.)

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

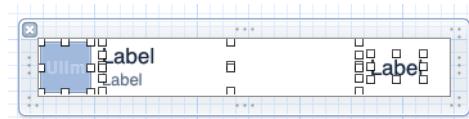
In **BNRItemCell.xib**, select **BNRItemCell.xib** and drag a **UITableViewCell** instance from the object library to the canvas. (Make sure you choose **UITableViewCell**, not **UITableView** or **UITableViewCellController**.)

Select the Table View Cell in the outline view and then the identity inspector (the **Identity** tab). Change the Class to **BNRItemCell** (Figure 19.4).

Figure 19.4 Changing the cell class



A **BNRItemCell** will display three text elements and an image, so drag three **UILabel** objects and one **UIImageView** object onto the cell. Configure them as shown in Figure 19.5. Make the text of the bottom label a slightly smaller font and a dark shade of gray.

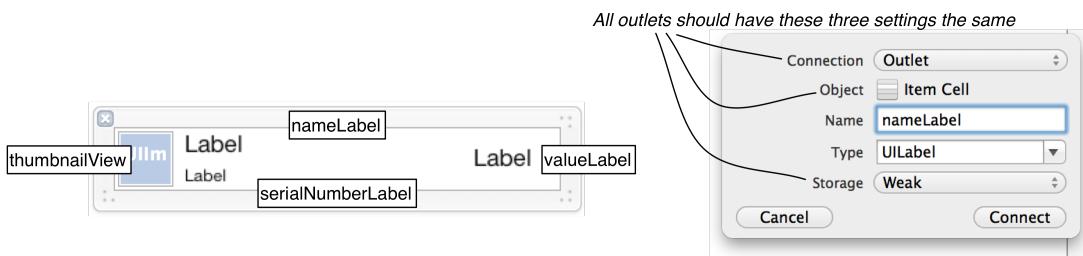
Figure 19.5 **BNRItemCell**'s layout

Exposing the properties of **BNRItemCell**

In order for **BNRItemsViewController** to configure the content of a **BNRItemCell** in **tableView:cellForRowAtIndexPath:**, the cell must have properties that expose the three labels and image view. These properties will be set through outlet connections in **BNRItemCell.xib**.

The next step, then, is to create and connect outlets on **BNRItemCell** for each of its subviews. You will use the same technique you have been using in the last few chapters – Control-dragging from the XIB file into the source file to create the outlets.

Option-click on **BNRItemCell.h** while **BNRItemCell.xib** is open. Control-drag from each subview to the method declaration area in **BNRItemCell.h**. Name each outlet and configure the other attributes of the connection as shown in Figure 19.6. (Pay attention to the Connection, Storage, and Object fields.)

Figure 19.6 **BNRItemCell** connections

Double-check that **BNRItemCell.h** looks like this:

```
@interface BNRItemCell : UITableViewCell

@property (weak, nonatomic) IBOutlet UIImageView *thumbnailView;
@property (weak, nonatomic) IBOutlet UILabel *nameLabel;
@property (weak, nonatomic) IBOutlet UILabel *serialNumberLabel;
@property (weak, nonatomic) IBOutlet UILabel *valueLabel;

@end
```

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 **BNRItemCell**

In **BNRItemsViewController**'s **tableView:cellForRowAtIndexPath:** method, you will create an instance of **BNRItemCell** for every row in the table.

In **BNRItemsViewController.m**, import the header file for **BNRItemCell** so that **BNRItemsViewController** knows about it.

```
#import "BNRItemCell.h"
```

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 `BNRItemsViewController.m`, modify `viewDidLoad` to register `BNRItemCell.xib` for the "BNRItemCell" reuse identifier.

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    [self.tableView registerClass:[UITableViewCell class]
        forCellReuseIdentifier:@"UITableViewCell"];

    // Load the NIB file
    UINib *nib = [UINib nibWithNibName:@"BNRItemCell" bundle:nil];

    // Register this NIB, which contains the cell
    [self.tableView registerNib:nib
        forCellReuseIdentifier:@"BNRItemCell"];
}
```

The registration of a NIB for a table view is not anything fancy: the table view simply stores the `UINib` instance in an `NSDictionary` for the key "BNRItemCell". 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 `BNRItemCell` when given the reuse identifier "BNRItemCell".

In `BNRItemsViewController.m`, modify `tableView:cellForRowIndexPath:`.

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:@"UITableViewCell"
            forIndexPath:indexPath];

    // Get a new or recycled cell
    BNRItemCell *cell =
        [tableView dequeueReusableCellWithIdentifier:@"BNRItemCell"
            forIndexPath:indexPath];

    NSArray *items = [[BNRItemStore sharedStore] allItems];
    BNRItem *item = items[indexPath.row];
    cell.textLabel.text = item.description;

    // Configure the cell with the BNRItem
    cell.nameLabel.text = item.itemName;
    cell.serialNumberLabel.text = item.serialNumber;
    cell.valueLabel.text =
        [NSString stringWithFormat:@"%d", item.valueInDollars];

    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 `BNRItem`. (You will deal with the `thumbnailView` later.)

Build and run the application and create a new `BNRItem`. The cells load, but the layout of each cell is most likely off. You have not set up constraints for each of the subviews of the cell's `contentView`, so let's do that now.

Constraints for `BNRItemCell`

`BNRItemsViewController`'s table view will change its size to match the size of the window. When a table view changes its width, each of its cells also change their width to match. Thus, you need to set up constraints in the cell that account for this change in width. (The height of a cell will not change unless you explicitly ask the table view to change it, either through the property `rowHeight` or its delegate method `tableView:heightForRowAtIndexPath::`)

Right now, `BNRItemCell.xib` has an initial position and size for each view. For example, the `thumbnailView`'s size in the XIB file is 40 points wide and 40 points tall (if yours is not exactly 40x40, do not worry; it will be soon.). However, Auto Layout does not care about how big a view is when it is first created; it only cares about what the constraints say. If you were to add a constraint to the image view that pins its width to 500 points, the width would be 500 points – the original size does not factor in.

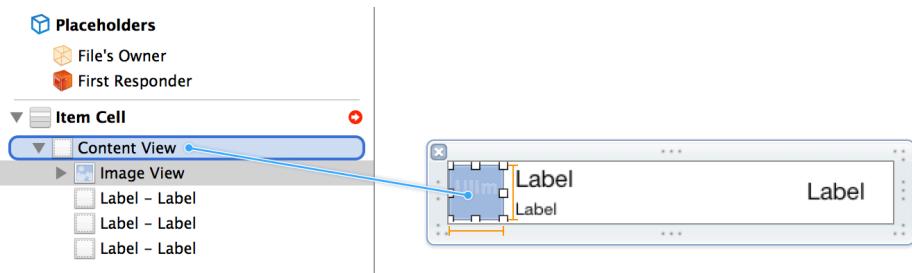
Here are the constraints you need:

1. Make the `UIImageView` 40x40 points, close to the left edge of the content view and vertically centered with the content view.
2. Make sure both the `nameLabel` and `serialNumberLabel` stay the same fixed distance away from the image view, stretch to fill the length of the cell (minus the size of the image view and the `valueLabel`), and maintain their vertical stacking.
3. Keep the `valueLabel` centered vertically with the content view on the right edge of the cell and a fixed distance away from the other two labels.

First, pin the width and height of `imageView`. You can use the Pin menu or Control-click and drag diagonally from the image view to itself.

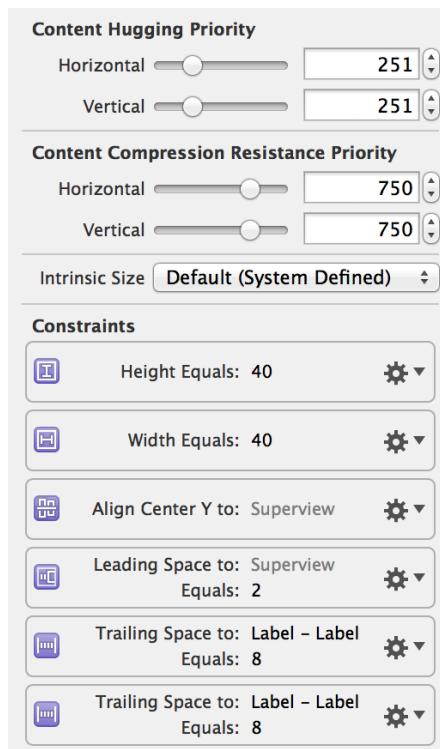
Next, center the image view within its container. You can use the Align menu to do this (selecting `Vertical Center in container`) or Control-click and drag from the image view to the container (selecting `Center Vertically in Container`). If you Control-click and drag, make sure you do not drag to another subview. An easy way to make sure you drag to the correct view is to drag to the Content View in the document outline (Figure 19.7).

Figure 19.7 Dragging to the document outline



Let's set up the horizontal constraints for all of the views at once. Select the four subviews and then, from the Pin menu, check the left and right struts at the top. Click Add 6 Constraints. The image view now has all of the constraints that it needs, as evident by its blue constraint lines. (If yours are not blue yet, do not worry. Your image view's frame may not match its constraints, and you will make sure this is fixed soon.) The completed constraints for the image view are shown in Figure 19.8.

Figure 19.8 Image view constraints



Now, finish the constraints for nameLabel and serialNumberLabel. Select these two labels and open the Pin menu. Check the top and bottom strut, as well as Height, and click Add 5 Constraints. The constraints for those two labels will turn blue, but there could be unsatisfiable constraints if the table

view cell's height ever changes. Why? Right now, all of the vertical constraints that were added have their equality set to Equal. In the visual format language, this gives the equation:

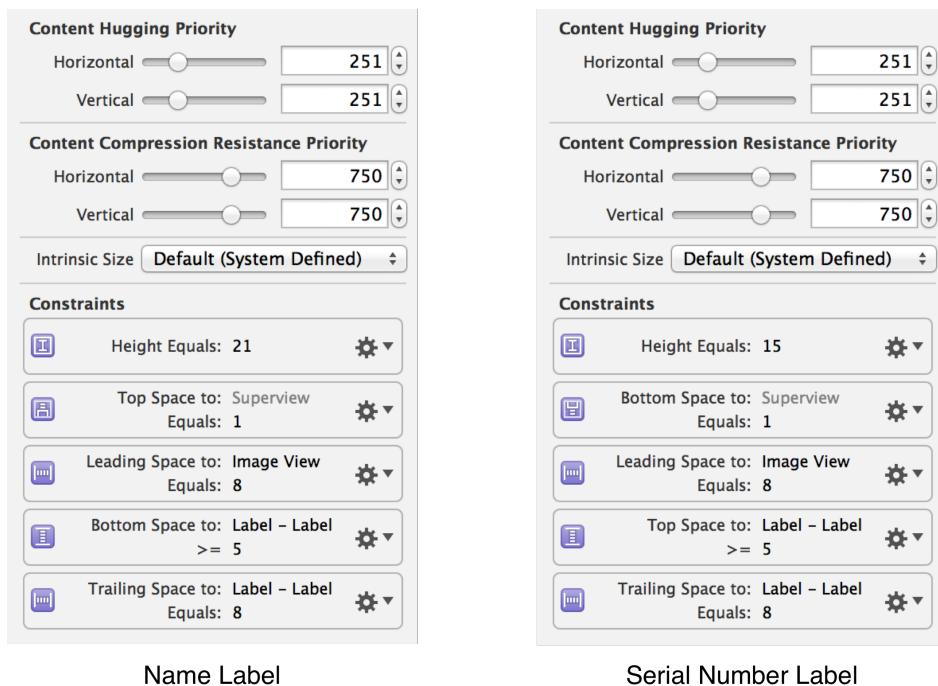
```
V: | -1-[ nameLabel(==21)] -5-[ serialNumberLabel(==15) ] -1- |
```

Notice that these add up to 43, exactly the current height of the content view. If the height of the table view cell changes, one of these constraints will have to break. (Want to see this? Go ahead and change the height of the cell from the Size Inspector. It is probably a good idea to change it back after you see the effects, but it is not absolutely necessary.) You will fix this by changing the relation of one of the constraints to be greater than or equal.

The constraint that you will modify is the one that pins the bottom of `nameLabel` to the top of `serialNumberLabel`. While you could select this constraint in the canvas, it is very small and difficult to select. Instead, select `nameLabel`, and then open up its Size inspector.

Here you can see all of the constraints that affect the selected view. Click on the gear icon associated with the Bottom Space constraint. This will pop up a menu allowing you to Select and Edit... or Delete the constraint. Choose Select and Edit.... At the top of the Attributes inspector, change the Relation to Greater Than or Equal. The constraints for these two labels are shown in Figure 19.9.

Figure 19.9 Name and serial number label constraints



Next, select `valueLabel` and select and add the constraint for Vertical Center in Container from the Align menu.

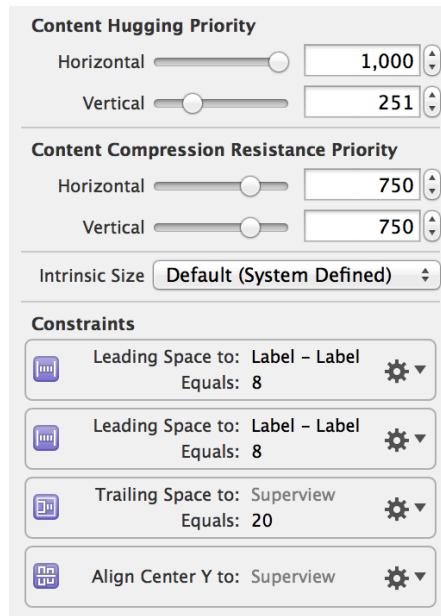
You need to make one last change to get rid of an ambiguous layout. Since all three labels do not have their widths pinned, they all want to be the width of their `intrinsicContentSize`. Since the

width of the two labels in the middle + the spacing + the width of the value label is greater than their intrinsic widths + that spacing, something will have to give: either the width of `nameLabel` and `serialNumberLabel` will have to grow, or the width of `valueLabel` will have to grow.

Let's fix this ambiguity by adjusting the priority of the value label's Content Hugging Priority to be higher than that of the other two labels. This is a better approach than pinning the width of the value label. If the width is pinned, then text longer than can be displayed will be truncated. By increasing the Content Hugging Priority, the value label's width will always be exactly the width needed to display all of the text. (Unless the label is so long that the text must be truncated to satisfy all the constraints.)

Select `valueLabel` and open the Size Inspector. Change the Horizontal Content Hugging Priority to be 1000. The finished `valueLabel` constraints are shown in Figure 19.10.

Figure 19.10 Value label constraints



You are done! All of the subviews should have blue constraint lines. If any do not, select the cell and then click Update All Frames in Item Cell from the Resolve Auto Layout Issues menu.

It was a bit of work to configure the cell, but its contents will now scale elegantly if the size of the cell changes or if the textual content changes.

Image Manipulation

Now let's address the `thumbnailView`'s contents in `BNRItemCell`. To display an image within a cell, you could just resize the large image of the item from the image store. However, doing so would incur a performance penalty because a large number of bytes would need to be read, filtered, and resized to fit within the cell. A better idea is to create and use a thumbnail of the image instead.

To create a thumbnail of a **BNRItem** image, you are going to draw a scaled-down version of the full image to an offscreen context and keep a reference to that new image inside a **BNRItem** instance. You also need a place to store this thumbnail image so that it can be reloaded when the application launches again.

In Chapter 11, you put the full-sized images in the **BNRImageStore** so that they can be flushed if necessary. However, the thumbnail images will be small enough that you can archive them with the other **BNRItem** properties.

Open **BNRItem.h**. Declare a new property for a thumbnail and a new method that will configure that thumbnail.

```
@property (nonatomic, copy) NSString *itemKey;
@property (nonatomic, strong) UIImage *thumbnail;
- (void)setThumbnailFromImage:(UIImage *)image;
@end
```

When an image is chosen for a **BNRItem**, you will give that image to the **BNRItem**. It will chop it down to a much smaller size and then keep that smaller-sized image as its thumbnail.

The method that will do this is **setThumbnailFromImage:**. This method will take a full-sized image, create a smaller representation of it in an offscreen graphics context object, and set the **thumbnail** property to the image produced by the offscreen context. (If you do not know what a graphics context object is, read the section called “For the More Curious: Core Graphics” in Chapter 4).

iOS provides a convenient suite of functions to create offscreen contexts and produce images from them. To create an offscreen image context, you use the function **UIGraphicsBeginImageContextWithOptions**. This function accepts a **CGSize** structure that specifies the width and height of the image context, a scaling factor, and whether the image should be opaque. When this function is called, a new **CGContextRef** is created and becomes the current context.

To draw to a **CGContextRef**, you use Core Graphics, just as though you were implementing a **drawRect:** method for a **UIView** subclass. To get a **UIImage** from the context after it has been drawn, you call the function **UIGraphicsGetImageFromCurrentImageContext**.

Once you have produced an image from an image context, you must clean up the context with the function **UIGraphicsEndImageContext**.

In **BNRItem.m**, implement the following methods to create a thumbnail using an offscreen context.

```

- (void)setThumbnailFromImage:(UIImage *)image
{
    CGSize origImageSize = image.size;

    // The rectangle of the thumbnail
    CGRect newRect = CGRectMake(0, 0, 40, 40);

    // Figure out a scaling ratio to make sure we maintain the same aspect ratio
    float ratio = MAX(newRect.size.width / origImageSize.width,
                      newRect.size.height / origImageSize.height);

    // Create a transparent bitmap context with a scaling factor
    // equal to that of the screen
    UIGraphicsBeginImageContextWithOptions(newRect.size, NO, 0.0);

    // Create a path that is a rounded rectangle
    UIBezierPath *path = [UIBezierPath bezierPathWithRoundedRect:newRect
                                                          cornerRadius:5.0];

    // Make all subsequent drawing clip to this rounded rectangle
    [path addClip];

    // Center the image in the thumbnail rectangle
    CGRect projectRect;
    projectRect.size.width = ratio * origImageSize.width;
    projectRect.size.height = ratio * origImageSize.height;
    projectRect.origin.x = (newRect.size.width - projectRect.size.width) / 2.0;
    projectRect.origin.y = (newRect.size.height - projectRect.size.height) / 2.0;

    // Draw the image on it
    [image drawInRect:projectRect];

    // Get the image from the image context; keep it as our thumbnail
    UIImage *smallImage = UIGraphicsGetImageFromCurrentImageContext();
    self.thumbnail = smallImage;

    // Cleanup image context resources; we're done
    UIGraphicsEndImageContext();
}

```

In `BNRDetailViewController.m`, add the following line of code to `imagePickerController:didFinishPickingMediaWithInfo:` to create a thumbnail when the camera takes the original image.

```

- (void)imagePickerController:(UIImagePickerController *)picker
didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    UIImage *image = info[UIImagePickerControllerOriginalImage];
    [self.item setThumbnailFromImage:image];
}

```

Now that instances of `BNRItem` have a thumbnail, you can use this thumbnail in `BNRItemsViewController`'s table view. In `BNRItemsViewController.m`, update `tableView:cellForRowAtIndexPath:`.

```

    cell.valueLabel.text =
        [NSString stringWithFormat:@"%@", item.valueInDollars];

    cell.thumbnailView.image = item.thumbnail;

    return cell;
}

```

Now build and run the application. Take a picture for a **BNRItem** instance and return to the table view. That row will display a thumbnail image along with the name and value of the **BNRItem**. (Note that you will have to retake pictures for existing items.)

Do not forget to add the thumbnail data to your archive! Open **BNRItem.m** and make the following changes:

```

- (id)initWithCoder:(NSCoder *)aDecoder
{
    self = [super init];

    if (self) {
        _itemName = [aDecoder decodeObjectForKey:@"itemName"];
        _serialNumber = [aDecoder decodeObjectForKey:@"serialNumber"];
        _dateCreated = [aDecoder decodeObjectForKey:@"dateCreated"];
        _itemKey = [aDecoder decodeObjectForKey:@"itemKey"];
        _thumbnail = [aDecoder decodeObjectForKey:@"thumbnail"];

        _valueInDollars = [aDecoder decodeIntForKey:@"valueInDollars"]
    }
}

return self;
}

- (void)encodeWithCoder:(NSCoder *)aCoder
{
    [aCoder encodeObject:self.itemName forKey:@"itemName"];
    [aCoder encodeObject:self.serialNumber forKey:@"serialNumber"];
    [aCoder encodeObject:self.dateCreated forKey:@"dateCreated"];
    [aCoder encodeObject:self.itemKey forKey:@"itemKey"];
    [aCoder encodeObject:self.thumbnail forKey:@"thumbnail"];

    [aCoder encodeInt:self.valueInDollars forKey:@"valueInDollars"];
}

```

Build and run the application. Take some photos of items and then exit and relaunch the application. The thumbnails will now appear for saved items.

Relying Actions from UITableViewCells

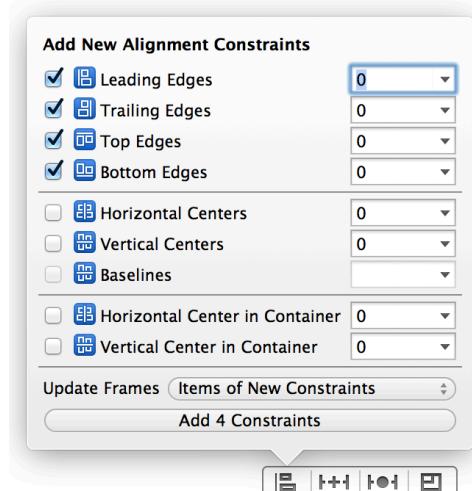
Sometimes, it is useful to add a **UIControl** or one of its subclasses, like a **UIButton**, to a **UITableViewCell**. For instance, you want users to be able to tap the thumbnail image in a cell and see a full-size image for that item. In this section, you will do that by adding a transparent button on top of the thumbnail. Tapping this button will show the full-size image in a **UIPopoverController** when the application is running on an iPad.

Open **BNRItemCell.m** and stub out a method that will trigger showing the image.

```
- (IBAction)showImage:(id)sender
{}
```

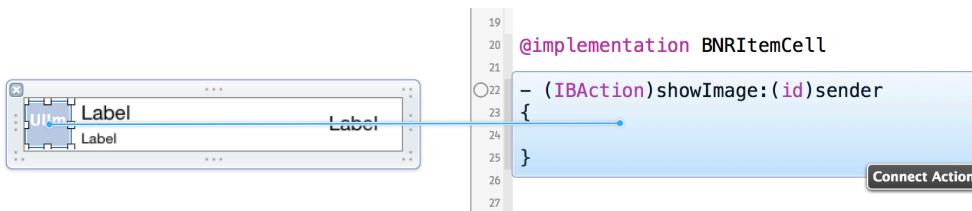
Now, in `BNRItemCell.xib`, drag a **UIButton** onto the content view. Remove the text from the button. Select both the **UIImageView** and **UIButton**, and then from the Align Auto Layout menu, select Leading Edges, Trailing Edges, Top Edges, and Bottom Edges. For the Update Frames drop-down, select Items of New Constraints, and then click Add 4 Constraints (Figure 19.11).

Figure 19.11 **UIButton** constraints



Finally, you need to connect the **UIButton** to the `showImage:` action. Open `BNRItemCell.xib` and `BNRItemCell.m` in the assistant editor, and Control-drag from the **UIButton** to the `showImage:` method (Figure 19.12).

Figure 19.12 **UIButton** constraints



So now you have a button that will send `showImage:` to **BNRItemCell** when it is tapped. Obviously, you will have to implement that method, but here you run into a problem: this message will be sent to the **BNRItemCell**, but **BNRItemCell** is not a controller and does not have access to any of the image data necessary to get the full-size image. In fact, it does not even have access to the **BNRItem** whose thumbnail it is displaying.

You might consider letting **BNRItemCell** keep a pointer to the **BNRItem** it displays. But table view cells are view objects, and they should not manage model objects or be able to present additional interfaces (like the **UIPopoverController**).

A better solution is to give **BNRItemCell** a block to execute when the button is tapped. This block will be supplied by the **BNRItemsViewController** that is responsible for configuring the cell.

Adding a block to the cell subclass

We briefly looked at blocks in Chapter 17, but let's take a closer look now.

Open **BNRItemCell.h** and add a property for the block that will be executed.

```
@interface BNRItemCell : UITableViewCell

@property (nonatomic, weak) IBOutlet UIImageView *thumbnailView;
@property (nonatomic, weak) IBOutlet UILabel *nameLabel;
@property (nonatomic, weak) IBOutlet UILabel *serialNumberLabel;
@property (nonatomic, weak) IBOutlet UILabel *valueLabel;

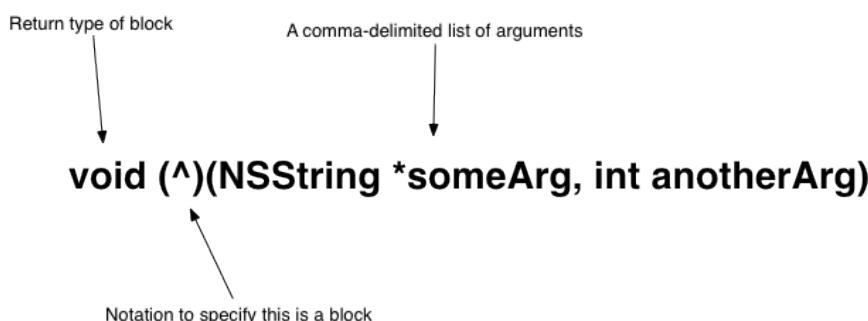
@property (nonatomic, copy) void (^actionBlock)(void);

@end
```

The syntax may still be a bit scary, but recall that a block looks a lot like a function. Figure 19.13 shows the various components of a block.

Notice that the property is declared as **copy**. This is very important. Blocks behave a bit differently than the rest of the objects you have been working with so far. When a block is created, it is created on the stack, as opposed to being created on the heap like other objects. This means that when the method that a block is declared in is returned, any blocks that are created will be destroyed along with all of the other local variables. In order for a block to persist beyond the lifetime of the method it is declared in, a block must be sent the **copy** message. By doing so, the block will be copied to the heap – thus you declare the property to have the **copy** attribute.

Figure 19.13 Block syntax



In **BNRItemCell.m**, call the block when the button is tapped.

```
- (IBAction)showImage:(id)sender
{
    if (self.actionBlock) {
        self.actionBlock();
    }
}
```

Note that you have to make sure that the block exists before calling it.

Let's verify this all works according to plan. In **BNRItemsViewController.m**, update **tableView:cellForRowAtIndexPath:** to print out the index path.

```

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    BNRItem *item = [[BNRItemStore sharedStore] allItems][indexPath.row];

    // Get the new or recycled cell
    BNRItemCell *cell =
        [tableView dequeueReusableCellWithIdentifier:@"BNRItemCell"
            forIndexPath:indexPath];

    // Configure the cell with the BNRItem
    cell.nameLabel.text = item.itemName;
    cell.serialNumberLabel.text = item.serialNumber;
    cell.valueLabel.text = [NSString stringWithFormat:@"%@", item.valueInDollars];
    cell.thumbnailView.image = item.thumbnail;

    cell.actionBlock = ^{
        NSLog(@"Going to show image for %@", item);
    };

    return cell;
}

```

Build and run the application. Tap a thumbnail (or, more accurately, the transparent button on top of the thumbnail) and check the message in the console.

Presenting the image in a popover controller

Now, **BNRItemsViewController** needs to change the action block to grab the **BNRItem** associated with the cell whose button was tapped and display its image in a **UIPopoverController**.

To display an image in a popover, you need a new **UIViewController** whose view shows an image. Create a new Objective-C subclass. Name this new class **BNRImageViewController**, select **UIViewController** as its superclass, and uncheck all boxes.

Since this view controller will only have one view, you will create it programmatically. In **BNRImageViewController.m**, implement **loadView**.

```

- (void)loadView
{
    UIImageView *imageView = [[UIImageView alloc] init];
    imageView.contentMode = UIViewContentModeScaleAspectFit;
    self.view = imageView;
}

```

Note that you do not need to set up any constraints because the **UIPopoverController** that the **BNRImageViewController** will appear in will always set the size of this image view to the size of the popover.

Now add a property to the public interface in **BNRImageViewController.h** to hold the image.

```

@interface BNRImageViewController : UIViewController

@property (nonatomic, strong) UIImage *image;

@end

```

When an instance of **BNRImageViewController** is created, it will be given an image. In **BNRImageViewController.m**, implement **viewWillAppear:** to set the view's image from this image.

```

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];

    // We must cast the view to UIImageView so the compiler knows it
    // is okay to send it setImage:
    UIImageView *imageView = (UIImageView *)self.view;
    imageView.image = self.image;
}

```

Now you can finish implementing the action block. In `BNRItemsViewController.m`, add a property to hang on to a popover controller in the class extension and have this class conform to the `UIPopoverControllerDelegate` protocol.

```

@interface BNRItemsViewController () <UIPopoverControllerDelegate>
@property (nonatomic, strong) UIPopoverController *imagePopover;
@end

```

Next, import the appropriate header files at the top of `BNRItemsViewController.m`.

```

#import "BNRImageStore.h"
#import "BNRImageViewController.h"

```

Flesh out the implementation of the action block to present the popover controller that displays the full-size image for the `BNRItem` represented by the cell that was tapped.

```

cell.actionBlock = ^{
    NSLog(@"Going to show image for %@", item);

    if ([UIDevice currentDevice].userInterfaceIdiom == UIUserInterfaceIdiomPad) {

        NSString *itemKey = item.itemKey;

        // If there is no image, we don't need to display anything
        UIImage *img = [[BNRImageStore sharedStore] imageForKey:itemKey];
        if (!img) {
            return;
        }

        // Make a rectangle for the frame of the thumbnail relative to
        // our table view
        // Note: there will be a warning on this line that we'll soon discuss
        CGRect rect = [self.view convertRect:cell.thumbnailView.bounds
                                      fromView:cell.thumbnailView];

        // Create a new BNRImageViewController and set its image
        BNRImageViewController *ivc = [[BNRImageViewController alloc] init];
        ivc.image = img;

        // Present a 600x600 popover from the rect
        self.imagePopover = [[UIPopoverController alloc]
                             initWithContentViewController:ivc];
        self.imagePopover.delegate = self;
        self.imagePopover.popoverContentSize = CGSizeMake(600, 600);
        [self.imagePopover presentPopoverFromRect:rect
                                         inView:self.view
                                         permittedArrowDirections:UIPopoverArrowDirectionAny
                                         animated:YES];
    }
};

```

Finally, in `BNRItemsViewController.m`, get rid of the popover if the user taps anywhere outside of it .

```
- (void)popoverControllerDidDismissPopover:(UIPopoverController *)popoverController
{
    self.imagePopover = nil;
}
```

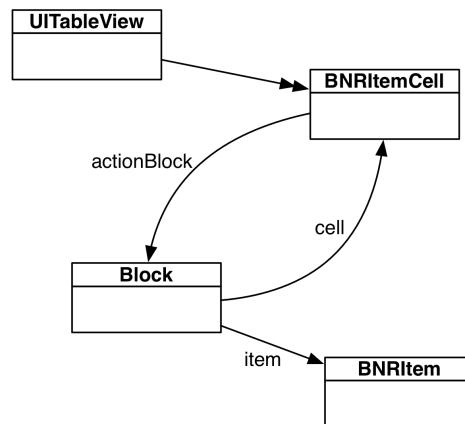
Build and run the application. Tap on the thumbnails in each row to see the full-size image in the popover. Tap anywhere else to dismiss the popover.

Variable Capturing

A block can use any variables that are visible within its *enclosing scope*. The enclosing scope of a block is the scope of the method in which it is defined. Thus, a block has access to all of the local variables of the method, arguments passed to the method, and instance variables that belong to the object running the method. In the `actionBlock` code above, both the `BNRItem` (`item`) and the `BNRItemCell` (`cell`) have been captured from the enclosing scope.

Blocks own the objects that they capture, and this can easily result in a strong reference cycle. Take a look back at the warning you had when creating `rect` within the `actionBlock`: “Capturing ‘cell’ strongly in this block is likely to lead to a strong reference cycle”. Since blocks own the objects they capture, this makes sense. The `cell` has ownership of `actionBlock`, and `actionBlock` has strong ownership of `cell` (Figure 19.14).

Figure 19.14 Cell and block own each other



To fix this problem, `actionBlock` should have a weak reference to `cell`, which will break the strong reference cycle.

In `BNRItemsViewController.m`, update the `actionBlock` code to have a weak reference to the `BNRItemCell`.

```
__weak BNRIItemCell *weakCell = cell;

cell.actionBlock = ^{
    NSLog(@"Going to show image for %@", item);

    BNRIItemCell *strongCell = weakCell;

    if ([UIDevice currentDevice].userInterfaceIdiom == UIUserInterfaceIdiomPad) {

        NSString *itemKey = item.itemKey;

        // If there is no image, we don't need to display anything
        UIImage *img = [[BNRImageStore sharedStore] imageForKey:itemKey];
        if (!img) {
            return;
        }

        // Make a rectangle that the frame of the thumbnail relative to
        // our table view
        // Note: there will be a warning on this line that we'll soon discuss
        CGRect rect = [self.view convertRect:cell.thumbnailView.bounds
                                         fromView:cell.thumbnailView];
        CGRect rect = [self.view convertRect:strongCell.thumbnailView.bounds
                                         fromView:strongCell.thumbnailView];

        // Create a new BNRIImageViewController and set its image
        BNRIImageViewController *ivc = [[BNRIImageViewController alloc] init];
        ivc.image = img;

        // Present a 600x600 popover from the rect
        self.imagePopover = [[UIPopoverController alloc]
                             initWithContentViewController:ivc];
        self.imagePopover.delegate = self;
        self.imagePopover.popoverContentSize = CGSizeMake(600, 600);
        [self.imagePopover presentPopoverFromRect:rect
                                         inView:self.view
                                         permittedArrowDirections:UIPopoverArrowDirectionAny
                                         animated:YES];
    }
};
```

Once the block begins executing, you need to guarantee that the cell hangs around until it is done executing. For that reason, you temporarily take strong ownership of that cell by creating a strong reference to it with `strongCell`. Unlike taking a permanent strong reference to a variable from the enclosing scope, this way the block only has a strong reference to the cell object as long as the `strongCell` variable exists – that is, while the block is actually executing.

Build and run the application. The behavior will be the same, but your application no longer has a memory leak.

Bronze Challenge: Color Coding

If a `BNRItem` is worth more than \$50, make its value label text appear in green. If it is worth less than \$50, make it appear in red.

Gold Challenge: Zooming

The **BNRImageViewController** should center its image and allow zooming. Implement this behavior in **BNRImageViewController.m**.

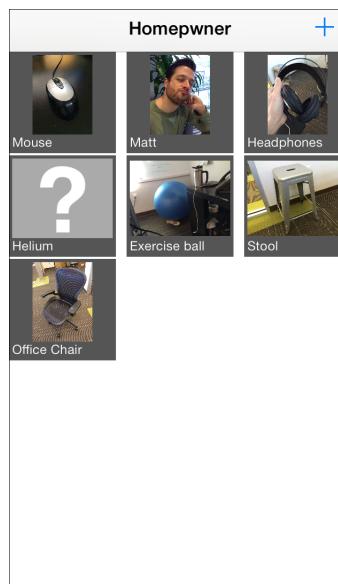
For the More Curious: UICollectionView

The class **UICollectionView** is very similar to **UITableView**:

- It is a subclass of **UIScrollView**.
- It displays cells, although these cells inherit from **UICollectionViewCell** instead of **UITableViewCell**.
- It has a data source that supplies it with those cells.
- It has a delegate that gets informed about things like a cell being selected.
- Similar to **UIViewController**, **UICollectionViewController** is a view controller class that creates a **UICollectionView** as its view and becomes its delegate and data source.

How is the collection view different? A table view only displays one column of cells; this is a huge limitation on a large-screen device like an iPad. A collection view can layout those cells any way you want. The most common layout is a grid (Figure 19.15).

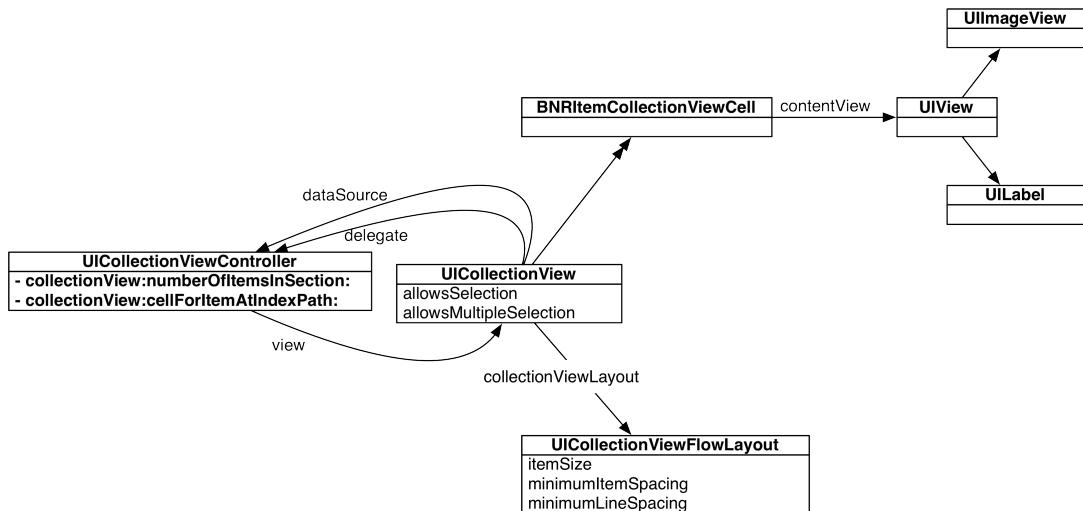
Figure 19.15 Homeowner with a **UICollectionView**



How does the **UICollectionView** figure out how to arrange the cells? It has a layout object that controls the attributes of each cell, including its position and size. These layout objects inherit from the abstract class **UICollectionViewLayout**. If you are just laying your cells out in a grid, you can use an

instance of **UICollectionViewFlowLayout**. If you are doing something fancy, you will need to create a custom subclass of **UICollectionViewLayout**.

Figure 19.16 Example object model for a **UICollectionView**



The default **UITableViewCell** is quite usable. (You used it for several chapters before this one.) **UICollectionViewCell** is not. It has a content view, but the content view has no subviews. So, if you are creating a **UICollectionView**, you will need to create a subclass of **UICollectionViewCell**.

That is all you need to know to get your first collection view up and running. After that, you will want to play around with the background view, supplementary views (which are mostly used as headers and footers for sections), and decoration views. The cell also has its own background view and a selected background view (which is laid over the background view when the cell is selected).

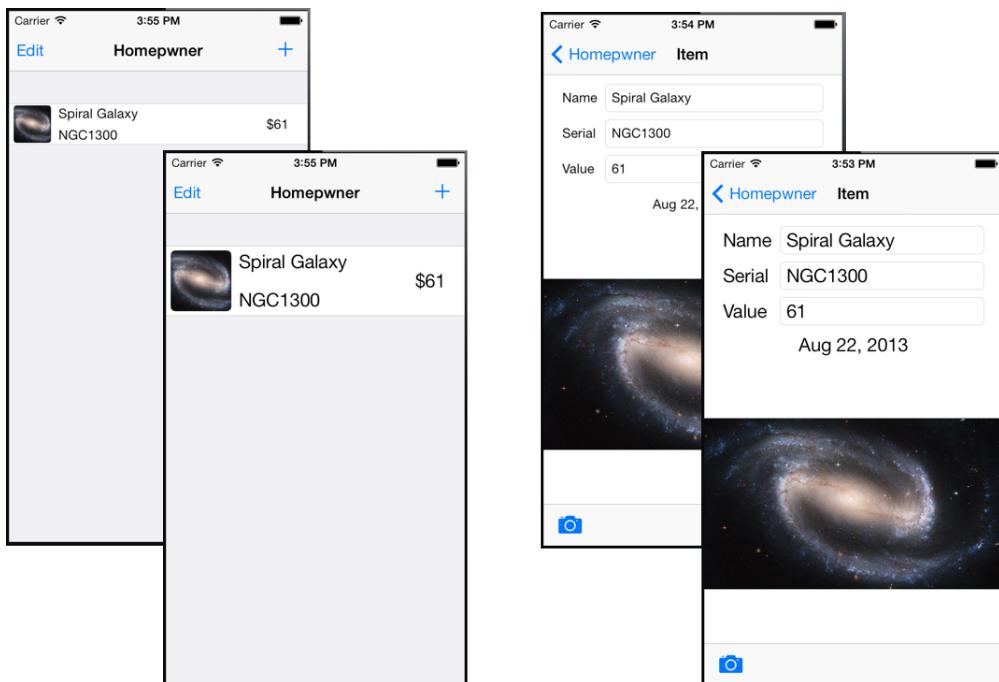
20

Dynamic Type

Creating an interface that appeals to everyone can be daunting. Some people prefer more compact interfaces so they can see more information at a time. Others might want to be able to easily see information at a glance, or perhaps they have poor eyesight. These people have different needs, and good developers strive to make apps that meet those needs.

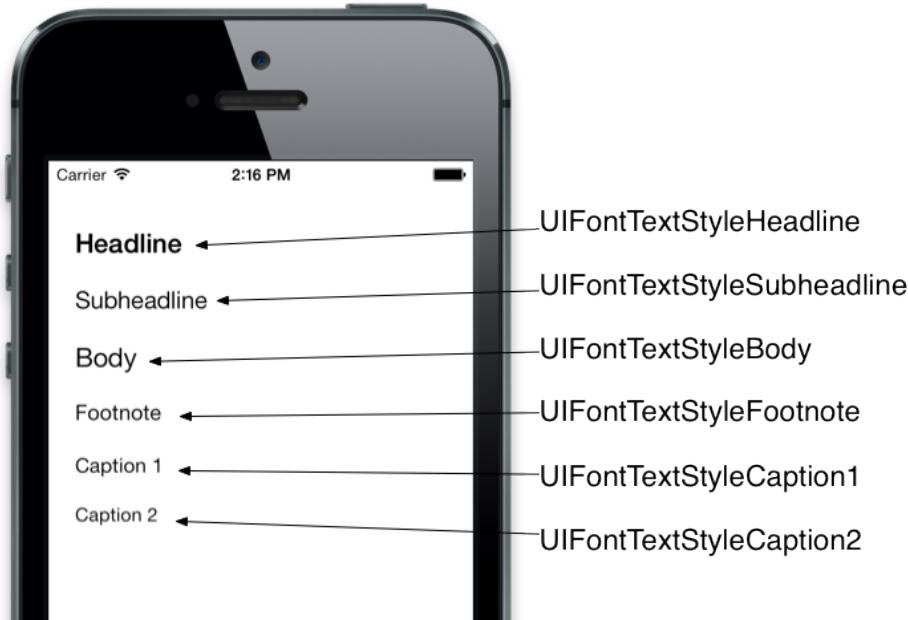
Dynamic Type is a technology introduced in iOS 7 that helps realize this goal by providing specifically designed *text styles* that are optimized for legibility. Perhaps more importantly, users can select one of seven different preferred text sizes from within Apple's **Settings** application, and apps that support Dynamic Type will have their fonts scaled appropriately. In this chapter, you will update **Homepwner** to support Dynamic Type. Figure 20.1 shows the application rendered at the smallest and largest user selectable Dynamic Type sizes.

Figure 20.1 Homepwner with Dynamic Type supported



The Dynamic Type system is centered around *text styles*. When a font is requested for a given text style, the system will use the user's preferred text size in association with the text style to return an appropriately configured font. Figure 20.2 shows the six different text styles.

Figure 20.2 Different text styles



Using Preferred Fonts

Implementing Dynamic Type is straightforward. At its most basic level, you get a `UIFont` for a specific text style and then apply that font to something that displays text, such as a `UILabel`. Let's start by updating `BNRDetailViewController`.

You are going to need to update some attributes of the labels programmatically soon, so add outlets to each of the labels to the class extension in `BNRDetailViewController.m`.

```

@interface BNRDetailViewController ()
@property (nonatomic, strong) UIPopoverController *imagePickerPopover;
@property (weak, nonatomic) IBOutlet UITextField *nameField;
@property (weak, nonatomic) IBOutlet UITextField *serialNumberField;
@property (weak, nonatomic) IBOutlet UITextField *valueField;
@property (weak, nonatomic) IBOutlet UILabel *dateLabel;
@property (weak, nonatomic) IBOutlet UIImageView *imageView;
@property (weak, nonatomic) IBOutlet UIToolbar *toolbar;
@property (weak, nonatomic) IBOutlet UIBarButtonItem *cameraButton;

@property (weak, nonatomic) IBOutlet UILabel *nameLabel;
@property (weak, nonatomic) IBOutlet UILabel *serialNumberLabel;
@property (weak, nonatomic) IBOutlet UILabel *valueLabel;
@end

```

Now that there is an outlet to each of the labels, add a method that sets the font for each to use the preferred Body style.

```

- (void)updateFonts
{
    UIFont *font = [UIFont preferredFontForTextStyle:UIFontTextStyleBody];

    self.nameLabel.font = font;
    self.serialNumberLabel.font = font;
    self.valueLabel.font = font;
    self.dateLabel.font = font;

    self.nameField.font = font;
    self.serialNumberField.font = font;
    self.valueField.font = font;
}

```

Now call this method at the end of **viewWillAppear:** to update the labels before they are visible.

```

self.imageView.image = imageToDisplay;

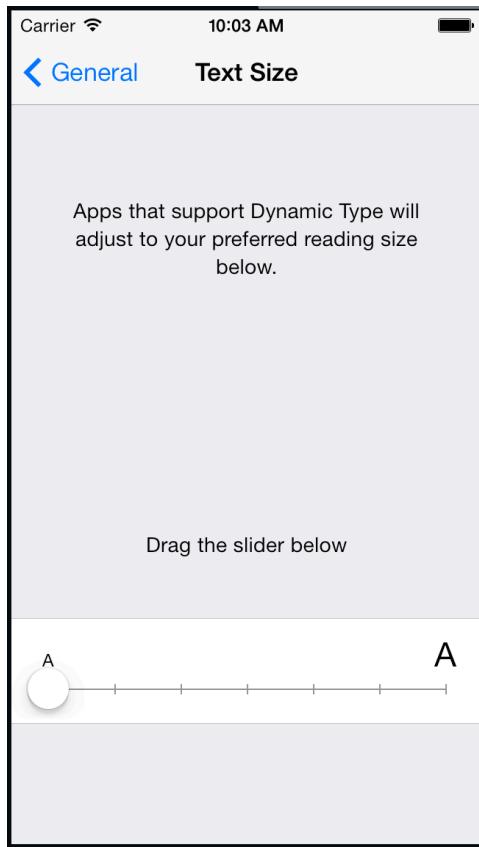
[self updateFonts];
}

```

The **preferredFontForTextStyle:** method will return a preconfigured font that is customized for the user's preferences. Build and run the application, and you will notice that the interface looks largely the same.

Now let's change the preferred font size. Press the Home button (or use Home from the Hardware menu), and open Apple's Settings application. Under General, select Text Size, and then drag the slider all the way to the left to set the font size to the smallest value (Figure 20.3).

Figure 20.3 Text size settings



Now, go back into Homeowner. If you return to the `BNRDetailViewController`, you will notice that the interface has not changed at all! Why is this? Since `viewWillAppear:` is not called when the application returns from the background, your interface is not getting updated. Luckily, you can be informed when the user changes the preferred font size.

Responding to User Changes

When the user changes the preferred text size, a notification gets posted that the application's objects can register to listen for. This is the `UIContentSizeCategoryDidChangeNotification`, and this is a great time to update the user interface.

In `BNRDetailViewController.m`, register for this notification in `initForNewItem:` and remove the class as an observer in `dealloc`.

```

        self.navigationItem.leftBarButtonItem = cancelItem;
    }

    // Make sure this is NOT in the if (isNew) { } block of code
    NSNotificationCenter *defaultCenter = [NSNotificationCenter defaultCenter];
    [defaultCenter addObserver:self
                      selector:@selector(updateFonts)
                        name:UIContentSizeCategoryDidChangeNotification
                      object:nil];
}

return self;
}

- (void)dealloc
{
    NSNotificationCenter *defaultCenter = [NSNotificationCenter defaultCenter];
    [defaultCenter removeObserver:self];
}

```

Notice that the method getting called from the notification center is the same method you implemented earlier that also gets called on `viewWillAppear:`. Build and run the application again, and the interface will update as you change the Dynamic Type preferred text size in Settings. The labels and text fields in `BNRDetailViewController` will now scale elegantly to the user's preferences.

Is that it? Well, mostly. Now that your interface can grow and shrink dynamically, you need to revisit your Auto Layout constraints.

Updating Auto Layout

When the constraints were set up for the Name, Serial Number, and Value labels, you pinned the width and the height. This worked well when the text was a fixed font and text size, but now that you have introduced Dynamic Type, your interface does not scale. If the user selects a small text size, there will be a lot of empty whitespace; if the user selects a very large text size, there is a chance that the text could be clipped. What you need to do is utilize the `intrinsicContentSize` of the labels to allow them to resize themselves to exactly the size they need to be.

Open `BNRDetailViewController.xib`. In the canvas, select each of the four labels, one by one, and remove their explicit width and height constraints. If you have any misplaced views, select the `UIControl` on the canvas and then Update All Frames in Control from the Resolve Auto Layout Issues menu.

If you look closely, or if you temporarily change the text of one of the labels to be a bit shorter or longer, you may notice that the text fields no longer line up. It would look better if they lined up, but before we investigate a fix you should understand how Auto Layout is computing all of the frames.

Content Hugging and Compression Resistance Priorities revisited

Recall that every view has a preferred size, which is its `intrinsicContentSize`. This gives each view an implied width and height constraint. For a view to grow larger than its `intrinsicContentSize` in a given dimension, there has to be a constraint with a higher priority than that view's Content Hugging

Priority. For a view to grow smaller than its `intrinsicContentSize` in a given dimension, there has to be a constraint with a higher priority than that view's Content Compression Resistance Priority.

This is important to remember when you are determining how your interfaces will get laid out. Let's inspect the layout of the Name label and corresponding text field in the horizontal direction. The constraints affecting these views are:

- `nameLabel.leadingAnchor = superview.leadingAnchor + 8`
- `nameField.leadingAnchor = nameLabel.trailingAnchor + 8`
- `nameField.trailingAnchor = superview.trailingAnchor - 8`

This gives us a visual format string that looks like:

H: | -8- [nameLabel] -8- [nameField] -8- |

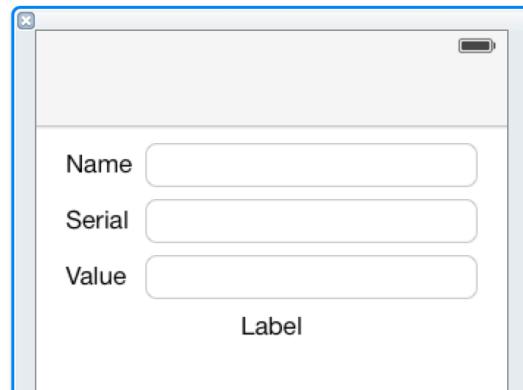
Notice that there are no constraints directly impacting the widths of the views. Because of this, both views want to be at the width of their `intrinsicContentSize`. One or both of the labels will have to stretch in order to satisfy the existing constraints.

So which view will get stretched? Since both views want to be wider than their `intrinsicContentSize`, the view with the lower Content Hugging Priority will stretch. If you compare the **UILabel** and the **UITextField**, you will see that the label has a Content Hugging Priority of 251 whereas the text field's is 250. Since the label wants to "hug" more, the label will be the width of its `intrinsicContentSize` and the text field will stretch enough to satisfy the equations.

Remember that the goal is to have all of the text fields aligned. The way that you will accomplish this is by having the three top labels be the same width. This may sound like what you just removed, but there is a subtle difference: earlier, each of the labels independently had their width (and height) pinned, but now they will instead always have equal widths.

Select the Name, Serial, and Value labels together and open the Pin menu. Select Equal Widths and from the Update Frames drop-down choose All Frames in Container. Finally, click Add 2 Constraints. Your interface should look like Figure 20.4.

Figure 20.4 Equal width constraints added



That was easy to achieve, but you may now be wondering how this is all working. Let's take a detailed look, recapping some earlier information along the way. First, the Content Hugging Priority of each label (251) is higher than that of its corresponding text field (250), so the label will hug before the text field will. Finally, you just added equal width constraints for the three top labels.

So each of the top three labels has two constraints impacting its width: a required (priority 1000) Equal Widths constraint, and the two implicit constraints that try to keep the label at its `intrinsicContentSize`. (Content Hugging with a priority of 251, and Content Compression Resistance with a priority of 750). Each of the views wants to satisfy all of these constraints. The only view that will be able to do this, however, is the view that has the maximum width. The other two views will have a required constraint with a higher priority than their Content Hugging Priority, so they will stretch to be the same width as the longest label.

What has happened here is very important. You have created an interface that scales beautifully as the text content changes. Text changes can be due to a number of different reasons. Most commonly, this will be due to the use of Dynamic Type or localizing the application for different languages (which we will discuss in Chapter 25).

The interface for `BNRDetailViewController` is done. Go ahead and test the `BNRDetailViewController` at different Dynamic Type sizes, and you should notice that the interface scales appropriately.

Determining the User's Preferred Text Size

Now it is time to turn our attention to the `BNRItemsViewController`. You will need to update two parts of this view controller for Dynamic Type: the rows of your table view will grow or shrink in response to the user changing the preferred text size, and the `BNRItemCell` will need to be updated similarly to how you updated the `BNRDetailViewController`. Let's begin by updating the table view row height.

The goal is to have the table view row heights reflect the preferred Dynamic Type text size of the user. If the user chooses a larger text size, the rows will be taller in order to accommodate the text. Since this is not a problem that Auto Layout will solve, the row heights will need to be set manually. To do this, you need a way of determining which text size the user has selected.

`UIApplication` exposes the text size that user selected through its `preferredContentSizeCategory` property. The method will return a constant `NSString` with the name of the content size category, which will be one of the following values:

- `UIContentSizeCategoryExtraSmall`
- `UIContentSizeCategorySmall`
- `UIContentSizeCategoryMedium`
- `UIContentSizeCategoryLarge` (Default)
- `UIContentSizeCategoryExtraLarge`
- `UIContentSizeCategoryExtraExtraLarge`
- `UIContentSizeCategoryExtraExtraExtraLarge`

Open `BNRItemsViewController.m`. Create a method that will update the table view row height based on the user-selected text size and call this method in `viewWillAppear:`.

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    [self.tableView reloadData];
    [self updateTableViewForDynamicTypeSize];
}

- (void)updateTableViewForDynamicTypeSize
{
    static NSDictionary *cellHeightDictionary;

    if (!cellHeightDictionary) {
        cellHeightDictionary = @{@"UIContentSizeCategoryExtraSmall" : @44,
                               @"UIContentSizeCategorySmall" : @44,
                               @"UIContentSizeCategoryMedium" : @44,
                               @"UIContentSizeCategoryLarge" : @44,
                               @"UIContentSizeCategoryExtraLarge" : @55,
                               @"UIContentSizeCategoryExtraExtraLarge" : @65,
                               @"UIContentSizeCategoryExtraExtraExtraLarge" : @75 };
    }

    NSString *userSize =
        [[UIApplication sharedApplication] preferredContentSizeCategory];

    NSNumber *cellHeight = cellHeightDictionary[userSize];
    [self.tableView setRowHeight:cellHeight.floatValue];
    [self.tableView reloadData];
}
```

Build and run the application. If you change the Dynamic Type preferred text size and restart the application again, you should notice that the table view row heights reflects the user's selected text size. (If you don't restart the application from Xcode, you'll need to go to the `BNRDetailViewController` and then return to the `BNRItemsViewController`.)

Just as you did with the `BNRDetailViewController` earlier, you need to have the `BNRItemsViewController` register itself as an observer for the `UIContentSizeCategoryDidChangeNotification`.

In `BNRItemsViewController.m`, register for the notification in `init`, and remove the view controller as an observer in `dealloc`. Finally, implement the notification call back to call the `updateTableViewForDynamicTypeSize` method that you just created.

```

self.navigationItem.rightBarButtonItem = bbi;
self.navigationItem.leftBarButtonItem = [self editButtonItem];

NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
[nc addObserver:self
    selector:@selector(updateTableViewForDynamicTypeSize)
    name:UIContentSizeCategoryDidChangeNotification
    object:nil];
}

return self;
}

- (void)dealloc
{
    NotificationCenter *nc = [NotificationCenter defaultCenter];
    [nc removeObserver:self];
}

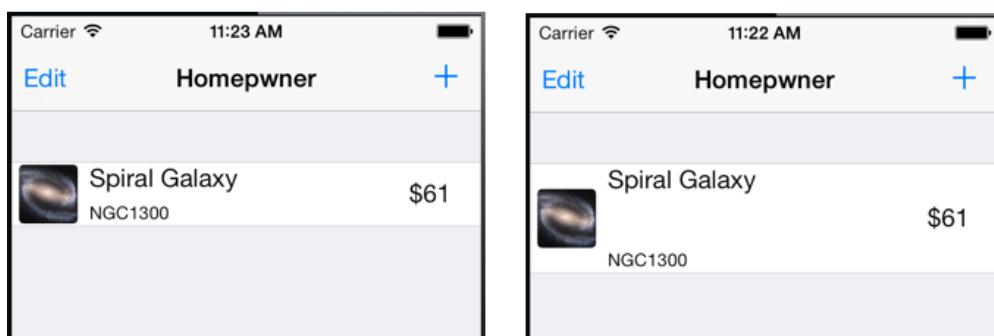
```

The table view row height should update now as you change the preferred Dynamic Type text size. Build and run the application and test this out.

Updating BNRItemCell

Although the row heights adjust based on the preferred text size, the cell content is not adjusting. You will add Dynamic Type support to **BNRItemCell** shortly. First, pay attention to how the cell subviews adjust appropriately based on the cell height. Because of your thoughtful use of Auto Layout when creating **BNRItemCell**, the interface adjusts elegantly (Figure 20.5). The `imageView` stays vertically centered in the cell and pinned to the left, `nameLabel` is pinned to the top, `serialNumberLabel` is pinned to the bottom, and `valueLabel` stays centered and pinned to the right.

Figure 20.5 Auto Layout in action



This worked great while the text was a fixed size, but when you update the class to use Dynamic Type you will need to make some changes. Let's begin by implementing the Dynamic Type code to update the labels, which will closely follow what you did with **BNRDetailViewController** and **BNRItemsViewController**.

Open `BNRItemCell.m` and make the following changes:

```
- (void)updateInterfaceForDynamicTypeSize
{
    UIFont *font = [UIFont preferredFontForTextStyle:UIFontTextStyleBody];
    self.nameLabel.font = font;
    self.serialNumberLabel.font = font;
    self.valueLabel.font = font;
}

- (void)awakeFromNib
{
    [self updateInterfaceForDynamicTypeSize];

    NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
    [nc addObserver:self
        selector:@selector(updateInterfaceForDynamicTypeSize)
        name:UIContentSizeCategoryDidChangeNotification
        object:nil];
}

- (void)dealloc
{
    NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
    [nc removeObserver:self];
}
```

The only new piece of information here is the `awakeFromNib` method. This is called on an object after it has been unarchived from a NIB file, and is a great place to do any additional UI work that cannot be done within the XIB file. You can do any additional configuration for the cell that cannot be done within the XIB file within this method. Build and run the application, and the text will update to reflect the user's preferred text size.

Now there are a few Auto Layout issues that you need to resolve.

Initially, you pinned the height of `nameLabel` and `serialNumberLabel`. This does not work well now that your text sizes can dynamically change.

In `BNRItemCell.xib`, delete the two height constraints from those two labels. If you have any misplaced views now, open the Resolve Auto Layout Issues menu and select Update All Frames in Homeowner Item Cell. Build and run the application and the label heights will be based on the preferred text size.

The interface is looking fantastic now, but wouldn't it be great if the size of `imageView` reflected the preferred font size? It makes sense that the image should scale with the text size. Let's do this.

Constraint outlets

To update the position or size of a view, either in absolute terms or relative to another view, you should update the constraints on that view. This is very important! If you modify the `frame` (or `bounds`), instead of the constraints, the next time the view needs to be laid out, it will be laid out based on the constraints that it has. In other words, the changes made to the `frame` will not persist.

In order to change the width and height of the image view, the constants on the respective constraints will need to be updated at run time. To do this, you will need to create an outlet to both the vertical and horizontal constraints. Constraints are objects (`NSLayoutConstraint`), so just like you can create outlets to views, the same can be done with constraints.

In `BNRItemCell.m`, create and connect the outlet for these two constraints to the class extension. When you are done, the class extension should look like this:

```
@interface BNRItemCell ()
```

```
@property (nonatomic, weak) IBOutlet NSLayoutConstraint *imageViewHeightConstraint;
@property (nonatomic, weak) IBOutlet NSLayoutConstraint *imageViewWidthConstraint;
```

```
@end
```

With outlets to the size constraints of `imageView` created, you can now adjust `imageView`'s size programmatically. In `BNRItemCell.m`, modify `updateInterfaceForDynamicTypeSize` to get the currently selected preferred text size, and use that to adjust the size of `imageView`.

```
- (void)updateInterfaceForDynamicTypeSize
{
    UIFont *font = [UIFont preferredFontForTextStyle:UIFontTextStyleBody];
    self.nameLabel.font = font;
    self.serialNumberLabel.font = font;
    self.valueLabel.font = font;

    static NSDictionary *imageSizeDictionary;

    if (!imageSizeDictionary) {
        imageSizeDictionary = @{@"UIContentSizeCategoryExtraSmall" : @40,
                               @"UIContentSizeCategorySmall" : @40,
                               @"UIContentSizeCategoryMedium" : @40,
                               @"UIContentSizeCategoryLarge" : @40,
                               @"UIContentSizeCategoryExtraLarge" : @45,
                               @"UIContentSizeCategoryExtraExtraLarge" : @55,
                               @"UIContentSizeCategoryExtraExtraExtraLarge" : @65 };
    }

    NSString *userSize =
        [[UIApplication sharedApplication] preferredContentSizeCategory];

    NSNumber *imageSize = imageSizeDictionary[userSize];
    self.imageViewHeightConstraint.constant = imageSize.floatValue;
    self.imageViewWidthConstraint.constant = imageSize.floatValue;
}
```

Build and run the application, and play with the Dynamic Type text sizes. The `imageView` adjusts its size appropriately. Also notice that because you pinned the leading edge of `nameLabel` and `serialNumberLabel` to the trailing edge of the `imageView`, the interface scales very well as the preferred text size changes. Had those two labels been pinned to their superview's leading edge instead, the `imageView` would have overlapped the labels.

The interface is great, but let's make one final change.

Placeholder constraints

Currently, you are updating both the width and height constraints for the `imageView`. This is not a problem, but you can do better. Instead of updating both constraints, you will add one additional constraint to `imageView` that will constrain the `imageView`'s width and height to be equal.

You cannot create this constraint in Interface Builder, so return to `BNRItemCell.m` and create this constraint programmatically in `awakeFromNib`.

```
- (void)awakeFromNib
{
    [self updateInterfaceForDynamicTypeSize];

    NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
    [nc addObserver:self
        selector:@selector(updateInterfaceForDynamicTypeSize)
        name:UIContentSizeCategoryDidChangeNotification
        object:nil];

    NSLayoutConstraint *constraint =
        [NSLayoutConstraint constraintWithItem:self.thumbnailView
            attribute:NSLayoutAttributeHeight
            relatedBy:NSLayoutRelationEqual
            toItem:self.thumbnailView
            attribute:NSLayoutAttributeWidth
            multiplier:1
            constant:0];
    [self.thumbnailView addConstraint:constraint];
}
```

Now, remove the `imageViewWidthConstraint` property and corresponding code.

```
@interface BNRItemCell ()

@property (nonatomic, weak) IBOutlet NSLayoutConstraint *imageViewHeightConstraint;
@property (nonatomic, weak) IBOutlet NSLayoutConstraint *imageViewWidthConstraint;

@end

@implementation

- (void)updateInterfaceForDynamicTypeSize
{
    // Other code here

    NSNumber *imageSize = imageSizeDictionary[userSize];
    self.imageViewHeightConstraint.constant = imageSize.floatValue;
    self.imageViewWidthConstraint.constant = imageSize.floatValue;
}
```

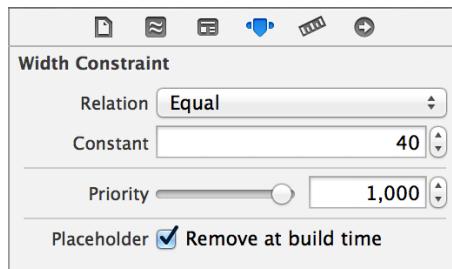
Open `BNRItemCell.xib` and make sure that the outlet for `imageViewWidthConstraint` is removed, since the outlet no longer exists.

There is one final change that must be made. There are now two constraints affecting the width of the `imageView`: the programmatic constraint you just created and the explicit width constraint in the XIB file. This will create unsatisfiable (conflicting) constraints if the two constraints do not agree on a size.

To fix this, you could delete the explicit width constraint from the `imageView`. This will work, but Interface Builder might warn about misplaced views or an ambiguous layout. Instead, you can make the width constraint a *placeholder constraint*. Placeholder constraints, as the name implies, are only temporary and are removed at build time, so they will not exist when the application is running.

In `BNRItemCell.xib`, select the width constraint on the `imageView`, and open the attributes inspector. Check the Placeholder box that says Remove at build time (Figure 20.6). Build and run the application, and everything will work just as it did before. Homepwner now scales appropriately with the user's preferred text size.

Figure 20.6 Placeholder constraints



This page intentionally left blank

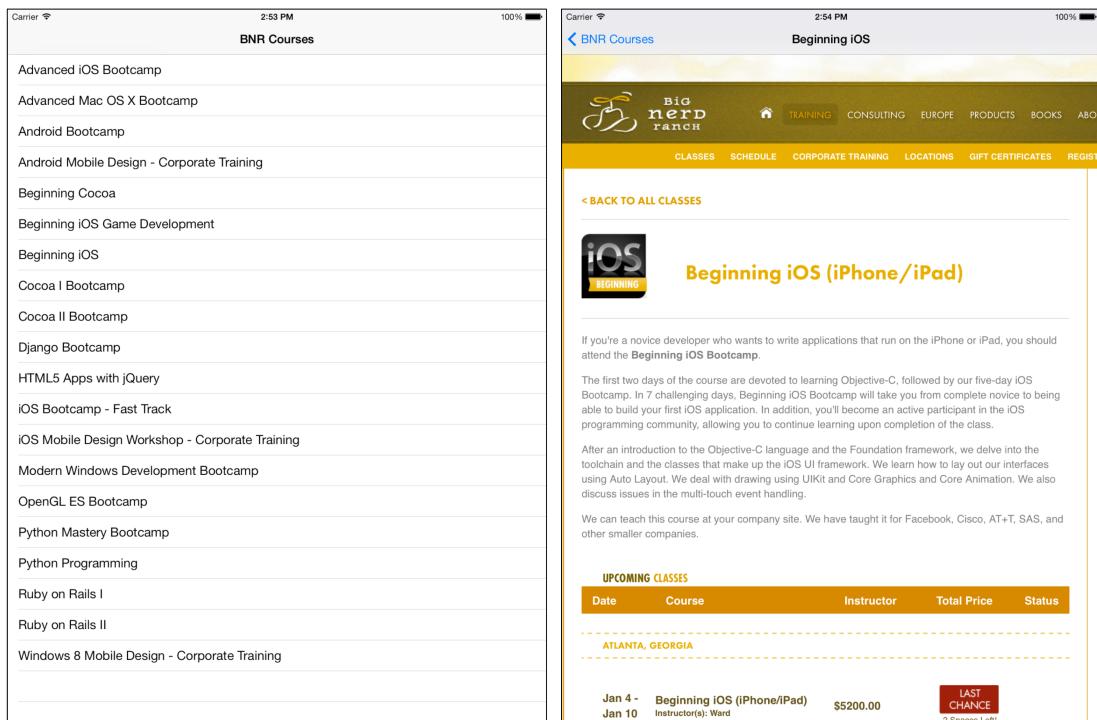
21

Web Services and UIWebView

For the next two chapters, you are going to take another break from Homepwner to work with web services and split view controllers.

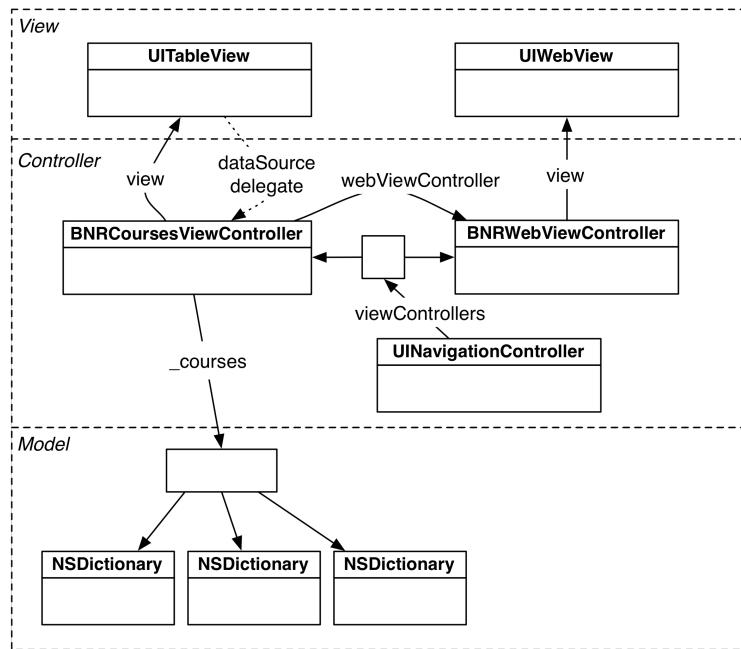
In this chapter, you will lay the foundation for an application named Nerdfeed that reads in a list of the courses that Big Nerd Ranch offers. Each course will be listed in a table view, and selecting a course will open that course's web page. Figure 21.1 shows Nerdfeed at the end of this chapter.

Figure 21.1 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 **UIWebView** class to display web content. Figure 21.2 shows an object diagram for Nerdfeed.

Figure 21.2 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.

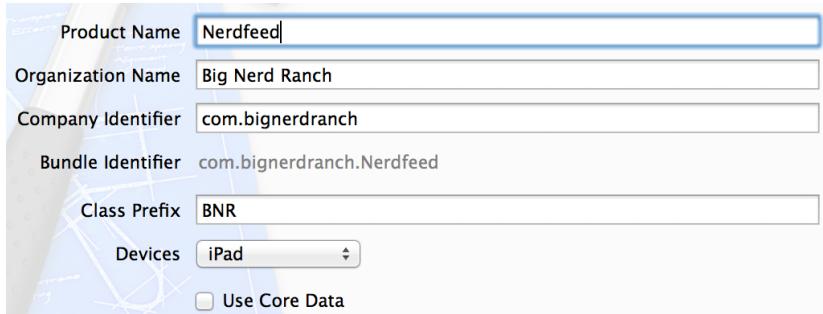
Because the HTTP protocol does not care what data it transports, these exchanges can contain complex data. This data is typically in JSON (JavaScript Object Notation) or XML format. If you control the web server as well as the client, you can use any format you like; if not, you have to build your application 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 Empty Application for the iPad Device Family. Name this application Nerdfeed, as shown in Figure 21.3. (If you do not have an iPad to deploy to, use the iPad simulator.)

Figure 21.3 Creating an iPad Empty Application



Let's knock out the basic UI before focusing on web services. Create a new **NSObject** subclass and name it **BNRCoursesViewController**. In **BNRCoursesViewController.h**, change the superclass to **UITableViewController**.

```
@interface BNRCoursesViewController : NSObject
@interface BNRCoursesViewController : UITableViewController
```

In **BNRCoursesViewController.m**, write stubs for the required data source methods so that you can build and run as you go through this exercise.

```
- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return 0;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    return nil;
}
```

In **BNRAppDelegate.m**, create an instance of **BNRCoursesViewController** and set it as the root view controller of a navigation controller. Make that navigation controller the root view controller of the window.

```
#import "BNRAppDelegate.h"
#import "BNRCoursesViewController.h"

@implementation BNRAppDelegate

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];

    BNRCoursesViewController *cvc =
        [[BNRCoursesViewController alloc] initWithStyle:UITableViewStylePlain];

    UINavigationController *masterNav =
        [[UINavigationController alloc] initWithRootViewController:cvc];

    self.window.rootViewController = masterNav;

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];

    return YES;
}


```

Build and run the application. You should see an empty **UITableView** and a navigation bar.

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 21.4).

Figure 21.4 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.

```
NSString *search = @"Play some \"Abba\"";  
NSString *escaped =  
    [search stringByAddingPercentEscapesUsingEncoding:NSUTF8StringEncoding];  
  
// escaped is now "Play%20some%20%22Abba%22"
```

If you need to un-escape a percent-escaped string, **NSString** has the method:

```
- (NSString *)stringByRemovingPercentEncoding;
```

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 **BNRCoursesViewController**, 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 `BNRCoursesViewController.m` add a property to the class extension to hold onto an instance of **NSURLSession**.

```
@interface BNRCoursesViewController ()  
  
@property (nonatomic) NSURLSession *session;  
  
@end
```

Then override `initWithStyle:` to create the **NSURLSession** object.

```
- (instancetype)initWithStyle:(UITableViewStyle)style  
{  
    self = [super initWithStyle:style];  
    if (self) {  
        self.navigationItem.title = @"BNR Courses";  
  
        NSURLSessionConfiguration *config =  
            [NSURLSessionConfiguration defaultSessionConfiguration];  
        _session = [NSURLSession sessionWithConfiguration:config  
                                         delegate:nil  
                                         delegateQueue:nil];  
    }  
    return self;  
}
```

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 `BNRCoursesViewController.m`, 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.

```
- (void)fetchFeed  
{  
    NSString *requestString = @"http://bookapi.bignerdranch.com/courses.json";  
    NSURL *url = [NSURL URLWithString:requestString];  
    NSURLRequest *req = [NSURLRequest requestWithURL:url];  
  
    NSURLSessionDataTask *dataTask =  
        [self.session dataTaskWithRequest:req  
                           completionHandler:  
                               ^(NSData *data, NSURLResponse *response, NSError *error) {  
                                   NSString *json = [[NSString alloc] initWithData:data  
                                                       encoding:NSUTF8StringEncoding];  
                                   NSLog(@"%@", json);  
                               }];  
    [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.

A project may have multiple instances of **NSURLSession**, but since Nerdfeed only initiates a single, simple request, it will use the **sharedSession**. The **sharedSession** is set up with a default configuration.

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 **BNRCoursesViewController** is created. In **BNRCoursesViewController.m**, update **initWithStyle:**:

```
- (instancetype)initWithStyle:(UITableViewStyle)style
{
    self = [super initWithStyle:style];
    if (self) {
        self.navigationItem.title = @"BNR Courses";

        NSURLSessionConfiguration *config =
            [NSURLSessionConfiguration defaultSessionConfiguration];
        _session = [NSURLSession sessionWithConfiguration:config
                                                delegate:nil
                                           delegateQueue:nil];
        [self fetchFeed];
    }
    return self;
}
```

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`).

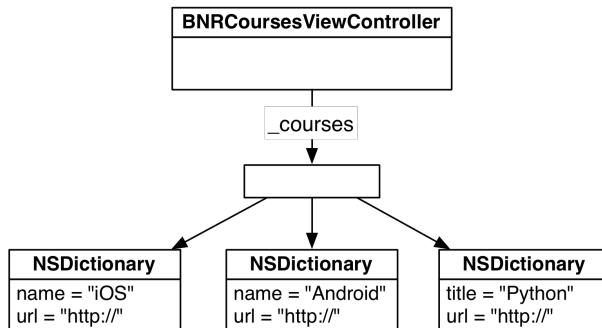
A string is represented by using text within quotations. Strings are used as the keys within a dictionary and can be used as values, too. Thus, the value of the `name` key in the top-level dictionary is the string `Christian`.

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 21.5 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 `NSDictionary` for every JSON object, `NSArray` for every JSON array, `NSString` for every JSON string, and `NSNumber` for every JSON number.

In `BNRCoursesViewController.m`, modify the `NSURLSessionDataTask` completion handler to use the `NSJSONSerialization` class to convert the raw JSON data into the basic foundation objects.

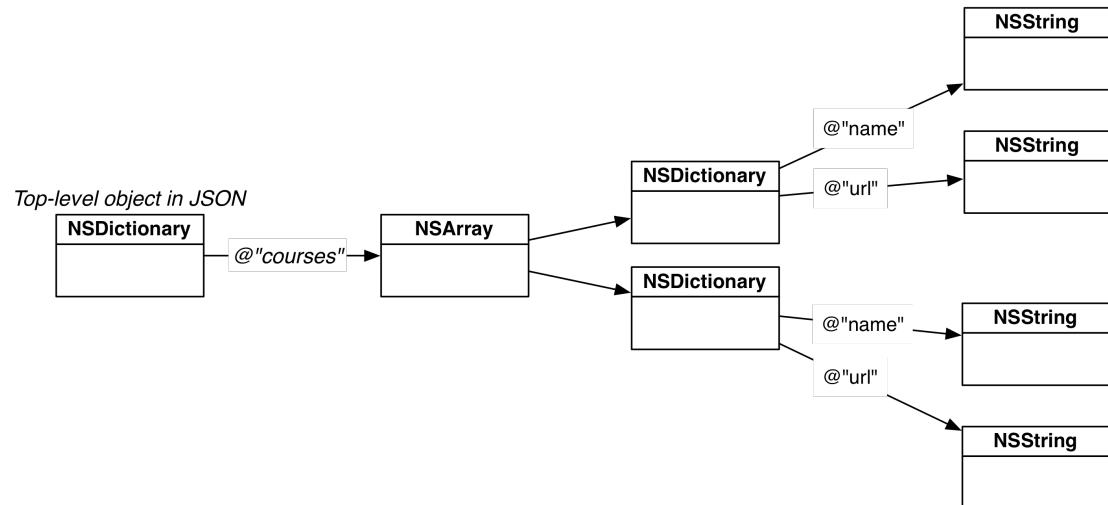
```
^(NSData *data, NSURLResponse *response, NSError *error) {
    NSString *json = [[NSString alloc] initWithData:data
                                                encoding:NSUTF8StringEncoding];
    NSLog(@"%@", json);

    NSDictionary *jsonObject = [NSJSONSerialization JSONObjectWithData:data
                                                               options:0
                                                                 error:nil];
    NSLog(@"%@", jsonObject);
}
```

Build and run, then check the console. You will see the JSON data, except now it will be formatted slightly differently because `NSLog` does a good job formatting dictionaries and arrays. The `jsonObject` is an instance of `NSDictionary` and it has an `NSString` key with an associated value of type `NSArray`.

When the `NSURLSessionDataTask` finishes, you will use `NSJSONSerialization` to convert the JSON data into an `NSDictionary`. Figure 21.6 shows how the data will be structured.

Figure 21.6 JSON objects



In `BNRCoursesViewController.m`, add a new property to the class extension to hang on to that array, which is an array of `NSDictionary` objects that describe each course.

```
@interface BNRCoursesViewController ()

@property (nonatomic) NSURLSession *session;
@property (nonatomic, copy) NSArray *courses;

@end
```

Then, in the same file, change the implementation of the **NSURLSessionDataTask** completion handler:

```
^(NSData *data, NSURLResponse *response, NSError *error) {
    NSDictionary *jsonObject = [NSJSONSerialization JSONObjectWithData:data
        options:0
        error:nil];
    NSLog(@"%@", jsonObject);
    self.courses = jsonObject[@"courses"];
    NSLog(@"%@", self.courses);
}
```

Now, still in `BNRCoursesViewController.m`, update the data source methods so that each of the course titles are shown in the table. You will also want to override **viewDidLoad** to register the table view cell class.

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    [self.tableView registerClass:[UITableViewCell class]
        forCellReuseIdentifier:@"UITableViewCell"];
}

- (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
{
    return 0;
    return [self.courses count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    return nil;
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:@"UITableViewCell"
            forIndexPath:indexPath];

    NSDictionary *course = self.courses[indexPath.row];
    cell.textLabel.text = course[@"title"];

    return cell;
}
```

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 **dispatch_async** function.

In **BNRCoursesViewController.m**, update the completion handler to reload the table view data on the main thread.

```
^(NSData *data, NSURLResponse *response, NSError *error) {
    NSDictionary *jsonObject = [NSJSONSerialization JSONObjectWithData:data
        options:0
        error:nil];
    self.courses = jsonObject[@"courses"];
    NSLog(@"%@", self.courses);

    dispatch_async(dispatch_get_main_queue(), ^{
        [self.tableView reloadData];
    });
}
```

Build and run the application. After the web service completes, you should see a list of Big Nerd Ranch's courses.

UIWebView

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 **UIWebView**.

Instances of **UIWebView** render web content. In fact, the Safari application on your device uses a **UIWebView** to render its web content. In this part of the chapter, you will create a view controller whose view is an instance of **UIWebView**. 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 **NSDictionary**.

Create a new **NSObject** subclass and name it **BNRWebViewController**. In **BNRWebViewController.h**, add a property and change the superclass to **UIViewController**:

```
@interface BNRWebViewController : NSObject
@interface BNRWebViewController : UIViewController

@property (nonatomic) NSURL *URL;

@end
```

In `BNRWebViewController.m`, write the following implementation.

```
@implementation BNRWebViewController

- (void)loadView
{
    UIWebView *webView = [[UIWebView alloc] init];
    webView.scalesPageToFit = YES;
    self.view = webView;
}

- (void)setURL:(NSURL *)URL
{
    _URL = URL;
    if (_URL) {
        NSURLRequest *req = [NSURLRequest requestWithURL:_URL];
        [(UIWebView *)self.view loadRequest:req];
    }
}

@end
```

In `BNRCoursesViewController.h`, add a new property to hang on to an instance of **BNRWebViewController**.

```
@class BNRWebViewController;

@interface BNRCoursesViewController : UITableViewController

@property (nonatomic) BNRWebViewController *webViewController;

@end
```

In `BNRAppDelegate.m`, import the header for **BNRWebViewController**, create an instance of **BNRWebViewController**, and set it as the `BNRWebViewController` of the **BNRCoursesViewController**.

```
#import "BNRWebViewController.h"

@interface BNRCoursesViewController ()  

@property (nonatomic) NSURLSession *session;  

@property (nonatomic, copy) NSArray *courses;  

@end  

@implementation BNRCAppDelegate  

- (BOOL)application:(UIApplication *)application  

    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions  

{  

    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];  

    BNRCoursesViewController *cvc =  

        [[BNRCoursesViewController alloc] initWithStyle:UITableViewStylePlain];  

    UINavigationController *masterNav =  

        [[UINavigationController alloc] initWithRootViewController:cvc];  

    BNRWebViewController *wvc = [[BNRWebViewController alloc] init];  

    cvc.webViewController = wvc;  

    self.window.rootViewController = masterNav;  

    self.window.backgroundColor = [UIColor whiteColor];  

    [self.window makeKeyAndVisible];  

    return YES;  

}  


```

(Note that you are instantiating the **BNRWebViewController** in the application delegate in preparation for the next chapter, where you will use a **UISplitViewController** to present view controllers on the iPad.)

In **BNRCoursesViewController.m**, import the header files for **BNRWebViewController** and then implement **tableView:didSelectRowAtIndexPath:** to configure and push the **webViewController** onto the navigation stack when a row is tapped.

```
#import "BNRWebViewController.h"

@implementation BNRCoursesViewController  

- (void)tableView:(UITableView *)tableView  

    didSelectRowAtIndexPath:(NSIndexPath *)indexPath  

{  

    NSDictionary *course = self.courses[indexPath.row];  

    NSURL *URL = [NSURL URLWithString:course[@"url"]];  

    self.webViewController.title = course[@"title"];  

    self.webViewController.URL = URL;  

    [self.navigationController pushViewController:self.webViewController  

                                         animated:YES];  

}
```

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 `BNRCoursesViewController.m` and update **fetchFeed** to hit a secure Big Nerd Ranch courses web service. (Do not forget to use https instead of http.)

```
- (void)fetchFeed
{
    NSString *requestString = @"http://bookapi.bignerdranch.com/courses.json";
    NSString *requestString = @"https://bookapi.bignerdranch.com/private/courses.json";
    NSURL *url = [NSURL URLWithString:requestString];
    NSURLRequest *req = [NSURLRequest requestWithURL:url];
```

The **NSURLSession** now needs its delegate to be set upon creation. Update **initWithStyle:** to set the delegate of the session.

```
- (instancetype)initWithStyle:(UITableViewStyle)style
{
    self = [super initWithStyle:style];
    if (self) {
        self.navigationItem.title = @"BNR Courses";

        NSURLSessionConfiguration *config =
            [NSURLSessionConfiguration defaultSessionConfiguration];
        _session = [NSURLSession sessionWithConfiguration:config
                                                delegate:nil
                                               delegateQueue:nil];
        _session = [NSURLSession sessionWithConfiguration:config
                                                delegate:self
                                               delegateQueue:nil];
        [self fetchFeed];
    }
    return self;
}
```

Then update the class extension in `BNRCoursesViewController.m` to conform to the **NSURLSessionDataDelegate** protocol.

```
@interface BNRCoursesViewController () <NSURLSessionDataDelegate>

@property (nonatomic) NSURLSession *session;
@property (nonatomic, copy) NSArray *courses;

@end
```

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 **BNRCoursesViewController** will not have an array of 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 **BNRCoursesViewController.m** implement the **NSURLSessionDataDelegate** method to handle the authentication challenge.

```
- (void)URLSession:(NSURLSession *)session task:(NSURLSessionTask *)task
didReceiveChallenge:(NSURLAuthenticationChallenge *)challenge
completionHandler:
{
    (void (^)(NSURLSessionAuthChallengeDisposition, NSURLCredential *))completionHandler
{
    NSURLCredential *cred =
        [NSURLCredential credentialWithUser:@"BigNerdRanch"
                                      password:@"AchieveNerdvana"
                                         persistence:NSURLConnectionPersistenceForSession];
    completionHandler(NSURLSessionAuthChallengeUseCredential, 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 **NSURLSessionAuthChallengeUseCredential**. 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 UIWebView

A **UIWebView** 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 **BNRWebViewController**'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 **UIWebView** to enable and disable the toolbar items.

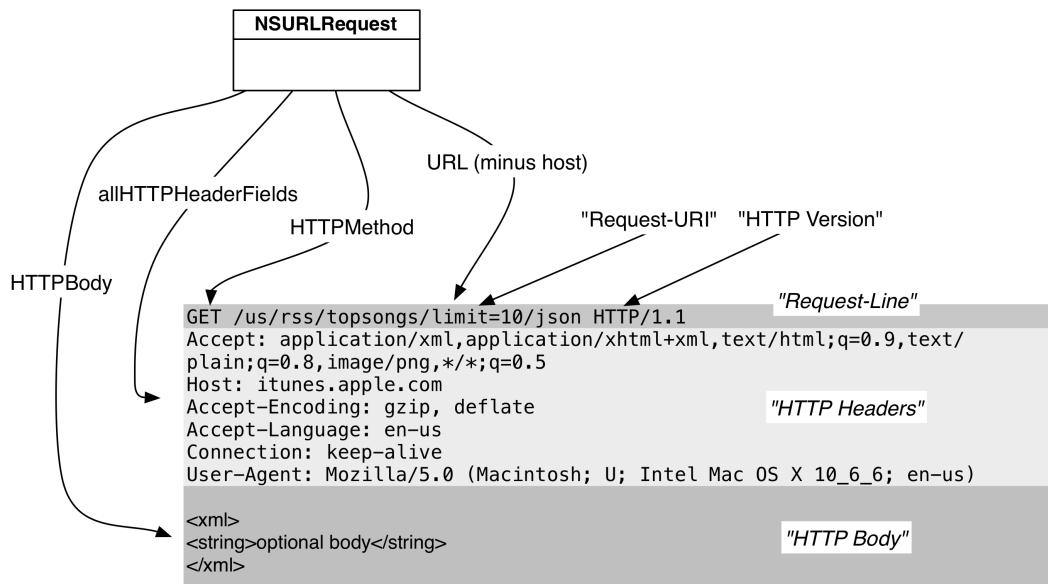
Gold Challenge: Upcoming Courses

In addition to providing general course information, the **courses** web service also returns information about upcoming courses, such as the date and instructor. Create a new **UITableViewCell** subclass that displays the course title and the next time the course is being offered. (Note: not all courses have an upcoming offering.)

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 21.7.

Figure 21.7 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 (which Apple calls a status 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.

```
NSURL *someURL = [NSURL URLWithString:@"http://www.photos.com/upload"];
UIImage *image = [self profilePicture];
NSData *data = UIImagePNGRepresentation(image);

NSMutableURLRequest *req =
    [NSMutableURLRequest requestWithURL:someURL
        cachePolicy:NSURLRequestReloadIgnoringCacheData
        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:[NSString stringWithFormat:@"%d", data.length]
    forHTTPHeaderField:@"Content-Length"];
```

This page intentionally left blank

22

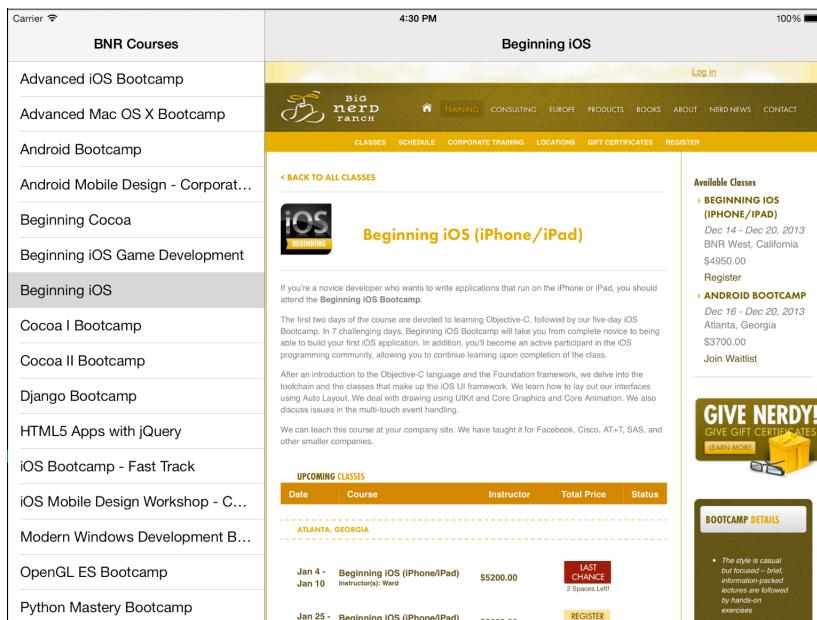
UISplitViewController

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 plenty of screen space to present both views using a built-in class called **UISplitViewController**. **UISplitViewController** is an iPad-only class that presents two view controllers in a master-detail relationship. The master view controller occupies a small strip on the lefthand side of the screen, and the detail view controller occupies the rest of the screen.

In this chapter, you will have Nerdfeed present its view controllers in a split view controller when running on an iPad (Figure 22.1). You will also make Nerdfeed a universal application and have it continue to use a **UINavigationController** when run on the iPhone.

Figure 22.1 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.

Open Nerdfeed.xcodeproj in Xcode. Then, open BNRAppDelegate.m.

In **application:didFinishLaunchingWithOptions:**, check whether the device is an iPad before instantiating a **UISplitViewController**. The **UISplitViewController** class does not exist on the iPhone, and trying to create an instance of **UISplitViewController** will cause an exception to be thrown.

```
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];

    BNRCoursesViewController *lvc =
        [[BNRCoursesViewController alloc] initWithStyle:UITableViewStylePlain];

    UINavigationController *masterNav =
        [[UINavigationController alloc] initWithRootViewController:lvc];

    BNRRWebViewController *wvc = [[BNRWebViewController alloc] init];
    lvc.webViewController = wvc;

    self.window.rootViewController = masterNav;

    // Check to make sure we are running on the iPad
    if ([UIDevice currentDevice].userInterfaceIdiom == UIUserInterfaceIdiomPad) {

        // webViewController must be in navigation controller; you will see why later
        UINavigationController *detailNav =
            [[UINavigationController alloc] initWithRootViewController:wvc];

        UISplitViewController *svc = [[UISplitViewController alloc] init];

        // Set the delegate of the split view controller to the detail VC
        // You will need this later - ignore the warning for now
        svc.delegate = wvc;

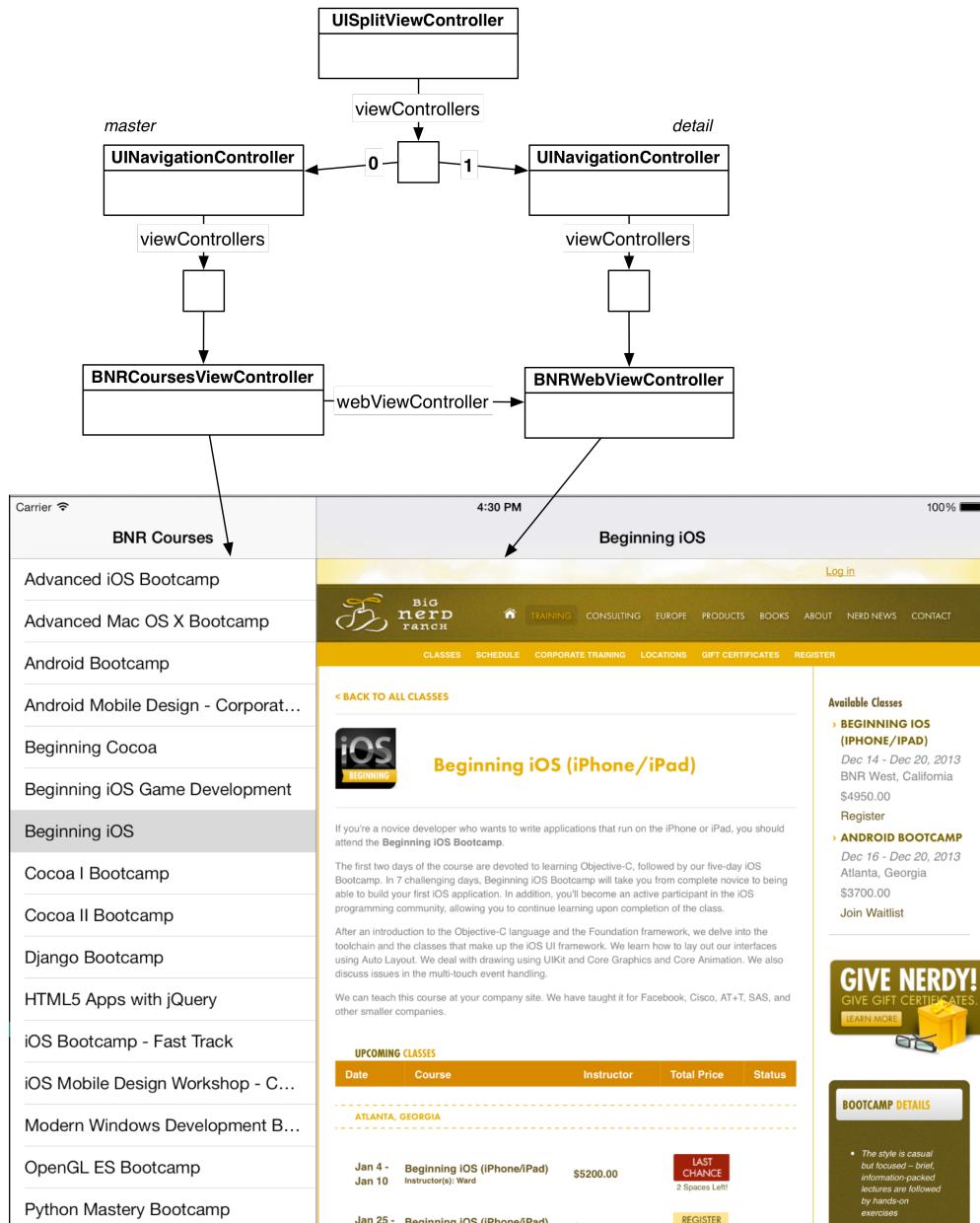
        svc.viewControllers = @[masterNav, detailNav];

        // Set the root view controller of the window to the split view controller
        self.window.rootViewController = svc;
    } else {
        // On non-iPad devices, just use the navigation controller
        self.window.rootViewController = masterNav;
    }

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

By placing the `UISplitViewController` code within an `if` statement in this method, you are laying the groundwork for making Nerdfeed a universal application. Also, now you can see why you created the instance of `BNRWebViewController` here instead of following the typical pattern of creating the detail view controller inside the implementation for the root view controller. A split view controller must have both the master and the detail view controller when it is created. The diagram for Nerdfeed's split view controller is shown in Figure 22.2.

Figure 22.2 Split view controller diagram

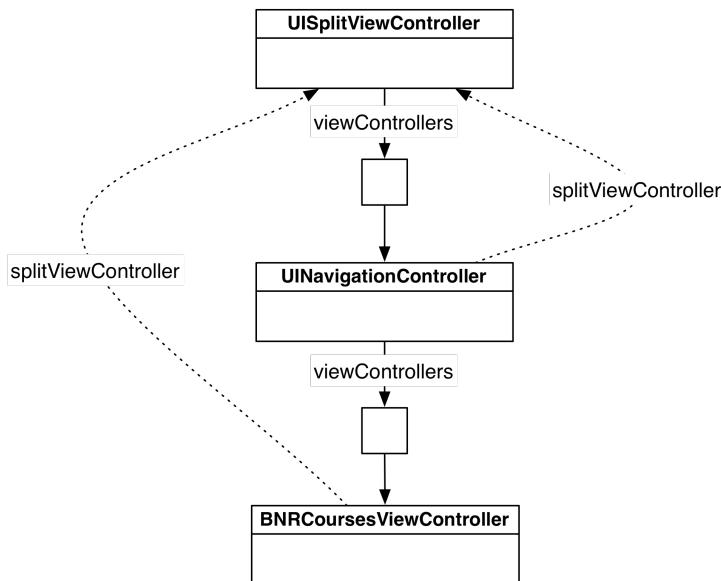


Build and run the application. You may not see anything yet if you are in portrait mode; however, if you rotate the device to landscape you will see both view controllers on the screen. This is how a **UISplitViewController** works: in landscape mode, the master view controller is shown in a small strip on the left hand side of the screen and the detail view controller takes over the rest of the screen.

But you are not done yet. If you tap a row in the list view controller, the web view controller will not appear in the detail panel like you want. Instead, it is pushed onto the master panel and replaces the list view controller. To address this problem, when a row is tapped, you need to check whether the **BNRCoursesViewController** is a member of a split view controller and, if it is, take a different action.

You can send the message **splitViewController** to any **UIViewController**, and if that view controller is part of a split view controller it will return a pointer to the split view controller (Figure 22.3). Otherwise, it returns **nil**. View controllers are smart: a view controller will return this pointer if it is a member of the split view controller's array or if it belongs to another controller that is a member of a split view controller's array (as is the case with both **BNRCoursesViewController** and **BNRWebViewController**).

Figure 22.3 **UIViewController**'s **splitViewController** property



In **BNRCoursesViewController.m**, locate the method **tableView:didSelectRowAtIndexPath:**. At the top of this method, check for a split view controller before pushing the **BNRWebViewController** onto the navigation stack.

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSDictionary *course = self.courses[indexPath.row];
    NSURL *URL = [NSURL URLWithString:course[@"url"]];

    self.webViewController.title = course[@"title"];
    self.webViewController.URL = URL;
    [self.navigationController pushViewController:self.webViewController
                                         animated:YES];

    if (!self.splitViewController) {
        [self.navigationController pushViewController:self.webViewController
                                         animated:YES];
    }
}
```

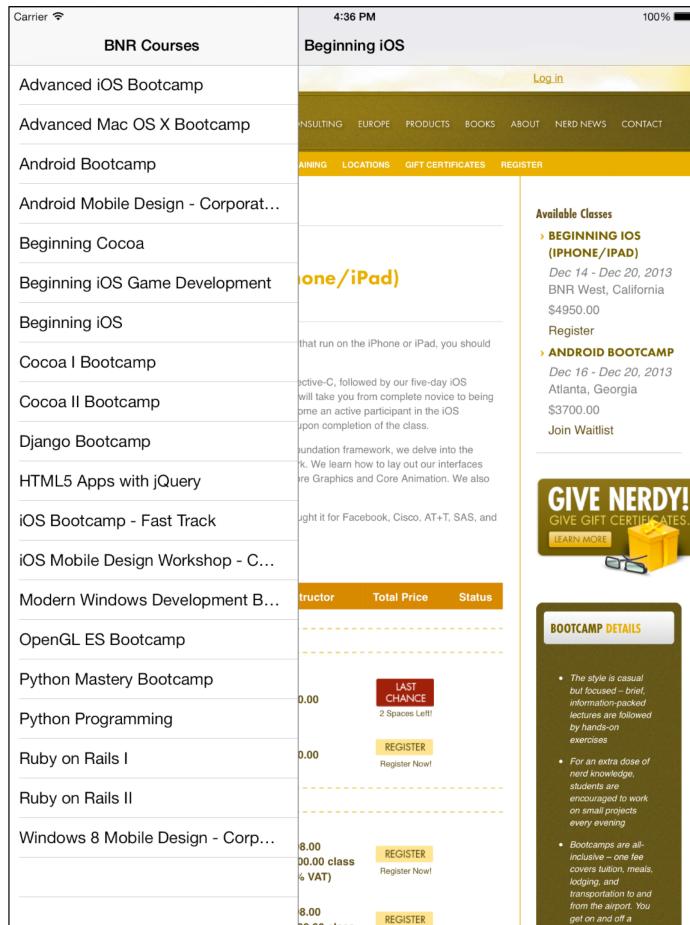
Now, if the **BNRCoursesViewController** is not in a split view controller, you assume the device is not an iPad and **BNRCoursesViewController** pushes the **BNRWebViewController** onto the navigation controller's stack. If **BNRCoursesViewController** is in a split view controller, then it is left to the **UISplitViewController** to place the **BNRWebViewController** on the screen.

Build and run the application again. Rotate to landscape and tap one of the rows. The class web page will now load in the detail panel.

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 **UIBarButtonItem**. Tapping this button shows the master view controller in a specialized **UIPopoverController** (Figure 22.4).

Figure 22.4 Master view controller in UIPopoverController



In your code, whenever a detail view controller was given to the split view controller, that detail view controller was set as the split view controller's delegate. As the delegate, the detail view controller will get a pointer to the **UIBarButtonItem** when rotating to portrait mode.

In **BNRWebViewController.h**, add this declaration:

```
@interface BNRWebViewController : UIViewController <UISplitViewControllerDelegate>
```

Build and run the application. The behavior will be the same, but there will not be any warnings.

In **BNRWebViewController.m**, implement the following delegate method to place the bar button item in the **BNRWebViewController**'s navigation item.

```
- (void)splitViewController:(UISplitViewController *)svc
    willHideViewController:(UIViewController *)aViewController
        withBarButtonItem:(UIBarButtonItem *)barButtonItem
    forPopoverController:(UIPopoverController *)pc
{
    // If this bar button item does not have a title, it will not appear at all
    barButtonItem.title = @"Courses";

    // Take this bar button item and put it on the left side of the nav item
    self.navigationItem.leftBarButtonItem = barButtonItem;
}
```

Notice that you explicitly set the title of the button. If the button does not have a title, it will not appear at all. (If the master view controller's `navigationItem` has a title, then the button will be automatically set to that title.)

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 `BNRWebViewController`'s view.)

There are two small issues left to address with your Courses button. First, when the device is rotated back to landscape mode, the button is still there. To remove it, the delegate needs to respond to another message from the `UISplitViewController`. Implement this delegate method in `BNRWebViewController.m`.

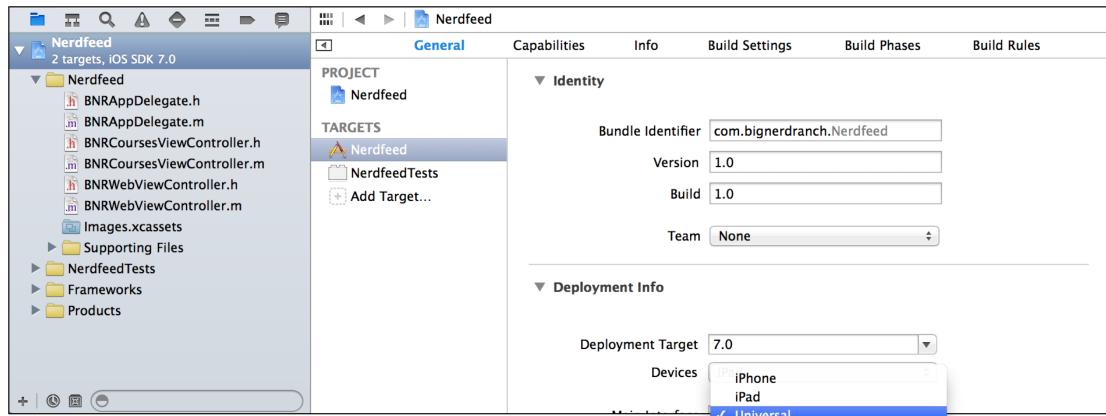
```
- (void)splitViewController:(UISplitViewController *)svc
    willShowViewController:(UIViewController *)aViewController
    invalidatingBarButtonItem:(UIBarButtonItem *)barButtonItem
{
    // Remove the bar button item from the navigation item
    // Double check that it is the correct button, even though we know it is
    if (barButtonItem == self.navigationItem.leftBarButtonItem) {
        self.navigationItem.leftBarButtonItem = nil;
    }
}
```

Build and run the application. The Courses button will now appear and disappear as you rotate between portrait and landscape modes.

Universalizing Nerdfeed

You created Nerdfeed as an iPad-only application; now you are going to universalize it. Select the Nerdfeed project from the project navigator. In the editor area, choose the Nerdfeed target and then the General tab.

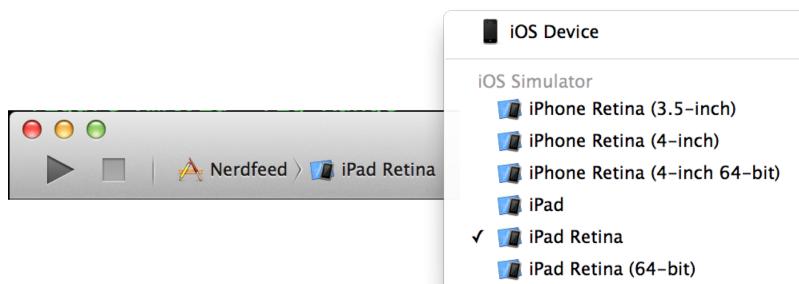
Figure 22.5 Universalizing Nerdfeed



From the Devices pop-up menu, choose Universal.

The application is now universal. It will run fine as-is on an iPhone. You can test it by building and running again on one simulator and then the other.

Figure 22.6 Changing simulators



There are two reasons the universalization process was so simple for Nerdfeed. Remembering these reasons will help you when you are writing your own applications.

- As you built Nerdfeed, you were careful about the device differences in the classes used. For example, knowing that a **UISplitViewController** does not exist on the iPhone or iPod touch, you included an alternative interface for those devices. In general, when using an Apple-provided class, you should read the discussion in the documentation about that class. It will give you tips about the availability of the class and its methods on the different devices.
- Nerdfeed is still a relatively simple application. It is always easier to universalize an application early in development. As an application grows, its details get buried in the massive pile of code. Finding and fixing issues as you are writing code is much easier than coming back later. Details are harder to find, and there is the risk of breaking what already works.

This page intentionally left blank

23

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 Objective-C 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 `BNRItemStore`.

Moving Homepwner to Core Data

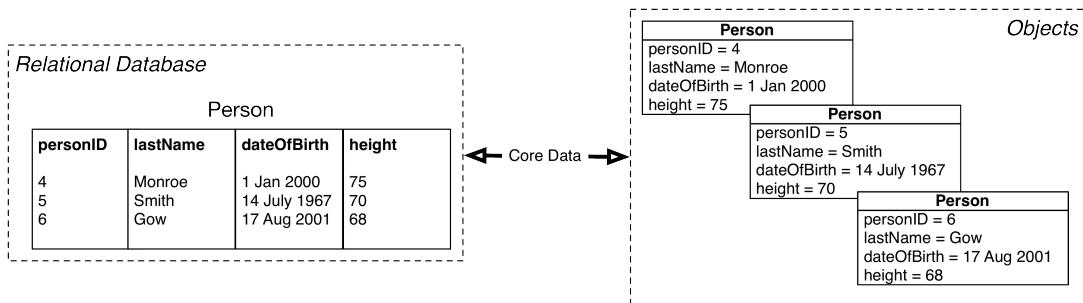
Your Homepwner application currently uses archiving to save and reload its data. For a moderately sized object model (say, fewer than 1000 objects), this is fine. As your object model gets larger, however, you will want to be able to do incremental fetches and updates, and Core Data can do this.

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 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 23.1 Role of Core Data

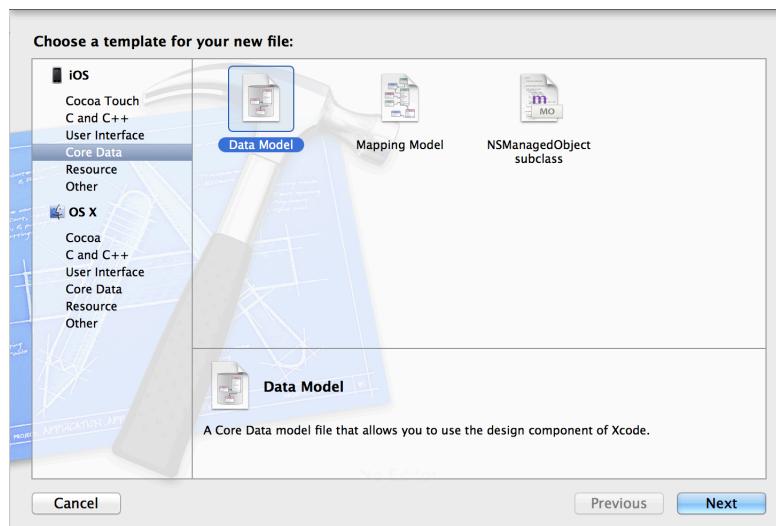


This organization translates well to Objective-C. Every table is like an Objective-C class. Every column is one of the class's properties. Every row is an instance of that class. Thus, Core Data's job is to move data to and from these two representations (Figure 23.1).

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 **BNRItem** 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`.

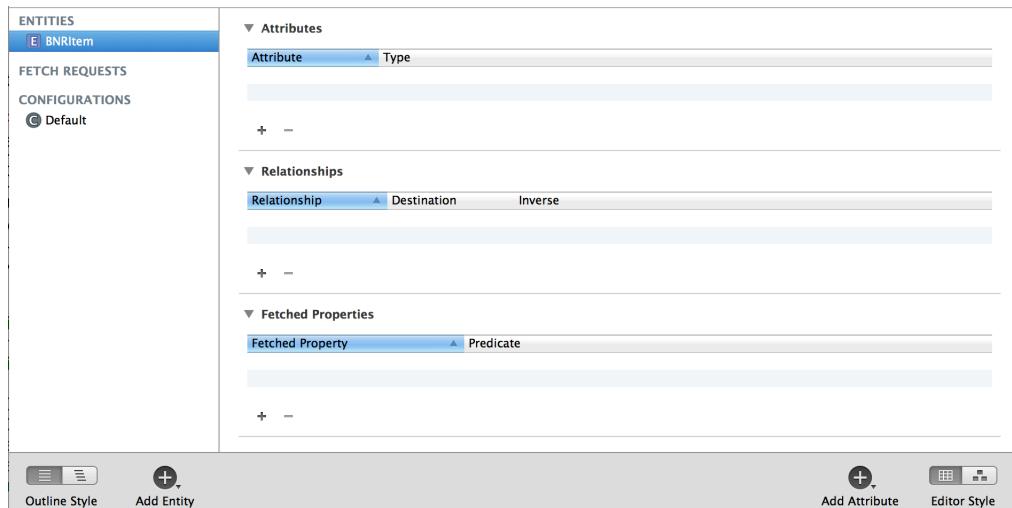
Figure 23.2 Create the model file



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 **BNRItem** (Figure 23.3).

Figure 23.3 Create the **BNRItem** entity



Now your **BNRItem** entity needs attributes. Remember that these will be the properties of the **BNRItem** class. The necessary attributes are listed below. For each attribute, click the + button in the Attributes section and edit the Attribute and Type values:

- `itemName` is a String
- `serialNumber` is a String
- `valueInDollars` is an Integer 32
- `dateCreated` is a Date
- `itemKey` is a String
- `thumbnail` is a Transformable (It is a **UIImage**, but that is not one of the possibilities. We will discuss **Transformable** shortly.)

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 **BNREmployee** 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`.

Core Data is only able to store certain data types in its store, and `UIImage` is not one of these types. Instead, you declared the thumbnail as *transformable*. With a transformable attribute, Core Data will convert the object into `NSData` when saving, and convert the `NSData` back into the original object when loading it from the file system. In order for Core Data to do this, you have to supply it with an `NSValueTransformer` subclass that handles these conversions.

Create a new class named `BNRImageTransformer` that is a subclass of `NSValueTransformer`. Open `BNRImageTransformer.m` and override the methods necessary for transforming the `UIImage` to and from `NSData`.

```
@implementation BNRImageTransformer

+ (Class)transformedValueClass
{
    return [NSData class];
}

- (id)transformedValue:(id)value
{
    if (!value) {
        return nil;
    }

    if ([value isKindOfClass:[NSData class]]) {
        return value;
    }

    return UIImagePNGRepresentation(value);
}

- (id)reverseTransformedValue:(id)value
{
    return [UIImage imageWithData:value];
}

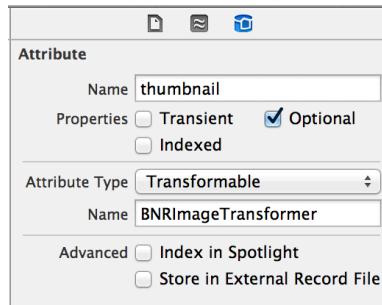
@end
```

The implementation of `BNRImageTransformer` is pretty straightforward. The class method `transformedValueClass` tells the transformer what type of object it will receive from the `transformedValue:` method. The `transformedValue:` method will be called when your transformable variable is to be saved to the file system, and it expects an object that can be saved to Core Data. In this example, the argument to the method will be a `UIImage` and it will return an instance of `NSData`. Finally, the `reverseTransformedValue:` method is called when the thumbnail data is loaded from the file system, and your implementation will create the `UIImage` from the `NSData` that was stored. With this file created, Core Data must know to use this class when working with the thumbnail.

Open `Homepwner.xcdatamodeld` and select the `BNRItem` entity. Select thumbnail from the Attributes list and then click the  tab in the inspector selector to show the *data model inspector*. Replace the

Value Transformer Name placeholder text in the second Name field with `BNRImageTransformer` (Figure 23.4).

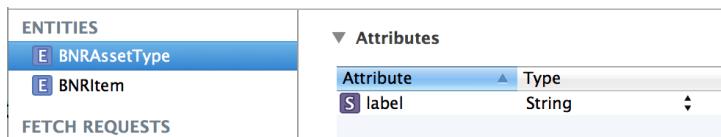
Figure 23.4 Update value transformer name for thumbnail attribute



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 `BNRAssetType` that describes a category of items. For example, a painting might be of the Art asset type. `BNRAssetType` will be an entity in the model file, and each row of that table will be mapped to an Objective-C object at runtime.

In `Homeowner.xcdatamodeld`, add another entity called `BNRAssetType`. Give it an attribute called `label` of type String. This will be the name of the category the `BNRAssetType` represents.

Figure 23.5 Create the `BNRAssetType` entity



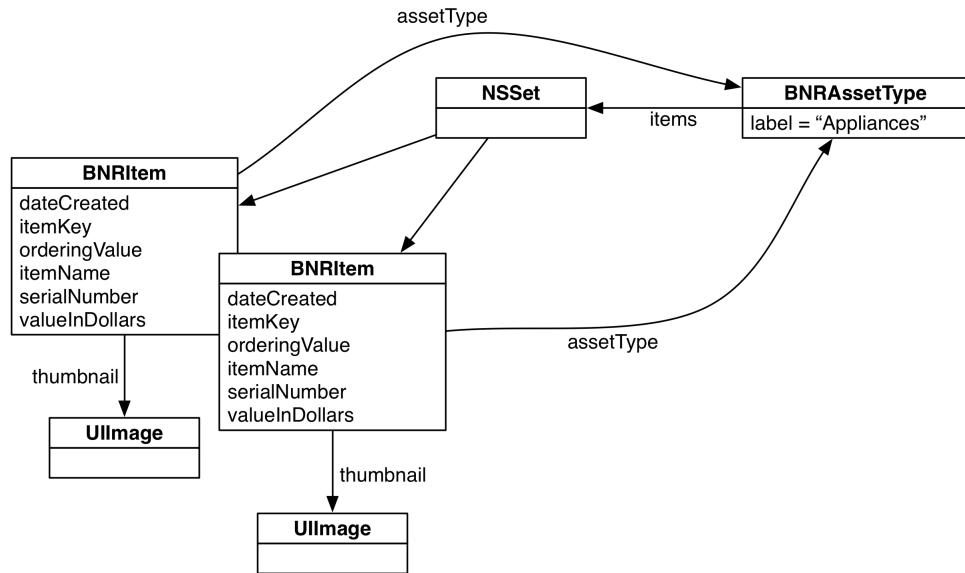
Now you need to establish relationships between `BNRAssetType` and `BNRItem`. Relationships between entities are represented by pointers between objects. There are two kinds of relationships: *to-one* and *to-many*.

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 `BNRItem` entity will have a to-one relationship to the `BNRAssetType` entity. Thus, a `BNRItem` instance will have a pointer to a `BNRAssetType` 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 `BNRAssetType` entity will have a to-many relationship to the `BNRItem` entity because many instances of `BNRItem` can have the same `BNRAssetType`. Thus, a `BNRAssetType` object will have a pointer to a set of all of the `BNRItem` objects that are its type of asset.

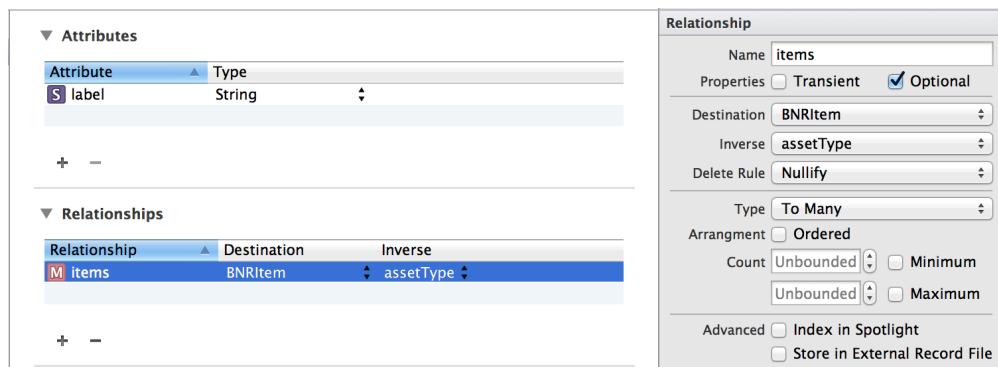
With these relationships set up, you can ask a `BNRAssetType` object for the set of `BNRItem` objects that fall into its category, and you can ask a `BNRItem` which `BNRAssetType` it falls under. Figure 23.6 diagrams the relationships between `BNRAssetType` and `BNRItem`.

Figure 23.6 Entities in Homepwner



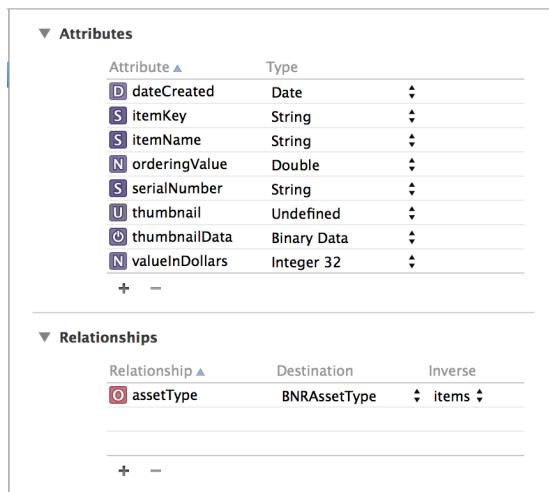
Let's add these relationships to the model file. Select the **BNRAssetType** entity and then click the **+** button in the Relationships section. Name this relationship **items** in the Relationship column. Then, select **BNRItem** from the Destination column. In the data model inspector, change the Type drop-down from To One to To Many (Figure 23.7).

Figure 23.7 Create the **items** relationship



Now go back to the **BNRItem** entity. Add a relationship named **assetType** and pick **BNRAssetType** as its destination. In the Inverse column, select **items** (Figure 23.8).

Figure 23.8 Create the assetType relationship



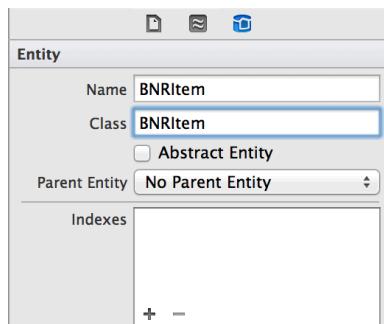
NSManagedObject and subclasses

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**.

Select the BNRItem entity. Show the data model inspector and change the Class field to **BNRItem**, as shown in Figure 23.9. Now, when a **BNRItem** entity is fetched with Core Data, the type of this object will be **BNRItem**. (**BNRAAssetType** instances will still be of type **NSManagedObject**.)

Figure 23.9 Changing the class of an entity



There is one problem: the **BNRItem** class already exists, and it does not inherit from **NSManagedObject**. Changing the superclass of the existing **BNRItem** to **NSManagedObject** will require considerable modifications. Thus, the easiest solution is to remove your current **BNRItem** class files, have Core Data generate a new **BNRItem** class, and then add your behavior methods back to the new class files.

In Finder, drag both **BNRItem.h** and **BNRItem.m** to your desktop for safekeeping. Then, in Xcode, delete these two files from the project navigator. (They will appear in red after you have moved the files).

Now, open **Homepwner.xcdatamodeld** again and select the **BNRItem** entity. Then, select **New File...** from the **New** menu.

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 **BNRItem** is checked, and then click **Next** one last time. Finally, click **Create** to generate the **NSManagedObject** subclass files.

Xcode will generate new **BNRItem.h** and **BNRItem.m** files. Open **BNRItem.h** and see what Core Data has wrought. Change the type of the **thumbnail** property to **UIImage** and add the method declaration from the previous **BNRItem**. 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/Foundation.h>
#import CoreData;

@interface BNRItem : NSManagedObject

@property (nonatomic, strong) NSDate * dateCreated;
@property (nonatomic, strong) NSString * itemKey;
@property (nonatomic, strong) NSString * itemName;
@property (nonatomic, strong) NSNumber * orderingValue;
@property (nonatomic) double orderingValue;
@property (nonatomic, strong) NSString * serialNumber;
@property (nonatomic, strong) id thumbnail;
@property (nonatomic, strong) UIImage *thumbnail;
@property (nonatomic, strong) NSData * thumbnailData;
@property (nonatomic, strong) NSNumber * valueInDollars;
@property (nonatomic) int valueInDollars;
@property (nonatomic, strong) NSManagedObject *assetType;

- (void)setThumbnailFromImage:(UIImage *)image;

@end
```

(Xcode might have created the **strong** properties as **retain**. These represent the same thing; before ARC, **strong** properties were called **retain**, and not all parts of the tools have been updated to the new terminology.)

Copy the **setThumbnailFromImage:** method from your old **BNRItem.m** to the new one:

```

- (void)setThumbnailFromImage:(UIImage *)image
{
    CGSize origImageSize = image.size;
    CGRect newRect = CGRectMake(0, 0, 40, 40);
    float ratio = MAX(newRect.size.width / origImageSize.width,
                       newRect.size.height / origImageSize.height);

    UIGraphicsBeginImageContextWithOptions(newRect.size, NO, 0.0);
    UIBezierPath *path = [UIBezierPath bezierPathWithRoundedRect:newRect
                                                          cornerRadius:5.0];
    [path addClip];

    CGRect projectRect;
    projectRect.size.width = ratio * origImageSize.width;
    projectRect.size.height = ratio * origImageSize.height;
    projectRect.origin.x = (newRect.size.width - projectRect.size.width) / 2.0;
    projectRect.origin.y = (newRect.size.height - projectRect.size.height) / 2.0;

    [image drawInRect:projectRect];

    UIImage *smallImage = UIGraphicsGetImageFromCurrentImageContext();
    self.thumbnail = smallImage;

    UIGraphicsEndImageContext();
}

```

Of course, when you first launch an application, there are no saved items or asset types. When the user creates a new **BNRItem** 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 **BNRItem**. Implement **awakeFromInsert** in `BNRItem.m`.

```

- (void)awakeFromInsert
{
    [super awakeFromInsert];

    self.dateCreated = [NSDate date];

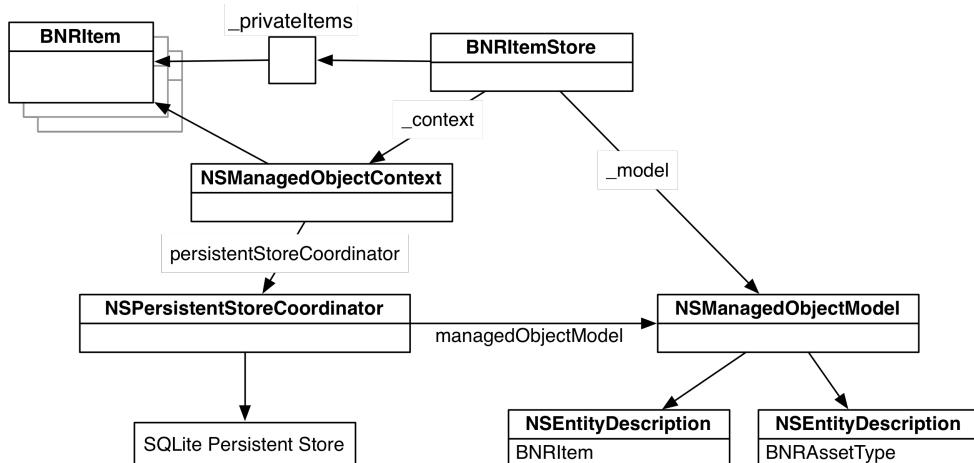
    // Create an NSUUID object - and get its string representation
    NSUUID *uuid = [[NSUUID alloc] init];
    NSString *key = [uuid UUIDString];
    self.itemKey = key;
}

```

This adds the extra behavior of **BNRItem**'s old designated initializer. Build the application to check for syntax errors, but do not run it.

Updating BNRItemStore

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 Homepwner, these objects will work with the **BNRItemStore**. These relationships are shown in Figure 23.10.

Figure 23.10 **BNRItemStore** and **NSManagedObjectContext**

In `BNRItemStore.m`, import Core Data and add three properties to the class extension.

```

#import CoreData;
 
@interface BNRItemStore : NSObject
  
```

`@property (nonatomic) NSMutableArray *privateItems;`

`@property (nonatomic, strong) NSMutableArray *allAssetTypes;`

`@property (nonatomic, strong) NSManagedObjectContext *context;`

`@property (nonatomic, strong) NSManagedObjectModel *model;`

Then change the implementation of `itemArchivePath` to return a different path that Core Data will use to save data.

```

- (NSString *)itemArchivePath
{
    NSArray *documentDirectories =
        NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                            NSUserDomainMask,
                                            YES);

    // Get one and only document directory from that list
    NSString *documentDirectory = [documentDirectories firstObject];

    return [documentDirectory stringByAppendingPathComponent:@"items.archive"];

    return [documentDirectory stringByAppendingPathComponent:@"store.data"];
}
  
```

When the **BNRItemStore** 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 `BNRItemStore.m`, update `initPrivate`.

```

- (instancetype)initPrivate
{
    self = [super init];
    if (self) {
        NSString *path = self.itemArchivePath;
        _privateItems = [NSKeyedUnarchiver unarchiveObjectWithFile:path];

        if (!_privateItems) {
            _privateItems = [[NSMutableArray alloc] init];
        }

        // Read in Homeowner.xcdatamodeld
        _model = [NSManagedObjectModel mergedModelFromBundles:nil];

        NSPersistentStoreCoordinator *psc =
            [[NSPersistentStoreCoordinator alloc] initWithManagedObjectModel:_model];

        // Where does the SQLite file go?
        NSString *path = self.itemArchivePath;
        NSURL *storeURL = [NSURL fileURLWithPath:path];

        NSError *error = nil;

        if (![psc addPersistentStoreWithType:NSSQLiteStoreType
                                      configuration:nil
                                         URL:storeURL
                                         options:nil
                                         error:&error]) {
            @throw [NSError exceptionWithName:@"OpenFailure"
                                         reason:[error localizedDescription]
                                         userInfo:nil];
        }

        // Create the managed object context
        _context = [[NSManagedObjectContext alloc] init];
        _context.persistentStoreCoordinator = psc;
    }
    return self;
}

```

Before, **BNRItemStore** would write out the entire **NSMutableArray** of **BNRItem** 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 **BNRItemStore.m**, change **saveChanges**.

```

- (BOOL)saveChanges
{
    NSString *path = [self itemArchivePath];
    return [NSKeyedArchiver archiveRootObject:_allItems
           toFile:[self itemArchivePath]];

    NSError *error;
    BOOL successful = [self.context save:&error];
    if (!successful) {
        NSLog(@"Error saving: %@", [error localizedDescription]);
    }
    return successful;
}

```

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 `BNRItem` instances, you specify the `BNRItem` 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 `BNRItem` by `orderingValue` in ascending order.

In `BNRItemStore.m`, define a new method, `loadAllItems`, to prepare and execute the fetch request and save the results into the `allItems` array.

```
- (void)loadAllItems
{
    if (!self.privateItems) {
        NSFetchRequest *request = [[NSFetchRequest alloc] init];

        NSEntityDescription *e = [NSEntityDescription entityForName:@"BNRItem"
                                inManagedObjectContext:self.context];
        request.entity = e;

        NSSortDescriptor *sd = [NSSortDescriptor
                               sortDescriptorWithKey:@"orderingValue"
                               ascending:YES];
        request.sortDescriptors = @[sd];

        NSError *error;
        NSArray *result = [self.context executeFetchRequest:request error:&error];
        if (!result) {
            [NSError raise:@"Fetch failed"
                      format:@"Reason: %@", [error localizedDescription]];
        }

        self.privateItems = [[NSMutableArray alloc] initWithArray:result];
    }
}
```

Also in `BNRItemStore.m`, send this message to the `BNRItemStore` at the end of `initPrivate`.

```
_context.persistentStoreCoordinator = psc;

    [self loadAllItems];
}
return self;
}
```

You can build to check for syntax errors.

In this application, you immediately fetched all the instances of the `BNRItem` 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.

Predicates can also be used to filter the contents of an array. So, even if you had already fetched the `allItems` array, you could still use a predicate:

```
NSArray *expensiveStuff = [allItems filteredArrayUsingPredicate:p];
```

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 `BNRItem`, you will not allocate and initialize this new `BNRItem`. Instead, you will ask the `NSManagedObjectContext` to insert a new object from the `BNRItem` entity. It will then return an instance of `BNRItem`.

In `BNRItemStore.m`, edit the `createItem` method.

```
- (BNRItem *)createItem
{
    BNRItem *item = [[BNRItem alloc] init];
    double order;
    if ([self.allItems count] == 0) {
        order = 1.0;
    } else {
        order = [[self.privateItems lastObject] orderingValue] + 1.0;
    }
    NSLog(@"Adding after %d items, order = %.2f", [self.privateItems count], order);

    BNRItem *item = [NSEntityDescription insertNewObjectForEntityForName:@"BNRItem"
                                                inManagedObjectContext:self.context];

    item.orderingValue = order;

    [self.privateItems addObject:item];
    return item;
}
```

When a user deletes a `BNRItem`, you must inform the context so that it is removed from the database. In `BNRItemStore.m`, add the following code to `removeItem`:

```
- (void)removeItem:(BNRItem *)item
{
    NSString *key = item.itemKey;
    [[BNRImageStore sharedStore] deleteImageForKey:key];

    [self.context deleteObject:item];
    [self.privateItems removeObjectIdenticalTo:item];
}
```

Reordering items

The last bit of functionality you need to replace for `BNRItem` is the ability to re-order items in the `BNRItemStore`. Because Core Data will not handle ordering automatically, you must update a `BNRItem`'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 `BNRItem` 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 `BNRItemStore.m`, modify `moveItemAtIndex:toIndex:` to handle reordering items.

```
- (void)moveItemAtIndex:(NSUInteger)fromIndex
                  toIndex:(NSUInteger)toIndex
{
    if (fromIndex == toIndex) {
        return;
    }
    BNRItem *item = self.privateItems[fromIndex];
    [self.privateItems removeObjectAtIndex:fromIndex];
    [self.privateItems insertObject:item atIndex:toIndex];

    // Computing a new orderValue for the object that was moved
    double lowerBound = 0.0;

    // Is there an object before it in the array?
    if (toIndex > 0) {
        lowerBound = [self.privateItems[(toIndex - 1)] orderingValue];
    } else {
        lowerBound = [self.privateItems[1] orderingValue] - 2.0;
    }

    double upperBound = 0.0;

    // Is there an object after it in the array?
    if (toIndex < [self.privateItems count] - 1) {
        upperBound = [self.privateItems[(toIndex + 1)] orderingValue];
    } else {
        upperBound = [self.privateItems[(toIndex - 1)] orderingValue] + 2.0;
    }

    double newOrderValue = (lowerBound + upperBound) / 2.0;

    NSLog(@"moving to order %f", newOrderValue);
    item.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 `BNRAssetType` to Homeowner

In the model file, you described a new entity, `BNRAssetType`, that every item will have a to-one relationship to. You need a way for the user to set the `BNRAssetType` of a `BNRItem`. Also, the `BNRItemStore` will need a way to fetch the asset types. (Creating new instances of `BNRAssetType` is left as a challenge at the end of this chapter.)

In `BNRItemStore.h`, declare a new method.

```
- (NSArray *)allAssetTypes;
```

In `BNRItemStore.m`, define this method. If this is the first time the application is being run – and therefore there are no `BNRAssetType` objects in the store – create three default types.

```
- (NSArray *)allAssetTypes
{
    if (!_allAssetTypes) {
        NSFetchRequest *request = [[NSFetchRequest alloc] init];
        NSEntityDescription *e = [NSEntityDescription entityForName:@"BNRAssetType"
                                inManagedObjectContext:self.context];
        request.entity = e;
        NSError *error = nil;
        NSArray *result = [self.context executeFetchRequest:request error:&error];
        if (!result) {
            [NSErrorException raise:@"Fetch failed"
                           format:@"Reason: %@", [error localizedDescription]];
        }
        _allAssetTypes = [result mutableCopy];
    }

    // Is this the first time the program is being run?
    if ([_allAssetTypes count] == 0) {
        NSManagedObject *type;

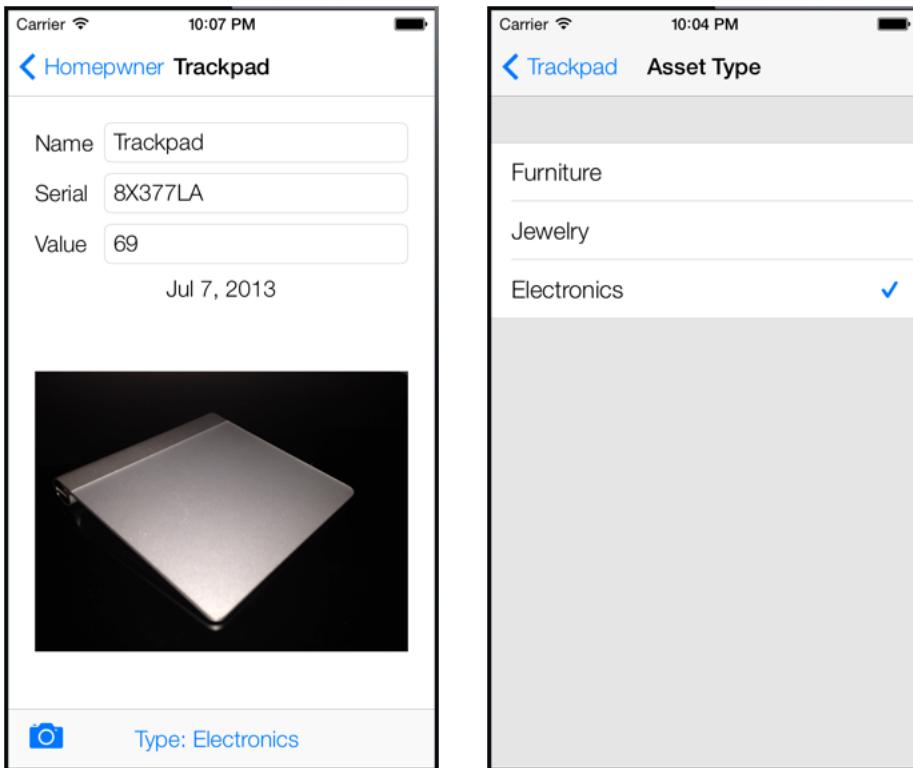
        type = [NSEntityDescription insertNewObjectForEntityForName:@"BNRAssetType"
                           inManagedObjectContext:self.context];
        [type setValue:@"Furniture" forKey:@"label"];
        [_allAssetTypes addObject:type];

        type = [NSEntityDescription insertNewObjectForEntityForName:@"BNRAssetType"
                           inManagedObjectContext:self.context];
        [type setValue:@"Jewelry" forKey:@"label"];
        [_allAssetTypes addObject:type];

        type = [NSEntityDescription insertNewObjectForEntityForName:@"BNRAssetType"
                           inManagedObjectContext:self.context];
        [type setValue:@"Electronics" forKey:@"label"];
        [_allAssetTypes addObject:type];
    }
    return _allAssetTypes;
}
```

Now you need to change the user interface so that the user can see and change the `BNRAssetType` of the `BNRItem` in the `BNRDetailViewController`.

Figure 23.11 Interface for **BNRAssetType**



Create a new Objective-C class template file and choose `NSObject` as the superclass. Name this class **BNRAssetTypeViewController**.

In `BNRAssetTypeViewController.h`, forward declare `BNRItem`, change the superclass to `UITableViewController`, and give it a `BNRItem` property.

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

@class BNRItem;

@interface BNRAssetTypeViewController : NSObject
@interface BNRAssetTypeViewController : UITableViewController

@property (nonatomic, strong) BNRItem *item;

@end
```

This table view controller will show a list of the available asset types. Tapping a button on the `BNRDetailViewController`'s view will display it. Implement the data source methods and import the appropriate header files in `BNRAssetTypeViewController.m`. (You have seen all this before.)

```
#import "BNRAssetTypeViewController.h"

#import "BNRItemStore.h"
#import "BNRItem.h"

@implementation BNRAssetTypeViewController

- (instancetype)init
{
    return [super initWithStyle:UITableViewStylePlain];
}

- (instancetype)initWithStyle:(UITableViewStyle)style
{
    return [self init];
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    [self.tableView registerClass:[UITableViewCell class]
        forCellReuseIdentifier:@"UITableViewCell"];
}

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    return [[[BNRItemStore sharedStore] allAssetTypes] count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:@"UITableViewCell"
            forIndexPath:indexPath];

    NSArray *allAssets = [[BNRItemStore sharedStore] allAssetTypes];
    NSManagedObject *assetType = allAssets[indexPath.row];

    // Use key-value coding to get the asset type's label
    NSString *assetLabel = [assetType valueForKey:@"label"];
    cell.textLabel.text = assetLabel;

    // Checkmark the one that is currently selected
    if (assetType == self.item.assetType) {
        cell.accessoryType = UITableViewCellAccessoryCheckmark;
    } else {
        cell.accessoryType = UITableViewCellAccessoryNone;
    }

    return cell;
}
```

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView cellForRowAtIndexPath:indexPath];
    cell.accessoryType = UITableViewCellAccessoryCheckmark;
    NSArray *allAssets = [[BNRItemStore sharedStore] allAssetTypes];
    NSManagedObject *assetType = allAssets[indexPath.row];
    self.item.assetType = assetType;
    [self.navigationController popViewControllerAnimated:YES];
}
```

```
@end
```

In `BNRDetailViewController.xib`, drag a `UIBarButton Item` onto the toolbar. Create an outlet to this button by selecting the toolbar then the new bar button and Control-dragging to the class extension of `BNRDetailViewController.m`. Name this outlet `assetTypeButton`. Then, create an action from this button in the same way by dragging into the `@implementation` section instead of the class extension and name it `showAssetTypePicker`.

The following method and instance variable should now be declared in `BNRDetailViewController.m`:

```
@property (weak, nonatomic) IBOutlet UIBarButtonItem *assetTypeButton;
@end
@implementation BNRDetailViewController
// Other methods here
- (IBAction)showAssetTypePicker:(id)sender
{
}
@end
```

At the top of `BNRDetailViewController.m`, import the header for this new table view controller.

```
#import "BNRDetailViewController.h"
#import "BNRAssetTypeViewController.h"
```

Finish implementing `showAssetTypePicker:` in `BNRDetailViewController.m`.

```
- (IBAction)showAssetTypePicker:(id)sender
{
    [self.view endEditing:YES];
    BNRAssetTypeViewController *avc = [[BNRAssetTypeViewController alloc] init];
    avc.item = self.item;
    [self.navigationController pushViewController:avc
                                    animated:YES];
}
```

And finally, update the title of the button to show the asset type of a **BNRItem**. In `BNRDetailViewController.m`, add the following code to `viewWillAppear:`.

```
if (self.itemKey) {
    // Get image for image key from image cache
    UIImage *imageToDisplay = [[BNRImageStore sharedStore]
                                imageForKey:self.itemKey];

    // Use that image to put on the screen in imageView
    self.imageView.image = imageToDisplay;
} else {
    // clear the imageView
    imageView.image = nil;
}
NSString *typeLabel = [self.item.assetType valueForKey:@"label"];
if (!typeLabel) {
    typeLabel = @"None";
}

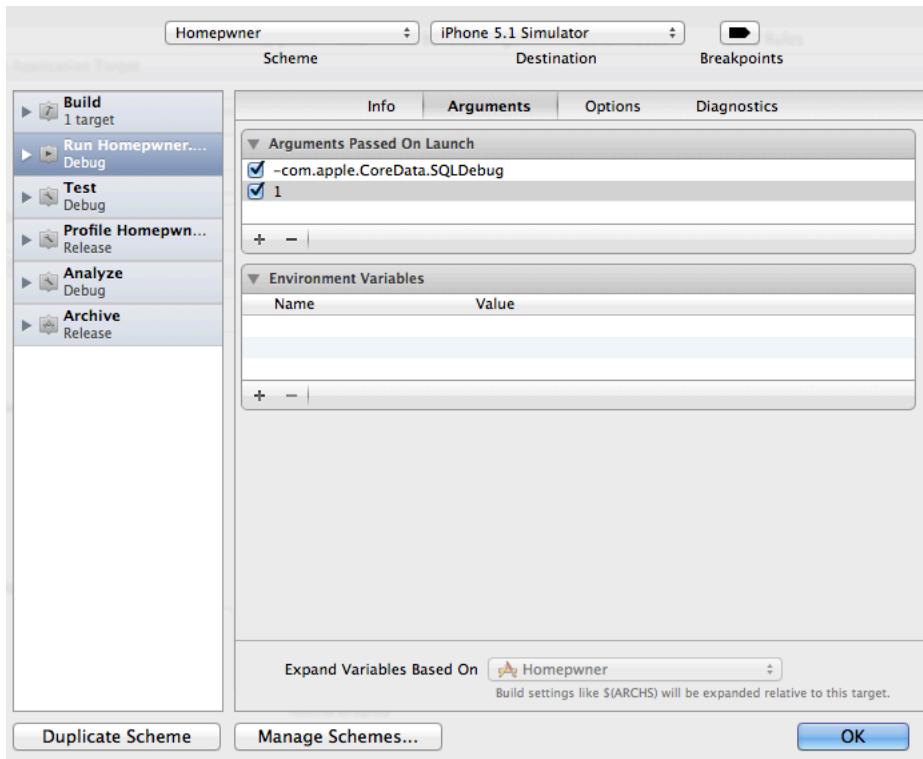
self.assetTypeButton.title = [NSString stringWithFormat:@"Type: %@", typeLabel];
[self updateFonts];
}
```

Build and run the application. Select a **BNRItem** and set its asset type.

More About SQL

In this chapter, you used SQLite via Core Data. If you are curious about what SQL commands Core Data is executing, you can use a command-line argument to log all communications with the SQLite database to the console. From the Product menu, choose Edit Scheme.... Select the Run Homeowner.app item and the Arguments tab. Add two arguments: `-com.apple.CoreData.SQLDebug` and `1`, as shown.

Figure 23.12 Turning on Core Data logging



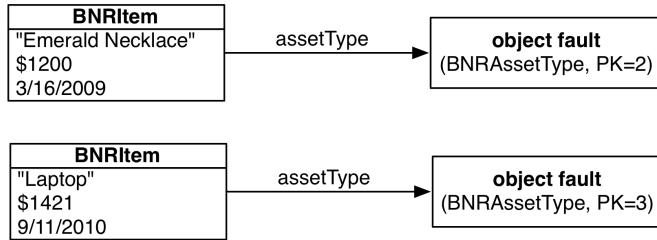
Build and run the application again. Make sure the debug area and console are visible so you can see the SQL logging. Add a few locations and inventory items. Then navigate around the application looking at various items.

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 **BNRItem** are fetched into your application, the instances of **BNRAssetType** are not. Instead, fault objects are created that stand in for the **BNRAssetType** objects until they are really needed.

Figure 23.13 Object faults

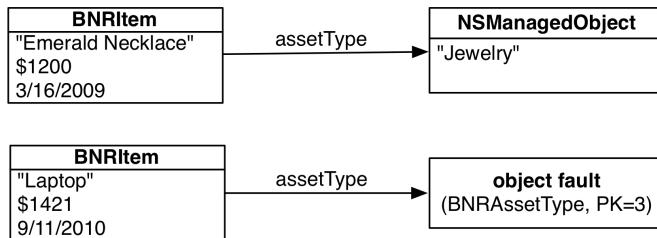


An object fault knows what entity it is from and what its primary key is. So, for example, when you ask a fault that represents an asset type what its label is, you will see SQL executed that looks something like this:

```
SELECT t0.Z_PK, t0.Z_OPT, t0.ZLABEL FROM ZBNRASSETTYPE 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.

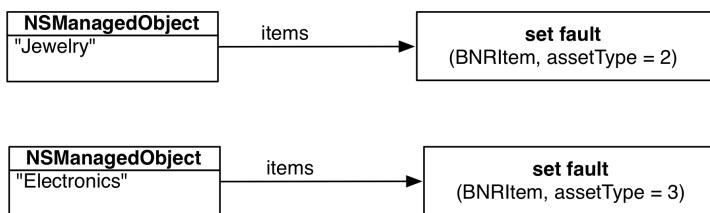
Figure 23.14 After one fault is replaced



This lazy fetching makes Core Data not only easy to use, but also quite efficient.

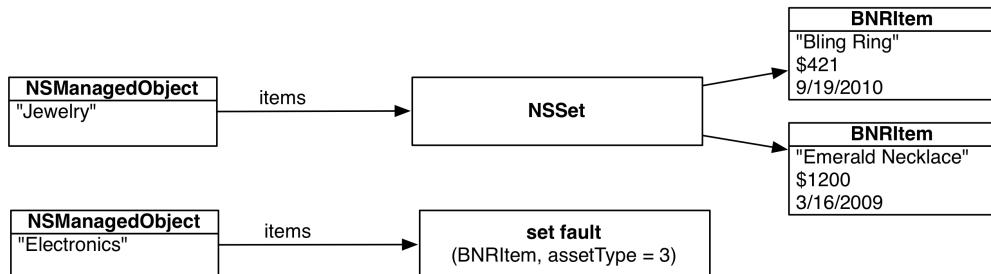
What about to-many faults? Imagine that your application worked the other way: the user is presented with a list of asset types to select from. Then, the items for that asset type are fetched and displayed. How would this work? When the assets are first fetched, each one has a set fault that is standing in for the `NSSet` of item objects:

Figure 23.15 Set faults



When the set fault is sent a message that requires the **BNRItem** objects, it fetches them and replaces itself with an **NSSet**:

Figure 23.16 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**, **NSEntityDescription**, and **NSEntityDescription**. 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 23.1 to help you decide.

Table 23.1 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 **BNRAssetTypeViewController** 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 **BNRAssetTypeViewController**'s `navigationItem`.

Gold Challenge: Showing Assets of a Type

In the **BNRAssetTypeViewController** 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.

This page intentionally left blank

24

State Restoration

As we discussed in Chapter 18, applications have limited life spans. If iOS ever needs more memory and your application is in the background, Apple might kill it to return memory to the system. This should be transparent to your users; they should always return to the last spot they were within the application.

To achieve this transparency, you must adopt *state restoration* within your application. State restoration works a lot like another technology you have used to persist data – archiving. When the application is suspended, a snapshot of the view controller hierarchy is saved. If the application was killed before the user opens it again, its state will be restored upon launch. (If the application has not been killed, then everything is still in memory and you have no need to restore any state.)

In this chapter, you will add state restoration to the Homepwner application.

Let’s start by demonstrating the need for state restoration. Open the Homepwner project. Create a new item and drill down to its detail screen (the `BNRDetailViewController`). Now you need to simulate the process that triggers state restoration. In the iOS Simulator, press the Home button (or use the keyboard shortcut Command-Shift-H). This will put the application into the background. Now, to kill the application as if the system killed it, go back to Xcode and click the stop button (Command-.). Then relaunch the application from Xcode.

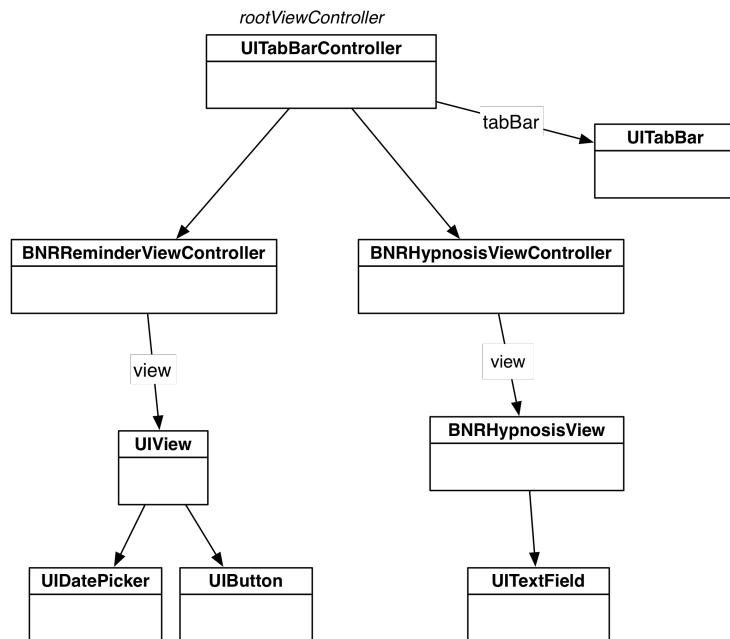
When the application launches, you will briefly see the `BNRDetailViewController`, but that will quickly be replaced with the `BNRItemsViewController`. Why is this? When applications go into the background, iOS takes a snapshot image of the user interface. Then when the application is relaunched, this snapshot is used as the launch image until the application is loaded into memory.

If an application does not implement state restoration, the user will briefly see the previous application state, but then the screen will change to its fresh-launch state. With Homepwner, you see a flicker of the detail view controller and then the items view controller. This is a disorienting experience.

How State Restoration Works

A running app can be thought of as a tree of view controllers and views, with the root view controller as the root of the tree. For example, the interface of your HypnoNerd app can be thought of as a tree like this:

Figure 24.1 An app is a tree



If you opt-in to state restoration, before your app is terminated, the system walks this tree asking each node, “What is your name?”, “What is your class?”, and “Do you have any data you want me to hold on to?” While the app is dead, this description of the tree is stored on the file system.

A node’s name, actually called the *restoration identifier*, is typically the object’s class name. The *restoration class* is typically the class of the object. The data holds the state of the object. For example, the data for a tab view controller includes which tab is currently selected.

When the app is restarted, the system tries to recreate the tree of view controllers and views from that saved description. For each node:

- The system asks the restoration class to create a new view controller for that node.
- The new node is given an array of restoration identifiers: The identifier for the node being created and the identifiers for all of its ancestors. The first identifier in the array is the identifier for the root node. The last is the identifier for the node being recreated.
- The new node is given its state data. This data comes in the form of an **NSCoder**, which you used in Chapter 18.

In this chapter, you are going to extend the view controllers in Homeowner to properly give their node information when the app is terminating and to use that node information when they are resurrected.

Opting In to State Restoration

State restoration is disabled by default in applications. To enable state restoration, you must opt in within the application delegate.

Open `BNRAppDelegate.m` and implement the two delegate methods to enable state saving and restoration.

```
@implementation BNRAppDelegate

- (BOOL)application:(UIApplication *)application
    shouldSaveApplicationState:(NSCoder *)coder
{
    return YES;
}

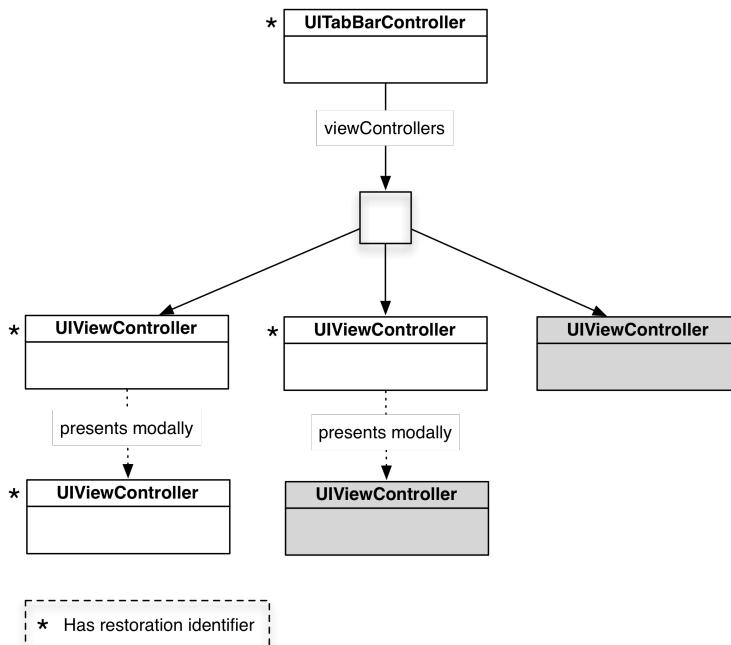
- (BOOL)application:(UIApplication *)application
    shouldRestoreApplicationState:(NSCoder *)coder
{
    return YES;
}
```

When the application goes into the background, its state will attempt to be saved, and if the app is launching fresh, its save will attempt to be restored. To understand what gets stored, we need to discuss *restoration identifiers*.

Restoration Identifiers and Classes

When the application's state is being saved, the window's `rootViewController` is asked for its `restorationIdentifier`. If it has a `restorationIdentifier`, it is asked to save (or *encode*) its state. Then it is responsible for processing its child view controllers in the same way, and they, in turn, pass the torch to *their* child view controllers. If any view controller in the hierarchy does not have a `restorationIdentifier`, however, it (and its child view controllers, whether or not they have `restorationIdentifiers`) will be excluded from the saved state.

Figure 24.2 Restoration identifiers



For example, with the application shown in Figure 24.2, the state of the two gray view controllers – and any child view controllers they might have – would not be saved.

Typically, the restoration identifier for a class is the same name as the class itself.

In `BNRAppDelegate.m`, give the navigation controller a restoration identifier in `application:didFinishLaunchingWithOptions:`.

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:UIScreen.mainScreen.bounds]];
    // Override point for customization after application launch

    BNRItemsViewController *itemsViewController
        = [[BNRItemsViewController alloc] init];

    // Create an instance of a UINavigationController
    // Its stack contains only itemsViewController
    UINavigationController *navController = [[UINavigationController alloc]
        initWithRootViewController:itemsViewController];

    // Give the navigation controller a restoration identifier that is
    // the same name as the class
    navController.restorationIdentifier = NSStringFromClass([navController class]);

    // Place navigation controller's view in the window hierarchy
    self.window.rootViewController = navController;

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];

    return YES;
}
```

Now that the navigation controller has a restoration identifier, it will attempt to save the state of its `viewControllers` if they have restoration identifiers themselves.

For your two `UIViewController` subclasses (`BNRItemsViewController` and `BNRDetailViewController`), you will assign the restoration identifier within that class's designated initializer. Additionally, you will set the view controller's *restoration class*. When the view controller's state is being restored, it will ask this restoration class for an instance of the view controller to restore.

Open `BNRItemsViewController.m` and update `init` to assign the restoration identifier and class.

```
- (instancetype)init
{
    // Call the superclass's designated initializer
    self = [super initWithStyle:UITableViewStylePlain];
    if (self) {
        UINavigationItem *navItem = self.navigationItem;
        navItem.title = @"Homepwner";

        self.restorationIdentifier = NSStringFromClass([self class]);
        self.restorationClass = [self class];
}
```

Open `BNRDetailViewController.m` and give the view controller a restoration identifier and class in `initForNewItem:`.

```
- (instancetype)initForNewItem:(BOOL)isNew
{
    self = [super initWithNibName:nil bundle:nil];
    if (self) {
        self.restorationIdentifier = NSStringFromClass([self class]);
        self.restorationClass = [self class];
    }
    if (isNew) {
```

Finally, there is one more view controller that is created in Homeowner: the **UINavigationController** that is presented modally when a new item is created.

Reopen `BNRItemsViewController.m` and update `addNewItem:` to give the **UINavigationController** a restoration identifier.

```
- (IBAction)addNewItem:(id)sender
{
    // Create a new BNRItem and add it to the store
    BNRItem *newItem = [[BNRItemStore sharedStore] createItem];

    BNRDetailViewController *detailViewController =
        [[BNRDetailViewController alloc] initForNewItem:YES];

    detailViewController.item = newItem;
    detailViewController.dismissBlock = ^{
        [self.tableView reloadData];
    };

    UINavigationController *navController = [[UINavigationController alloc]
        initWithRootViewController:detailViewController];
    navController.restorationIdentifier = NSStringFromClass([navController class]);
```

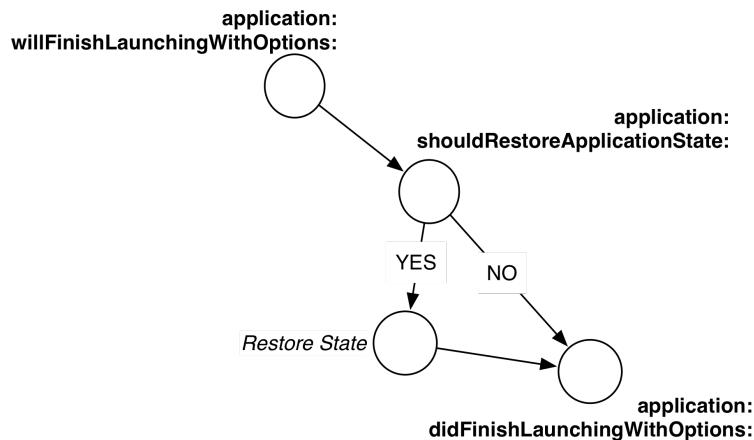
(The two instances of **UINavigationController** that Homeowner uses do not have restoration classes. Because of this, the application delegate will be asked to create new instances of these view controllers, as you will see shortly.)

Now that all of the view controllers in Homeowner have a restoration identifier, their state will be saved when the user leaves the application.

State Restoration Life Cycle

Now that you are working with state restoration, the application life cycle is going to be a bit different, as you can see in Figure 24.3. Currently, all of your `window` and `rootViewController` code is in `application:didFinishLaunchingWithOptions:`, but with state restoration, it will spread out a bit.

Figure 24.3 Restoration life cycle



The method **application:willFinishLaunchingWithOptions:**: gets called before state restoration has begun. You should use this method to set up the window and do anything that should happen before state restoration.

In `BNRAppDelegate.m`, override this method to initialize and set up the window.

```
- (BOOL)application:(UIApplication *)application  
willFinishLaunchingWithOptions:(NSDictionary *)launchOptions  
{  
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];  
    self.window.backgroundColor = [UIColor whiteColor];  
  
    return YES;  
}
```

Next, update **application:didFinishLaunchingWithOptions:**: to set up the view controller hierarchy in case state restoration did not occur (for example, on the first launch of the application). Also, remove the code that is now in **application:willFinishLaunchingWithOptions:**:

```

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen.bounds]];
    // Override point for customization after application launch.

    // If state restoration did not occur,
    // set up the view controller hierarchy
    if (!self.window.rootViewController) {
        BNRItemsViewController *itemsViewController
            = [[BNRItemsViewController alloc] init];

        // Create an instance of a UINavigationController
        // Its stack contains only itemsViewController
        UINavigationController *navController = [[UINavigationController alloc]
            initWithRootViewController:itemsViewController];

        // Give the navigation controller a restoration identifier that is
        // the same name as the class
        navController.restorationIdentifier = NSStringFromClass([navController class]);

        // Place navigation controller's view in the window hierarchy
        self.window.rootViewController = navController;
    }

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];

    return YES;
}

```

Restoring View Controllers

Since the two view controllers have a restoration class, the restoration class will be asked to create new instances of their respective view controller. In `BNRItemsViewController.h`, have the view controller conform to the `UIViewControllerRestoration` protocol.

```
@interface BNRItemsViewController : UITableViewController <UIViewControllerRestoration>
@end
```

Then, in `BNRItemsViewController.m`, implement the one required method for this protocol, which will return a new view controller instance.

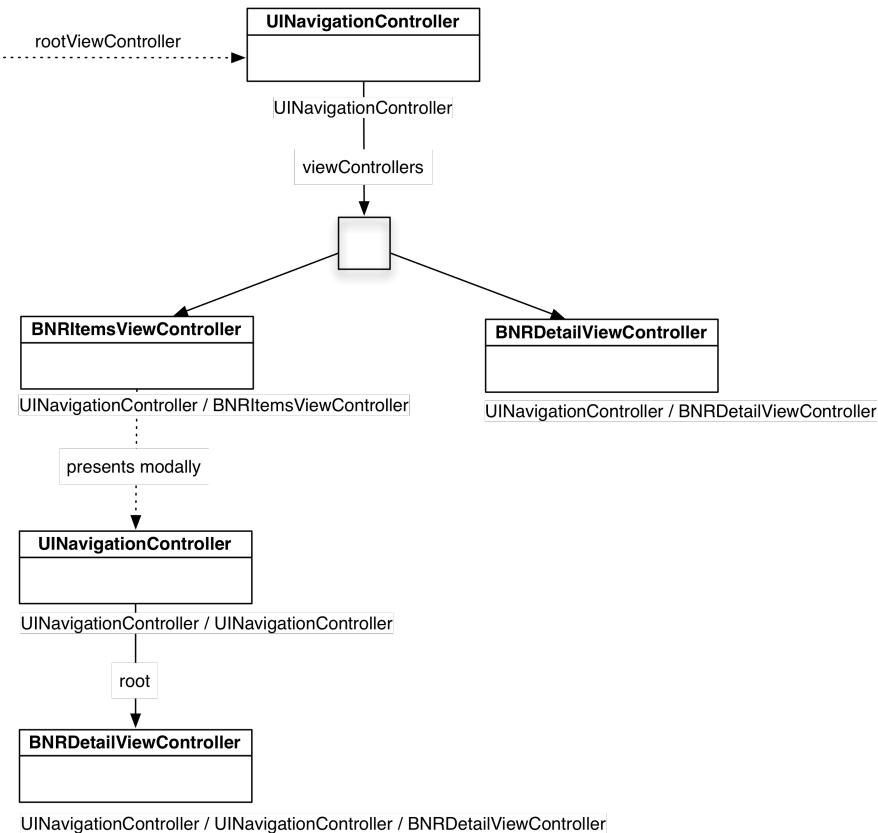
```
+ (UIViewController *)viewControllerWithRestorationIdentifierPath:(NSArray *)path
                                                       coder:(NSCoder *)coder
{
    return [[self alloc] init];
}
```

Now do the same for `BNRDetailViewController`. Open `BNRDetailViewController.h` and have it conform to the `UIViewControllerRestoration` protocol.

```
@interface BNRDetailViewController : UIViewController <UIViewControllerRestoration>
```

Implementing the protocol's required method for **BNRDetailViewController** is a bit trickier because you need to know whether to pass YES or NO to `initForNewItem:`. Thankfully, the restoration identifier path argument can help.

Figure 24.4 Restoration path



The restoration identifier path is an array of restoration identifiers that represents this view controller and its ancestors at the time the view controller's state was saved. Figure 24.4 shows the restoration paths for the different code paths of the **Homepwner** application. One path that might look odd is the restoration path of the **UINavigationController** that is presented modally. When the **BNRItemsViewController** presents the navigation controller modally, it actually gets presented from the parent of **BNRItemsViewController**, which is the root **UINavigationController**. Therefore, "BNRItemsViewController" is not in the path of the modally presented view controller's restoration identifier path array.

Armed with this knowledge, you know that the count of the restoration identifier path array will be 2 if you are viewing an existing **BNRItem** and 3 if you are creating a new **BNRItem**.

Open **BNRDetailViewController.m** and implement the one required method of the **UIViewControllerAnimatedProtocol**.

```
+ (UIViewController *)viewControllerWithRestorationIdentifierPath:(NSArray *)path
                                                       coder:(NSCoder *)coder
{
    BOOL isNew = NO;
    if ([path count] == 3) {
        isNew = YES;
    }

    return [[self alloc] initForNewItem:isNew];
}
```

Now the **BNRDetailViewController** will have the correct bar button items when its state is restored.

You have taken care of the **BNRItemsViewController** and **BNRDetailViewController**, but there are still two more view controllers that will need to be restored – the two navigation controllers that the application uses. You have given both of these view controllers restoration identifiers, but you will not give them a restoration class.

Instead, if a view controller that is getting restored does not have a restoration class, the application delegate is asked to provide the view controller. Open `BNRAppDelegate.m` and implement this method.

```
- (UIViewController *)application:(UIApplication *)application
viewControllerWithRestorationIdentifierPath:(NSArray *)identifierComponents
                                       coder:(NSCoder *)coder
{
    // Create a new navigation controller
    UIViewController *vc = [[UINavigationController alloc] init];

    // The last object in the path array is the restoration
    // identifier for this view controller
    vc.restorationIdentifier = [identifierComponents lastObject];

    // If there is only 1 identifier component, then
    // this is the root view controller
    if ([identifierComponents count] == 1) {
        self.window.rootViewController = vc;
    }

    return vc;
}
```

Build and run the application. Create an item and drill down to see its details. Trigger state restoration as you did at the beginning of this chapter. After relaunching the application, you will be returned to the **BNRDetailViewController** – but the item details are blank. Although the view controller hierarchy is currently being saved, no model information is saved, so the **BNRDetailViewController** has no idea which **BNRItem** it should be displaying. To fix this, you will need to manually encode a reference to the **BNRItem** being displayed.

Encoding Relevant Data

To persist other information, a **UIViewController** is given a chance to encode any relevant data in a process very similar to archiving. In fact, both encoding processes use an **NSCoder** object to do the work. You will use this to save out any necessary information in the view controller subclasses.

In `BNRDetailViewController.m`, encode the `itemKey` for the currently displayed **BNRItem**.

```
- (void)encodeRestorableStateWithCoder:(NSCoder *)coder
{
    [coder encodeObject:self.item.itemKey
        forKey:@"item.itemKey"];
    [super encodeRestorableStateWithCoder:coder];
}
```

Then implement the decoding method to search through the **BNRItemStore** for the appropriate **BNRItem**.

```
- (void)decodeRestorableStateWithCoder:(NSCoder *)coder
{
    NSString *itemKey =
    [coder decodeObjectForKey:@"item.itemKey"];
    for (BNRItem *item in [[BNRItemStore sharedStore] allItems]) {
        if ([itemKey isEqualToString:item.itemKey]) {
            self.item = item;
            break;
        }
    }
    [super decodeRestorableStateWithCoder:coder];
}
```

Build and run the application and drill down into a **BNRItem**. Then perform the state restoration steps again. This time, the values on the **BNRDetailViewController** will correctly reflect the **BNRItem** being displayed.

There is still one problem: the text fields and labels are being populated with the values of the **BNRItem**. If the user has typed in some other value, those values will be lost upon state restoration.

You will fix this by saving the current text field values into the **BNRItem**. Since view controller encoding occurs after the application has entered the background, you will also need to save the store again.

In **BNRDetailViewController.m** update the encode method.

```
- (void)encodeRestorableStateWithCoder:(NSCoder *)coder
{
    [coder encodeObject:self.item.itemKey
        forKey:@"item.itemKey"];
    // Save changes into item
    self.item.itemName = self.nameField.text;
    self.item.serialNumber = self.serialNumberField.text;
    self.item.valueInDollars = [self.valueField.text intValue];
    // Have store save changes to disk
    [[BNRItemStore sharedStore] saveChanges];
    [super encodeRestorableStateWithCoder:coder];
}
```

The **BNRDetailViewController** is now saving and restoring its state beautifully. Without a lot of work on your part, your users get a better experience. Now let's turn our attention to the **BNRItemsViewController**.

Saving View States

There are a number of problems with the **BNRItemsViewController**:

- The **UITableView** is not being restored to its existing scroll position, nor is the currently selected row (if the user has drilled down into an item) being restored.
- The view controller does not save whether or not it is in editing mode, and so it is always restored with the default value (which is not in editing mode).

You will fix the second point soon very similarly to how you encoded information for the **BNRDetailViewController**, but before you do let's take a look at the first point.

In addition to the view controllers having a restoration identifier, certain **UIView** subclasses can have a restoration identifier to save certain information about the view. Specifically, these subclasses can save some of their information: **UICollectionView**, **UIImageView**, **UIScrollView**, **UITableView**, **UITextField**, **UITextView**, and **UIWebView**.

The documentation for each of those classes states what information is preserved. For **UITableView**, the useful piece of information saved is the content offset of the table view (the scroll position).

In **BNRItemsViewController.m**, give the **UITableView** a restoration identifier.

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    // Load the NIB file
    UINib *nib = [UINib nibWithNibName:@"BNRItemCell" bundle:nil];

    // Register this NIB which contains the cell
    [self.tableView registerNib:nib
        forCellReuseIdentifier:@"BNRItemCell"];

    self.tableView.restorationIdentifier = @"BNRItemsViewControllerTableView";
}
```

Build and run the application and scroll down in the **BNRItemsViewController** (you will probably need to add some new items). Trigger state restoration and, upon relaunching, the table view will return to its previous content offset.

Let's turn our attention to the editing mode of the view controller so its state persists. In **BNRItemsViewController.m**, implement the encode and decode methods.

```
- (void)encodeRestorableStateWithCoder:(NSCoder *)coder
{
    [coder encodeBool:self.isEditing forKey:@"TableViewIsEditing"];

    [super encodeRestorableStateWithCoder:coder];
}

- (void)decodeRestorableStateWithCoder:(NSCoder *)coder
{
    self.editing = [coder decodeBoolForKey:@"TableViewIsEditing"];

    [super decodeRestorableStateWithCoder:coder];
}
```

You have one last issue to tackle to finish state restoration in the Homepwner application. **UITableView** will save information about the selected **UITableViewCell** row, but you need to help it out a little.

In **BNRItemsViewController.m**, have the view controller conform to the **UIDataSourceModelAssociation** protocol in the class extension.

```
@interface BNRItemsViewController ()  
    <UIPopoverControllerDelegate, UIDataSourceModelAssociation>  
@property (nonatomic, strong) UIPopoverController *imagePopover;  
@end
```

The **UIDataSourceModelAssociation** protocol helps state restoration locate the appropriate model objects for your application. When the application state is being saved, state restoration will ask for a unique identifier for the model object associated with the selected row or rows (a **BNRItem**, for example). When the application is relaunched, state restoration will provide the identifier and ask for an index path for that model object. Model objects may change positions in the **UITableView** on relaunch, but as long as your mapping is correct, the correct rows will be selected.

In **BNRItemsViewController.m**, implement the method to provide state restoration with a unique identifier for the selected **BNRItem**. You will use the **itemKey** property as the unique identifier.

```
- (NSString *)modelIdentifierForElementAtIndexPath:(NSIndexPath *)path  
                                         inView:(UIView *)view  
{  
    NSString *identifier = nil;  
  
    if (path && view) {  
        // Return an identifier of the given NSIndexPath,  
        // in case next time the data source changes  
        BNRItem *item = [[BNRItemStore sharedStore] allItems][path.row];  
        identifier = item.itemKey;  
    }  
  
    return identifier;  
}
```

Then implement the inverse method that returns an **NSIndexPath** for a given identifier.

```
- (NSIndexPath *)indexPathForElementWithModelIdentifier:(NSString *)identifier  
                                         inView:(UIView *)view  
{  
    NSIndexPath *indexPath = nil;  
  
    if (identifier && view) {  
        NSArray *items = [[BNRItemStore sharedStore] allItems];  
        for (BNRItem *item in items) {  
            if ([identifier isEqualToString:item.itemKey]) {  
                int row = [items indexOfObjectIdenticalTo:item];  
                indexPath = [NSIndexPath indexPathForRow:row inSection:0];  
                break;  
            }  
        }  
    }  
  
    return indexPath;  
}
```

Build and run the application and trigger state restoration. Homepwner is now fully set up for state restoration and will give users a good experience.

Silver Challenge: Another Application

Implement state restoration in the HypnoNerd application. The process will be nearly identical to what you did in this chapter. Make sure to encode the `UITextField` text and the currently selected date on the `UIDatePicker`.

(Hint: to completely finish this challenge, the `BNRHypnosisView` should also save and restore the labels that have been added to it. You will want to give the `BNRHypnosisView` a restoration identifier and implement the appropriate encode and decode methods.)

For the More Curious: Controlling Snapshots

As we mentioned in this chapter, the system takes a snapshot of the application when it goes into the background. Sometimes you may want to have some control over what is displayed to your users on the next launch (which is also what your users will see when viewing your application in the multitasking display).

For example, if your application displays sensitive information (such as a banking application showing account numbers and balances), you may want to hide this information so that it is not shown to unauthorized eyes. As another example, Apple blurs the camera contents when the Camera app goes into the background, not for security reasons, but to make it easier for users to notice the Camera app in the multitasking display (instead of getting distracted by whatever the camera was looking at when the app went into the background).

Modifying the snapshot is easy: you update the view before the snapshot is taken to have it display what you want the snapshot to be.

We talked about the various states that the application goes through in Chapter 18 and implemented the application delegate callbacks to see the state changes. In addition to the application delegate callbacks, the application also posts notifications when its state is transitioning. Observing the appropriate notifications in your view controllers will give you an opportunity to update the user interface.

Specifically, you will want to observe the `UIApplicationWillResignActiveNotification` to obscure anything necessary, and `UIApplicationDidBecomeActiveNotification` to get things ready for the user to see again.

```
NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
[nc addObserver:self
    selector:@selector(applicationResigningActive:)
    name:UIApplicationWillResignActiveNotification
    object:nil];
[nc addObserver:self
    selector:@selector(applicationBecameActive:)
    name:UIApplicationDidBecomeActiveNotification
    object:nil];

- (void)applicationResigningActive:(NSNotification *)note
{
    // Prepare app for snapshot
}

- (void)applicationBecameActive:(NSNotification *)note
{
    // Undo any changes before user returns to app
}
```

Finally, if your application implements state restoration but for some reason it does not make sense to use a snapshot the next time it launches, you can tell the application to ignore the snapshot:

```
// This method should be called during the code that preserves the state  
[[UIApplication mainApplication] ignoreSnapshotOnNextApplicationLaunch];
```

This might make sense, for example, if your application was displaying a network connectivity error message. The user may very well have a network connection the next time the application launches, and so to reduce user confusion you would ignore the snapshot for just the next launch of the application. The application would use the launch image in its place.

25

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 25.1 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.)

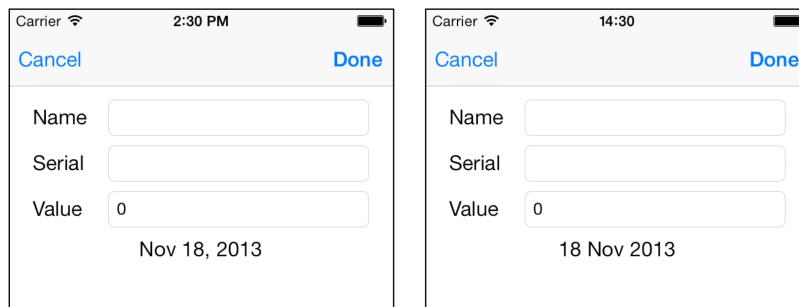
Internationalization Using `NSNumberFormat`

In this first section, you will use the class `NSNumberFormat` to internationalize the number format and currency symbol for the value of an item.

Did you know that `Homepwner` is already partially internationalized? Launch the application and add a new item. The date label in the `BNRDetailViewController` is formatted according to the current regional settings. In the US, the dates are displayed as *Month Day, Year*. Cancel adding a new item.

Now open the `Settings` application and change Region Format to United Kingdom (`General` → `International` → `Region Format`). Switch back to `Homepwner` and add a new item again. 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 25.2 Date format: US vs UK



In Chapter 10, you used an instance of `NSDateFormatter` to set the text of the date label of `BNRDetailViewController`. `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 `Region Format` 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.)

```
NSLocale *locale = [NSLocale currentLocale];
BOOL isMetric = [[locale objectForKey:NSLocaleUsesMetricSystem] boolValue];
NSString *currencySymbol = [locale objectForKey:NSLocaleCurrencySymbol];
```

Let's internationalize the value amount displayed in each `BNRItemCell`. Open `Homepwner.xcodeproj`.

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.

```
NSNumberFormatter *numberFormatter = [[NSNumberFormatter alloc] init];
NSString *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 **NSNumberFormatterCurrencyStyle**, 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.)

```
NSNumberFormatter *currencyFormatter = [[NSNumberFormatter alloc] init];
currencyFormatter.numberStyle = NSNumberFormatterCurrencyStyle;
NSString *numberAsString = [currencyFormatter stringFromNumber:@123456.789];
```

In **BNRItemsViewController.m**, locate the method **tableView:cellForRowAtIndexPath:**. Add the static variable **currencyFormatter** and set its **numberStyle** to **NSNumberFormatterCurrencyStyle**.

```
cell.serialNumberLabel.text = item.serialNumber;
// Create a number formatter for currency
static NSNumberFormatter *currencyFormatter = nil;
if (currencyFormatter == nil) {
    currencyFormatter = [[NSNumberFormatter alloc] init];
    currencyFormatter.numberStyle = NSNumberFormatterCurrencyStyle;
}
cell.valueLabel.text = [NSString stringWithFormat:@"$%d",
                      item.valueInDollars];
```

When the text of the cell's **valueLabel** is set in this method, the string "**\$%d**" 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
static NSNumberFormatter *currencyFormatter = nil;
if (currencyFormatter == nil) {
    currencyFormatter = [[NSNumberFormatter alloc] init];
    currencyFormatter.numberStyle = NSNumberFormatterCurrencyStyle;
}
cell.valueLabel.text = [NSString stringWithFormat:@"$%d",
                      item.valueInDollars];
cell.valueLabel.text = [currencyFormatter
                      stringFromNumber:@(item.valueInDollars)];

cell.thumbnailView.image = item.thumbnail;
```

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 should be United Kingdom if you followed the instructions at the beginning of this section. In the **Settings** application, change Region Format back to United States (**General** → **International** → **Region Format**). Return to Homepwner.

You were probably expecting to see values displayed in dollars (\$). However, it did not happen. To trigger the update to the table view, start adding a new item and immediately cancel. Now, you will see the values formatted correctly, because `viewWillAppear:` was called and it reloaded the table view. (Note that this is not a currency conversion; you are just replacing the symbol.)

To make Homepwner update when the region settings change, you need to use `NSNotificationCenter`. In `BNRItemsViewController`'s `init` method, register for locale change notifications:

```
NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
[nc addObserver:self
    selector:@selector(updateTableViewForDynamicTypeSize)
    name:UIContentSizeCategoryDidChangeNotification
    object:nil];

// Register for locale change notifications
[nc addObserver:self
    selector:@selector(localeChanged:)
    name:NSCurrentLocaleDidChangeNotification
    object:nil];
}

return self;
```

Add the method `localeChanged`.

```
- (void)localeChanged:(NSNotification *)note
{
    [self.tableView reloadData];
}
```

Build and run the application. Now, when you change the regional settings and return to Homepwner, the table view will be reloaded and the value label will display the amount properly formatted. To see why you used the number formatter instead of just retrieving the currency symbol, change the region to Germany. Not only did the currency symbol change, but also a few other things: the position of the currency symbol (after the number, instead of before), spacing (one space between the amount and the currency symbol, instead of no spaces), decimal mark (a comma instead of a dot), and thousands separator (a dot instead of a comma).

Figure 25.3 Number format: US vs UK vs Germany

| Carrier | | | 3:46 PM | |
|---------|----------------------------------|------------|---------|--|
| Edit | Homepwner | + | | |
| | iPhone 5S 1BA024 | \$700.00 | | |
| | Apple TV 7194ABFED | \$99.00 | | |
| | iPad mini 99B07F | \$399.00 | | |
| | MacBook Pro Retina 2013MBPR13 | \$1,799.00 | | |

| Carrier | | | 15:46 | |
|---------|----------------------------------|-----------|-------|--|
| Edit | Homepwner | + | | |
| | iPhone 5S 1BA024 | £700.00 | | |
| | Apple TV 7194ABFED | £99.00 | | |
| | iPad mini 99B07F | £399.00 | | |
| | MacBook Pro Retina 2013MBPR13 | £1,799.00 | | |

| Carrier | | | 15:47 | |
|---------|----------------------------------|------------|-------|--|
| Edit | Homepwner | + | | |
| | iPhone 5S 1BA024 | 700,00 € | | |
| | Apple TV 7194ABFED | 99,00 € | | |
| | iPad mini 99B07F | 399,00 € | | |
| | MacBook Pro Retina 2013MBPR13 | 1.799,00 € | | |

Localizing Resources

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.

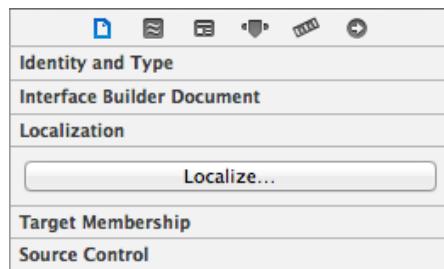
To simplify the process of localizing XIB files, Xcode has a feature called Base internationalization. When it is enabled for the project, Base internationalization creates the `Base.lproj` directory which contains the main XIB files. Localizing individual XIB files can then be done by creating just 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 Homeowner's interfaces: the `BNRDetailViewController.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 `BNRDetailViewController.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 25.4).

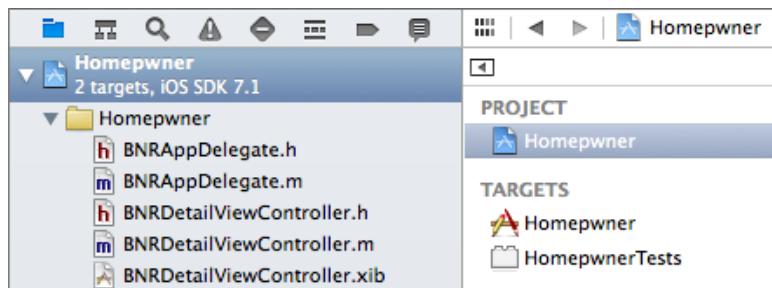
Figure 25.4 Localizing BNRDetailViewController.xib, beginning



Select English. This signifies to Xcode that this file can be localized, automatically creates en.lproj, and moves the BNRDetailViewController.xib file to it.

Now you need to enable Base Internationalization. Select the project file as shown in Figure 25.5. Make sure you select the project Homepwner, and not the target Homepwner.

Figure 25.5 Selecting Project Info

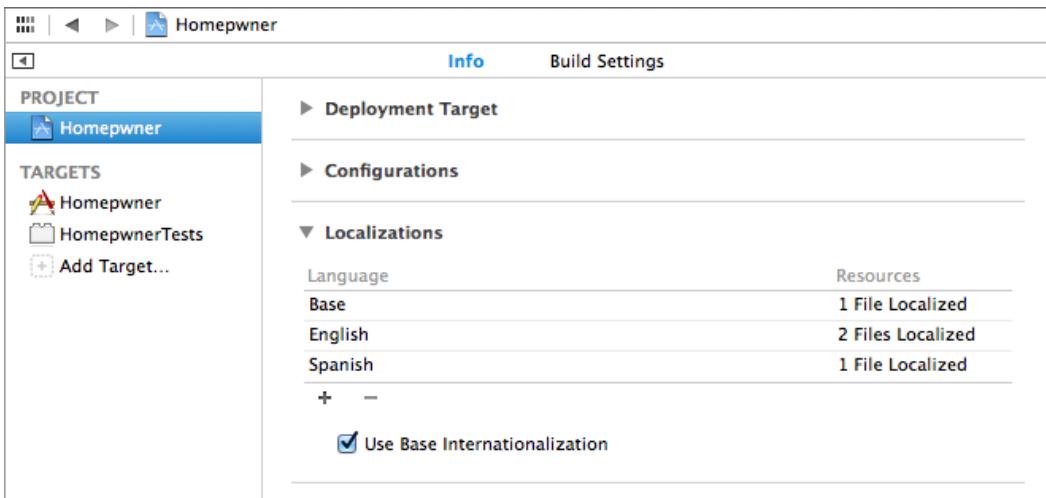


In the bottom section of the Info tab of the project, locate the Use Base Internationalization checkbox in the Localizations section and check it. You will see the prompt to select which files will be used to create the Base localization; the table will consist of just BNRDetailViewController.xib and English will be listed as the reference language. Click Finish.

This will create the Base.lproj directory and move BNRDetailViewController.xib to it.

Click the + button under the list of languages and select Spanish. In the dialog, you can uncheck the InfoPlist.strings files and only keep the BNRDetailViewController.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 BNRDetailViewController.strings in it that contains all the strings from the base XIB file. The Localizations configuration should look like Figure 25.6.

Figure 25.6 Localizations

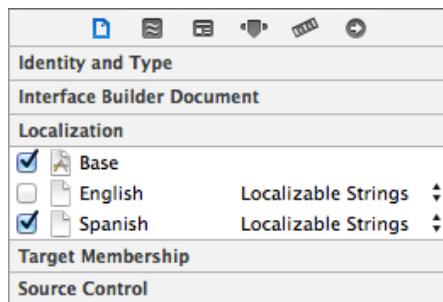


Look in the project navigator. Click the disclosure button next to `BNRDetailViewController.xib` (Figure 25.7). Xcode moved the `BNRDetailViewController.xib` file to the `Base.lproj` directory and created the `BNRDetailViewController.strings` file in the `es.lproj` directory.

Figure 25.7 Localized XIB in the project navigator



Select the `BNRDetailViewController.xib`. It does not matter if you select the top level one or the one marked `Base`. The file inspector should look like Figure 25.8.

Figure 25.8 Localizing `BNRDetailViewController.xib`, result

In the project navigator, click the Spanish version of `BNRDetailViewController.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.

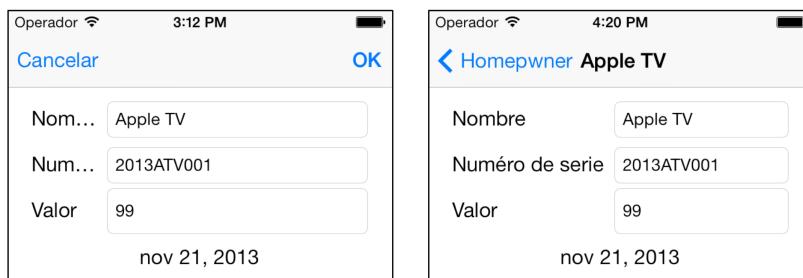
```
/* Class = "IBUILabel"; text = "Serial"; ObjectID = "JkL-nP-h3R"; */  
"JkL-nP-h3R.text" = "Número de serie";  
  
/* Class = "IBUILabel"; text = "Label"; ObjectID = "Q5n-Bc-7IH"; */  
"Q5n-Bc-7IH.text" = "Label";  
  
/* Class = "IBUILabel"; text = "Name"; ObjectID = "qzL-Fn-qch"; */  
"qzL-Fn-qch.text" = "Nombre";  
  
/* Class = "IBUILabel"; text = "Value"; ObjectID = "rhE-7e-oTE"; */  
"rhE-7e-oTE.text" = "Valor";  
  
/* Class = "IBUIBarButtonItem"; title = "Item"; ObjectID = "uNg-wM-Zcr"; */  
"uNg-wM-Zcr.title" = "Item";
```

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 alt/option button 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.

Homepwner's detail view will not appear in Spanish until you change the language settings on the device or simulator. In **Settings**, change the language settings to **Español** (**General** → **International** → **Language**) and then relaunch your application. Select an item row, and you will see the interface in Spanish. However, the labels are being cut off (Figure 25.9).

Figure 25.9 Spanish `BNRDetailViewController.xib`, before and after layout fix



Fortunately, thanks to Auto Layout, this is an easy one to fix. Open the `BNRDetailViewController.xib` file and select the name label. In the utility area, select the size inspector (the tab) and locate the width constraint. Currently, the width is set to be equal to 55,

which is too short for the longer Spanish labels. Click on the drop-down next to the gear button of the width constraint and choose Select and Edit.... Change the relation from Equal to Greater Than or Equal.

Now, you will fix the widths of the serial number and value labels. Select the serial number label and locate its width constraint. Instead of editing it, just delete it. You want the text fields to be of the same size and the way to achieve this is to make the labels the same widths. After you deleted the width constraint, control-click on the serial number label and drag to the name label. In the menu, select the Equal widths item. Repeat the same steps for the value label. Build and run the application. This time, the text fields will resize to make space for the labels, so that they will not be cut off.

NSLocalizedString() and Strings Tables

In many places in your applications, you create **NSString** 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:

```
NSString *greeting = @"Hello!"
```

To internationalize the string in your code, you replace literal strings with the function **NSLocalizedString**.

```
NSString *greeting = NSLocalizedString(@"Hello!", @"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 **BNRItemsViewController.m**, locate the **init** method and change the line of code that sets the title of the **navigationItem**.

```
- (instancetype)init
{
    // Call the superclass's designated initializer
    self = [super initWithStyle:UITableViewStylePlain];
    if (self) {
        UINavigationItem *navItem = [self navigationItem];
        navItem.title = @"Homepwner";
        navItem.title = NSLocalizedString(@"Homepwner", @"Name of application");
    }
}
```

Two more view controllers contain hard-coded strings that can be internationalized. The toolbar in the **BNRDetailViewController** shows the asset type. The title of the **BNRAssetTypeViewController** needs to be updated just like the title of the **BNRItemsViewController**.

In **BNRDetailViewController.m**, update the **viewWillAppear:** method:

```
NSString *typeLabel = [self.item.assetType valueForKey:@"label"];
if (!typeLabel) {
    typeLabel = @"None";
    typeLabel = NSLocalizedString(@"None", @"Type label None");
}
self.assetTypeButton.title = [NSString stringWithFormat:@"Type: %@", typeLabel];
self.assetTypeButton.title = [NSString stringWithFormat:
    NSLocalizedString(@"Type: %@", @"Asset type button"), typeLabel];

[self updateFonts];
}
```

In `BNRAssetTypeViewController.m`, update the `init` method:

```
if (self) {
    self.navigationItem.title = @"Asset Type";
    self.navigationItem.title =
        NSLocalizedString(@"Asset Type", @"BNRAssetTypeViewController title");
}
return self;
}
```

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 `BNRItemsViewController.m`. 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 `BNRItemsViewController.m` 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/aaron/Homepwner/Homepwner/
```

Use the terminal command `ls` to print out the contents of the working directory and confirm that `BNRItemsViewController.m` is in that list.

To generate the strings table, enter the following into Terminal and press Enter:

```
genstrings BNRItemsViewController.m
```

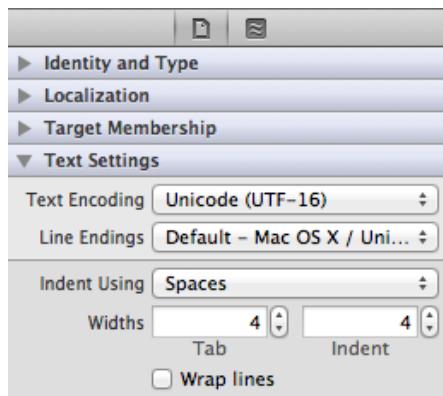
This creates a file named `Localizable.strings` in the same directory as `BNRItemsViewController.m`. Now you need to generate strings from the other two view controllers. Since the file `Localizable.strings` already exists, you will want to append to it, rather than create it from scratch. To do so, enter the following commands in the Terminal (do not forget the `-a` command line option) and press Enter after each line:

```
genstrings -a BNRDetailViewController.m
genstrings -a BNRAssetTypeViewController.m
```

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.

Oddly enough, Xcode sometimes has a problem with strings tables. Open the Localizable.strings file in the editor area. If you see a bunch of upside-down question marks, you need to reinterpret this file as Unicode (UTF-16). Show the utility area and select the file inspector. Locate the area named Text Settings and change the pop-up menu next to Text Encoding to Unicode (UTF-16) (Figure 25.10). It will ask if you want to reinterpret or convert. Choose Reinterpret.

Figure 25.10 Changing encoding of a file



The file should look like this:

```
/* Name of application */
"Homepwner" = "Homepwner";

/* BNRAAssetTypeViewController title */
"Asset Type" = "Asset Type";

/* Type label None */
"None" = "None";

/* Asset type button */
"Type: %@", "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. Add the Spanish localization and then 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"

/* AssetTypePicker title */
"Asset Type" = "Tipo de activo";

/* Type label None */
"None" = "Nada";

/* Asset type button */
"Type: %@", "Tipo: %@";
```

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.)

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`:

```
NSString *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: Localizing XIB files without Base Internationalization

Before Xcode had the Base internationalization feature, the localizable strings option was not available. Instead, you would maintain a XIB file for every locale that you wanted to support. So, you would have an `en.lproj/BNRDetailViewController.xib` and an `es.lproj/BNRDetailViewController.xib`. As you can imagine, maintaining every XIB in every language you wanted to support was a hassle.

To help with the creation and maintenance of localized XIB files, you could use a command-line tool named `ibtool` to suck the strings from your native language XIB file into a strings file. Then, you would translate these strings and create a new XIB file for each language.

To give it a try, open Terminal and navigate to the `en.lproj` directory. For example, my terminal command looks like this:

```
cd /Users/aaron/Homepwner/Homepwner/en.lproj
```

Next, use `ibtool` to suck the strings from this XIB file. Enter the following terminal command (all on one line) and enter it. (We only broke it up so that it would fit on the page.)

```
ibtool --export-strings-file ~/Desktop/BNRDetailViewController.strings  
BNRDetailViewController.xib
```

This will create a `BNRDetailViewController.strings` file on your desktop that contains all of the strings in your XIB file. Open the Spanish `BNRDetailViewController.strings`. This is the same file as the one that Xcode created using localizable strings. Edit this file like before.

Now you will use `ibtool` to create a new Spanish XIB file. This file will be based on the English version of `BNRDetailViewController.xib` but will replace all of the strings with the values from `BNRDetailViewController.strings`. To pull this off, you need to know the path of your English XIB file and the path of your Spanish directory in this project's directory. Remember, you opened these windows in Finder earlier.

In Terminal.app, enter the following command, followed by a space after `write`. (But do not press Enter yet!)

```
ibtool --import-strings-file ~/Desktop/BNRDetailViewController.strings --write
```

Next, find `BNRDetailViewController.xib` in `es.lproj` and drag it onto the terminal window. Then, find `BNRDetailViewController.xib` in the `en.lproj` folder and drag it onto the terminal window. Your command should look similar to this:

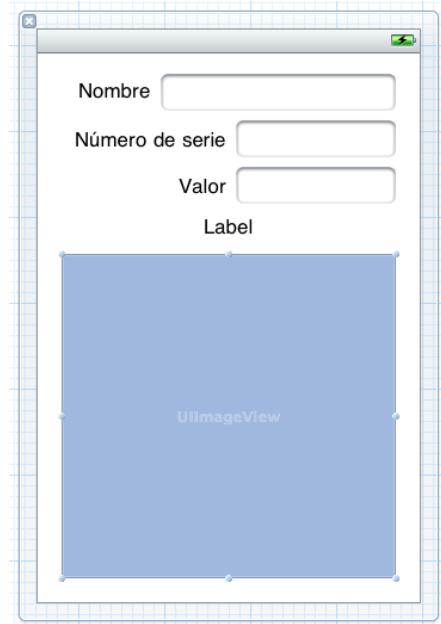
```
ibtool --import-strings-file ~/Desktop/BNRDetailViewController.strings --write  
/iphone/Homepwner/Homepwner/es.lproj/BNRDetailViewController.xib  
/iphone/Homepwner/Homepwner/en.lproj/BNRDetailViewController.xib
```

This command says, “Create `BNRDetailViewController.xib` in `es.lproj` from the `BNRDetailViewController.xib` in `en.lproj`, and then replace all of the strings with the values from `BNRDetailViewController.strings`.”

Press Enter. (You might see some sort of warning where `ibtool` complains about `GSCapabilities`; you can ignore it.)

Open `BNRDetailViewController.xib` (Spanish) in Xcode. This XIB file is now localized to Spanish. To finish things off, resize the label and text field for the serial number, as shown in Figure 25.11.

Figure 25.11 Spanish `BNRDetailViewController.xib`



26

NSUserDefaults

When you start an app for the first time, it uses its factory settings. As you use it, a good app learns your preferences. Where are your preferences stored? Inside each app bundle there is a *plist* that holds the user's preferences. As a developer, you will access this plist using the **NSUserDefaults** class. The preferences plist for your app can also be edited by the Settings app. To allow this, you create a *settings bundle* inside your app.

In this chapter, you will teach Homepwner how to read and write user preferences. Then you will create a settings bundle.

NSUserDefaults

The set of defaults for a user is a collection of key-value pairs. The key is the name of the default, and the value is some data that represents what the user prefers for that key. You ask the shared user defaults object for the value of that key – not unlike getting an object from a dictionary:

```
NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
NSString *greeting = [defaults objectForKey:@"FavoriteGreeting"];
```

If the user expresses a preference, you can set the value for that key:

```
NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
[defaults setObject:@"Hello" forKey:@"FavoriteGreeting"];
```

This value will automatically be stored to the app's preferences plist. Thus, the value must be a plist type: **NSArray**, **NSDictionary**, **NSString**, **NSData**, **NSDate**, or **NSNumber**. If you want to store a non-plist type to the user defaults, you will need to convert it to a plist. Often this is accomplished by archiving the object (or objects) into an **NSData**, which is a plist.

What if you ask for the value of a preference that has not been set by the user? **NSUserDefaults** will return the factory settings, the “default default,” if you will. These are not stored on the file system, so you need to tell the shared instance of **NSUserDefaults** what the factory settings are every time your app launches. And you need to do it early in the launch process – before any of your classes try to read the defaults. Typically you will override **+initialize** on your app delegate:

```
+ (void)initialize
{
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    NSDictionary *factorySettings = @{@"FavoriteGreeting": @"Hey!",
                                       @"HoursBetweenMothershipConnection : @2"};
    [defaults registerDefaults:factorySettings];
}
```

The class method `initialize` is called automatically by the Objective-C runtime before the first instance of that class is created.

In this section, you are going to add preferences for an item's initial value and name.

Register the factory settings

At launch time, the first thing that will happen is the registering of the factory settings. It is considered good style to declare your preference keys as global constants. Open `BNRAppDelegate.h` and declare two constant global variables:

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

extern NSString * const BNRNextItemValuePrefsKey;
extern NSString * const BNRNextItemNamePrefsKey;

@interface BNRAppDelegate : UIResponder <UIApplicationDelegate>

@property (strong, nonatomic) UIWindow *window;

@end
```

In `BNRAppDelegate.m`, define those global variables and use them to register the factory defaults in `+initialize`:

```
#import "BNRAppDelegate.h"
#import "BNRItemsViewController.h"
#import "BNRItemStore.h"

NSString * const BNRNextItemValuePrefsKey = @""NextItemValue";
NSString * const BNRNextItemNamePrefsKey = @""NextItemName";

@implementation BNRAppDelegate

+ (void)initialize
{
    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    NSDictionary *factorySettings = @{@"BNRNextItemValuePrefsKey": @75,
                                       BNRNextItemNamePrefsKey: @"Coffee Cup"};
    [defaults registerDefaults:factorySettings];
}
```

Read a preference

When you create a new item in `BNRItemStore.m`, use the default values. Be sure to import `BNRAppDelegate.h` at the top of `BNRItemStore.m` so that the compiler knows about `BNRNextItemValuePrefsKey`.

```
- (BNRItem *)createItem
{
    double order;
    if (_allItems.count == 0) {
        order = 1.0;
    } else {
        order = [[self.privateItems lastObject] orderingValue] + 1.0;
    }

    BNRItem *item = [NSEntityDescription insertNewObjectForEntityForName:@"BNRItem"
                                                inManagedObjectContext:self.context];
    item.orderingValue = order;

    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    item.valueInDollars = [defaults integerForKey:BNRNextItemValuePrefsKey];
    item.itemName = [defaults objectForKey:BNRNextItemNamePrefsKey];

    // Just for fun, list out all the defaults
    NSLog(@"%@", [defaults dictionaryRepresentation]);

    [self.privateItems addObject:item];
}

return item;
}
```

Notice the method `integerForKey:`. It is there as a convenience. It is equivalent to:

```
item.valueInDollars = [[defaults objectForKey:BNRNextItemValuePrefsKey] intValue];
```

There are also convenience methods for setting and getting `float`, `double`, `BOOL`, and `NSURL` values.

Change a preference

You could create a view controller for editing these preferences. Or, you could create a settings bundle for setting these preferences. Or, you can just try to guess the user's preferences from their actions. For example, if the user sets the value of an item to \$100, that may be a good indication that the next item might also be \$100. For this exercise, you will do that.

Open `BNRDetailViewController.m` and edit the `viewWillDisappear:` method.

```
- (void)viewWillDisappear:(BOOL)animated
{
    [super viewWillDisappear:animated];
    [self.view endEditing:YES];

    BNRItem *item = self.item;
    item.itemName = self.nameField.text;
    item.serialNumber = self.serialNumberField.text;

    int newValue = [self.valueField.text intValue];

    // Is it changed?
    if (newValue != item.valueInDollars) {

        // Put it in the item
        item.valueInDollars = newValue;

        // Store it as the default value for the next item
        NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
        [defaults setInteger:newValue
                      forKey:BNRNextItemValuePrefsKey];
    }
    item.valueInDollars = [self.valueField.text intValue];
}
```

Import `BNRAppDelegate.h` so that the compiler knows about the `BNRNextItemValuePrefsKey` constant. Build and run your app. Create an item named “Coffee Cup” with a value of \$75. The next item you create should default to the same value.

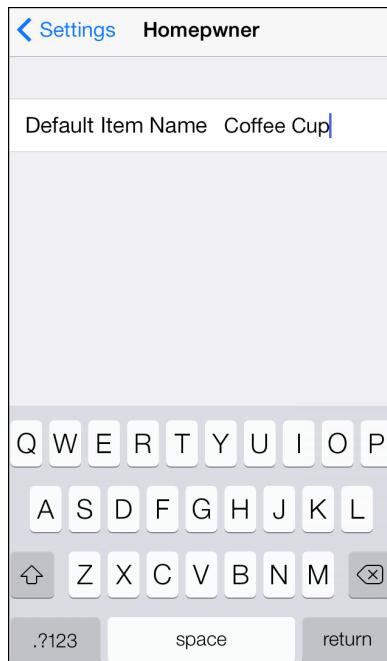
Also, on the console, you will see a list of all the defaults that are available to you. Most of them are from the `NSGlobalDomain`, which holds the global defaults for your entire device, like what language you prefer. `NextItemName`, because you have never set the value, is being read from the factory defaults, which is known as the `NSRegistrationDomain`. Now that you have set `NextItemValue`, it is being read from the `com.bignerdranch.Homeowner` domain, which is held in the preferences plist in your app’s sandbox. You can think of each domain as a dictionary of key-value pairs. **NSUserDefaults** gives the dictionaries different precedence. For example, the `com.bignerdranch.Homeowner` domain gets precedence over the `NSRegistrationDomain` – if the default has a value in the app’s preference plist, the registration domain is ignored.

Sometimes you will give the user a button that says “Restore factory default”, which will remove some defaults from the app’s preferences plist. To remove key-value pairs from your app’s preferences plist, **NSUserDefaults** has a `removeObjectForKey:` method.

Settings Bundle

Now you are going to create a settings bundle so that the `NextItemName` preference can be changed from “Coffee Cup” to whatever string the user desires.

Figure 26.1 Homepwner settings bundle

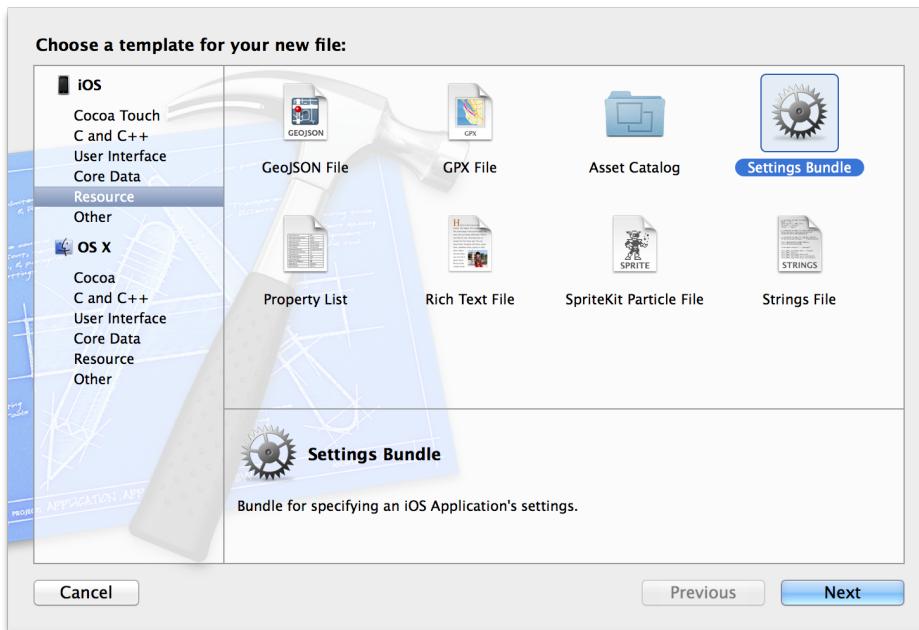


These days many designers consider settings bundles to be distasteful and most apps do not include a settings bundle. That said, many apps *do* have settings bundles, so it is a good idea to know how to make them.

The phrase “settings bundle” makes it sound scarier than it is. The bundle is just a directory that holds a plist that describes what controls should appear in this view and what default each control is manipulating. You will pull the user visible strings (like the label “Default Item Name” in Figure 26.1) into a strings file that is localized for the user. Those strings files will also be in the settings bundle.

To create a settings bundle inside your app, open Xcode’s File menu and choose New → File.... Under the iOS Resources pane, choose Settings Bundle (Figure 26.2).

Figure 26.2 Creating a new settings bundle



Accept the default name. Notice that a directory called `Settings.bundle` has been created in your project directory. It has a `Root.plist` file and an `en.lproj` subdirectory.

Editing the Root.plist

The `Root.plist` describes what controls will appear in your app's settings pane. It contains an array of dictionaries; each dictionary represents one view (typically a control) that will appear on the pane. Every dictionary must have `Type` key. Here are the acceptable values for `Type`:

| | |
|--------------------------------------|---|
| <code>PSTextFieldSpecifier</code> | a labeled text field |
| <code>PSToggleSwitchSpecifier</code> | a labeled toggle switch |
| <code>PSSliderSpecifier</code> | a slider (not labeled) |
| <code>PSRadioGroupSpecifier</code> | a list of radio buttons; only one can be selected |
| <code>PSMultiValueSpecifier</code> | a table view of possibilities; only one can be selected |
| <code>PSTitleValueSpecifier</code> | a title for formatting |
| <code>PSGroupSpecifier</code> | a group for formatting |
| <code>PSChildPaneSpecifier</code> | lets you move some preferences onto a child pane |

Several of these appear in the default `Root.plist`. Take a look, then build and run `Homepwner`. As soon as `Homepwner` is running, go to the `Settings` app and look at `Homepwner`'s pane.

Back in Xcode, open `Root.plist` and reduce it to an array containing just one text field:

- Set `Identifier` to `NextItemName`. This is the key for the default that is being set.
- Set `DefaultValue` to `Coffee Cup`. This is what comes up if there is no value for `Key` in your app's preferences plist.
- Set `Title` to `NextItemName`. This is used to look up the title in the strings file.

Your settings should look like Figure 26.3.

Figure 26.3 Root.plist

| Key | Type | Value |
|--------------------------|------------|----------------|
| ▼ iPhone Settings Schema | Dictionary | (2 items) |
| ▼ Preference Items | Array | (1 item) |
| ▼ Item 0 (Text Field - | Dictionary | (7 items) |
| Default Value | String | Coffee Cup |
| Text Field Is Secure | Boolean | NO |
| Identifier | String | NextItemName |
| Keyboard Type | String | Alphabet |
| Title | String | NextItemName |
| Type | String | Text Field |
| Autocorrection Style | String | Autocorrection |
| Strings Filename | String | Root |

Notice that you have been laying out a user interface using a plist file. When you create a settings bundle, you are not writing any executable code, and you are not creating any view controllers or other objects that you control. The `Settings` application will read your application's `Root.plist` and will construct its own view controllers based on the contents of the plist file.

If you are creating a settings bundle for one of your apps, you will need to refer to Apple's *Settings Application Schema Reference* for a complete list of all the keys/values that will work in this plist.

Localized Root.strings

Inside your settings bundle is an `en.lproj` which will hold your English strings. You can delete all the key-value pairs and give the title for your text field:

```
"NextItemName" = "Default Item Name";
```

That is it. Build and run your application. It should use the item name from the `Settings` app whenever you create a new item.

One final point: when your defaults are changed (by your own app or the `Settings` app) an `NSUserDefaultsDidChangeNotification` will get posted to your application. If you want to respond to changes in the `Settings` app immediately, register as an observer of this notification.

This page intentionally left blank

27

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 27.1).

Figure 27.1 **UIView** animation documentation

The screenshot shows the Xcode Documentation browser with the title "Documentation — UIView Class Reference". The main content area is titled "UIView" with a gear icon. It lists inheritance from "UIResponder : NSObject", conformance to "UIAppearance, NSCoding, UIAppearanceContainer, NSObject, UIDynamicItem", and its framework "UIKit in iOS 2.0 and later". A section titled "Animations" is expanded, containing text about animating view properties and two recommended methods: block-based animation methods (iOS 4+) and begin/commit animation methods. Below this, a note discusses block-based animations being available only in iOS 4+. It also lists animatable properties: frame, bounds, center, transform, alpha, backgroundColor, and contentStretch. At the bottom, a link to "View Programming Guide for iOS" and a "Provide Feedback" button are visible.

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 27.2).

Figure 27.2 Basic animation



Open HypnoNerd.xcodeproj.

The first animation you will add will animate the `alpha` value of the labels when they are added to the view.

Open `BNRHypnosisViewController.m` and add an animation to the labels in `drawHypnoticMessage:`.

```
[self.view addSubview:messageLabel];

// Set the label's initial alpha
messageLabel.alpha = 0.0;

// Animate the alpha to 1.0
[UIView animateWithDuration:0.5 animations:^{
    messageLabel.alpha = 1.0;
}];

UIInterpolatingMotionEffect *motionEffect =
[[UIInterpolatingMotionEffect alloc] initWithKeyPath:@"center.x"
type:UIInterpolatingMotionEffectTypeTiltAlongHorizontalAxis];
```

Build and run the application. After you enter some text and tap the return key, the labels should 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 block-based animation method on **UIView** is **animateWithDuration:animations:**. This method takes in the duration that the animation should run for and a block 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 animation block, you can also specify how long to delay before the animations should begin, some options (which we will look at shortly), and a completion block that will get called when the animation sequence completes.

In **BNRHypnosisViewController.m**, change the animation in **drawHypnoticMessage:** to use this new animation method:

```
[self.view addSubview:messageLabel];

// Set the label's initial alpha
messageLabel.alpha = 0.0;

// Animate the alpha to 1.0
UIView animateWithDuration:0.5 animations:^{
    messageLabel.alpha = 1.0;
}};

[UIView animateWithDuration:0.5
    delay:0.0
    options:UIViewAnimationOptionCurveEaseIn
    animations:^{
        messageLabel.alpha = 1.0;
}
completion:NULL];

UIInterpolatingMotionEffect *motionEffect =
    [[UIInterpolatingMotionEffect alloc] initWithKeyPath:@"center.x"
    type:UIInterpolatingMotionEffectTypeTiltAlongHorizontalAxis];
```

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

- UIViewAnimationOptionCurveEaseInOut
- UIViewAnimationOptionCurveEaseOut
- UIViewAnimationOptionCurveEaseIn
- UIViewAnimationOptionCurveLinear

UIViewAnimationOptionAllowUserInteraction

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.

UIViewAnimationOptionRepeat

This will repeat the animation indefinitely. This is often paired with the `UIViewAnimationOptionAutoreverse` option.

UIViewAnimationOptionAutoreverse

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 27.3). You can think of keyframe animations as multiple basic animations going back to back.

Figure 27.3 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 block using the **addKeyframeWithRelativeStartTime:relativeDuration:animations:** class method.

In **BNRHypnosisViewController.m**, update **drawHypnoticMessage:** to animate the center of the labels first to the middle of the screen and then to another random position on the screen.

```
[UIView animateWithDuration:0.5
                      delay:0.0
                     options:UIViewAnimationOptionCurveEaseIn
                   animations:^{
                       messageLabel.alpha = 1.0;
                   }
                   completion:NULL];

[UIView animateKeyframesWithDuration:1.0 delay:0.0 options:0 animations:^{
    [UIView addKeyframeWithRelativeStartTime:0 relativeDuration:0.8 animations:^{
        messageLabel.center = self.view.center;
    }];
    [UIView addKeyframeWithRelativeStartTime:0.8 relativeDuration:0.2 animations:^{
        int x = arc4random() % width;
        int y = arc4random() % height;
        messageLabel.center = CGPointMake(x, y);
    }];
} completion:NULL];

UIInterpolatingMotionEffect *motionEffect =
[[UIInterpolatingMotionEffect alloc] initWithKeyPath:@"center.x"
                                             type:UIInterpolatingMotionEffectTypeTiltAlongHorizontalAxis];
```

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 **UIViewControllerAnimatedOptions** instead of **UIViewAnimationOptions**. The duration passed into this method is the duration of the entire animation.

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 80% of the total duration (a

relative duration of 0.8). The last keyframe starts 80% into the total duration (a relative start time of 0.8) and lasts 20% of the total duration (a relative duration of 0.2).

Build and run the application and enter some text. The labels will now animate to the center of the screen before exploding out to a random final position.

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 block for the completion argument.

Update `BNRHypnosisViewController.m` so that it logs a message to the console when the animations complete.

```
[UIView animateWithDuration:1.0 delay:0.0 options:0 animations:^{
    [UIView addKeyframeWithRelativeStartTime:0 relativeDuration:0.8 animations:^{
        messageLabel.center = self.view.center;
    }];
    [UIView addKeyframeWithRelativeStartTime:0.8 relativeDuration:0.2 animations:^{
        int x = arc4random() % width;
        int y = arc4random() % height;
        messageLabel.center = CGPointMake(x, y);
    }];
} completion:NULL];

[UIView animateWithDuration:1.0 delay:0.0 options:0 animations:^{
    [UIView addKeyframeWithRelativeStartTime:0 relativeDuration:0.8 animations:^{
        messageLabel.center = self.view.center;
    }];
    [UIView addKeyframeWithRelativeStartTime:0.8 relativeDuration:0.2 animations:^{
        int x = arc4random() % width;
        int y = arc4random() % height;
        messageLabel.center = CGPointMake(x, y);
    }];
} completion:^(BOOL finished) {
    NSLog(@"Animation finished");
}];

UIInterpolatingMotionEffect *motionEffect =
[[UIInterpolatingMotionEffect alloc] initWithKeyPath:@"center.x"
    type:UIInterpolatingMotionEffectTypeTiltAlongHorizontalAxis];
```

Build and run the application, and log messages will appear in the console as soon as the animations complete.

You might be wondering, “What if the animation repeats? Will the completion block be executed after each repeat?” No, the completion block will only be executed once, at the very end.

Spring Animations

iOS has a powerful physics engine built into the SDK, and one of the easiest ways to use it is with the new *spring animations*. This type of animation has a timing function like that of an actual spring. You will use this to animate the text field dropping in from the top of the screen, as if it was attached to a spring.

In `BNRHypnosisViewController.m`, add a property for the text field to the class extension and update `loadView` to store the reference to the text field. Then start with the text field offscreen:

```
@interface BNRHypnosisViewController () <UITextFieldDelegate>
@property (nonatomic, weak) UITextField *textField;
@end

@implementation BNRHypnosisViewController
// Other methods

- (void)loadView
{
    CGRect frame = [UIScreen mainScreen].bounds;
    BNRHypnosisView *backgroundView = [[BNRHypnosisView alloc] initWithFrame:frame];

    CGRect textFieldRect = CGRectMake(40, 70, 240, 30);
    UITextField *textField = [[UITextField alloc] initWithFrame:textFieldRect];

    // Setting the border style on the text field will allow us to see it more easily
    textField.borderStyle = UITextBorderStyleRoundedRect;
    [backgroundView addSubview:textField];

    self.textField = textField;
    self.view = backgroundView;
}

@end
```

It will be best to begin the animation as soon as the view is on the screen, so the animation code will go into `viewDidAppear:`. Currently there is no property pointing to the text field, but you will need one in order to update its frame in `viewDidAppear:`.

Now, in `BNRHypnosisViewController.m`, override `viewDidAppear:` to drop in the text field using a spring animation.

```
- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];

    [UIView animateWithDuration:2.0
                      delay:0.0
                     usingSpringWithDamping:0.25
                     initialSpringVelocity:0.0
                           options:0
                         animations:^{
                             CGRect frame = CGRectMake(40, 70, 240, 30);
                             self.textField.frame = frame;
                         }
                     completion:NULL];
}
```

The individual components of this method are relatively straightforward:

| | |
|------------------------|--|
| <i>duration</i> | The total time the animation should last. |
| <i>delay</i> | How long until the animation should begin. |
| <i>spring damping</i> | A number between 0 and 1. The closer to 0, the more the animation oscillates. |
| <i>spring velocity</i> | The relative velocity of the view when the animation is to begin. You will almost always pass in 0 for this. |
| <i>options</i> | <code>UIViewControllerAnimatedOptions</code> , just like with the other animations. |
| <i>animations</i> | A block of changes to animate on one or more views. |
| <i>completion</i> | A block to run when the animation is finished. |

Build and run the application, and the text field should animate from the top of the screen, bouncing like a spring at the end.

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.

28

UIStoryboard

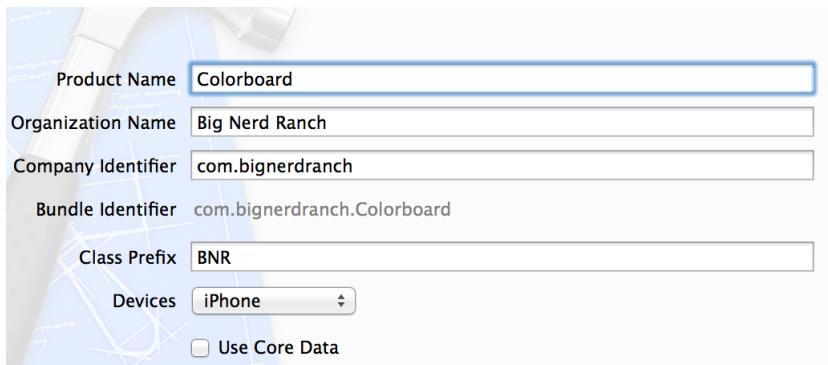
In your projects so far, you have laid out the interfaces of your view controllers in separate XIB files and then instantiated the view controllers programmatically. In this chapter, you will use a *storyboard* instead. Storyboards are a feature of iOS that allows you to instantiate and lay out all of 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. To see this simplification – and its drawbacks – let's create an application that uses a storyboard.

Creating a Storyboard

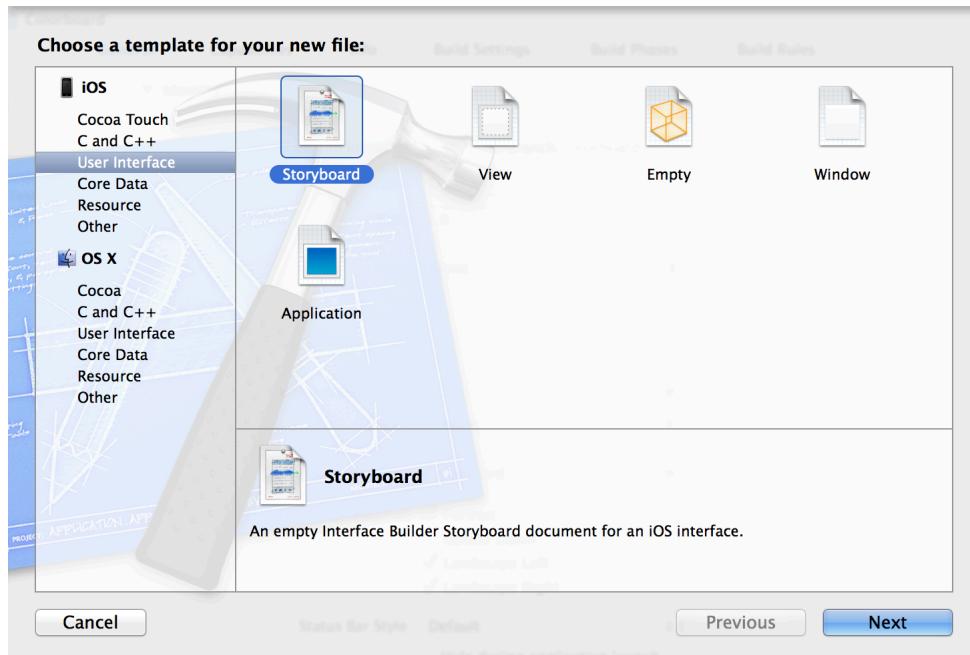
Create a new iOS Empty Application and name it Colorboard (Figure 28.1).

Figure 28.1 Creating Colorboard



Then, select **New File...** from the **New** menu. Select **User Interface** from the **iOS** section. Then, select the **Storyboard** template and click **Next** (Figure 28.2).

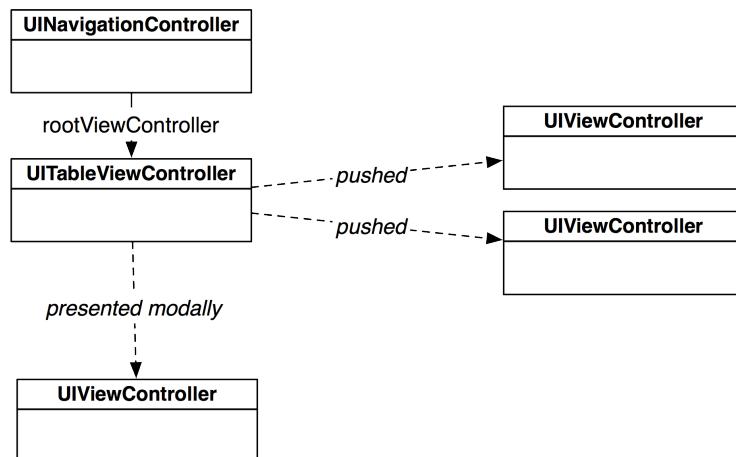
Figure 28.2 Creating a storyboard



On the next pane, select iPhone from the Device Family pop-up menu and click Next. Then, name this file `Colorboard`.

This will create a new file named `Colorboard.storyboard` and open it in the editor area. A storyboard is a lot like a XIB, except it allows you to lay out the relationships between view controllers in addition to laying out their interfaces. The Colorboard application will have a total of five view controllers, including a `UINavigationController` and a `UITableViewController`. Figure 28.3 shows an object diagram for Colorboard.

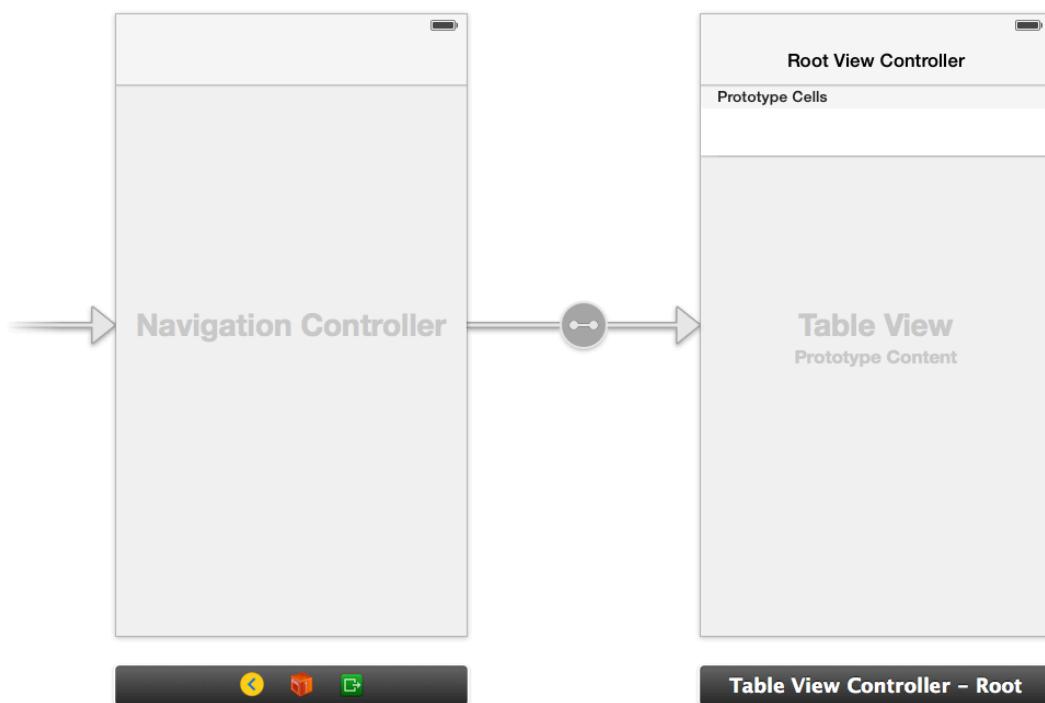
Figure 28.3 Object diagram for Colorboard



Using a storyboard, you can set up the relationships shown in Figure 28.3 without writing any code.

To get started, open the utility area and the Object Library. Drag a Navigation Controller onto the canvas. The canvas will now look like Figure 28.4.

Figure 28.4 Navigation controller in storyboard

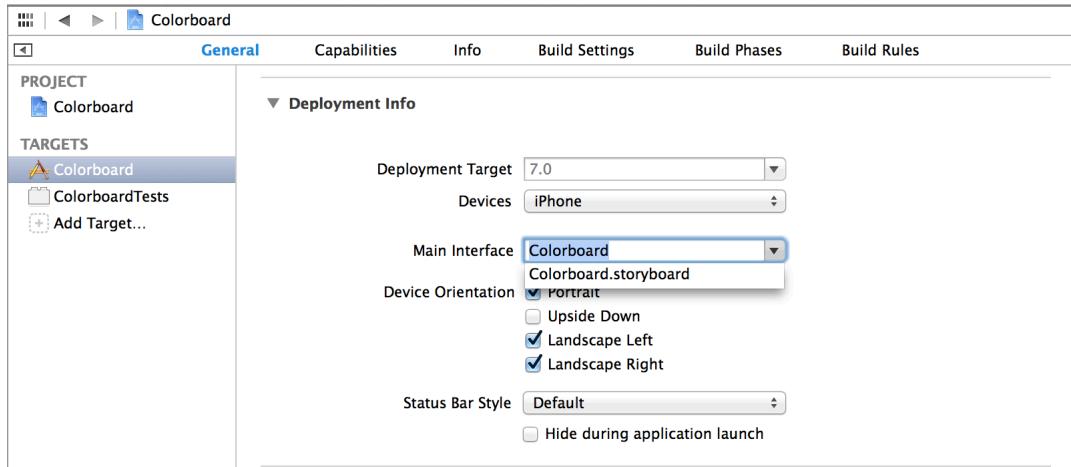


In addition to the **UINavigationController** object you asked for, the storyboard took the liberty of creating three other objects: the view of the navigation controller, a **UITableViewController**, and the view of the **UITableViewController**. In addition, the **UITableViewController** has been made the root view controller of the navigation controller.

The two view controller instances are represented by the black bars on the canvas, and their views are shown above them. You configure the view the same as you would in a normal XIB file. To configure the view controller itself, you select the black bar.

Before you go any further, you need to tell your application about this storyboard file. Select the Colorboard project from the project navigator. Then, select the Colorboard target and the General tab. Locate the Main Interface field and enter Colorboard (Figure 28.5) or select Colorboard.storyboard from the dropdown.

Figure 28.5 Setting the main storyboard



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.

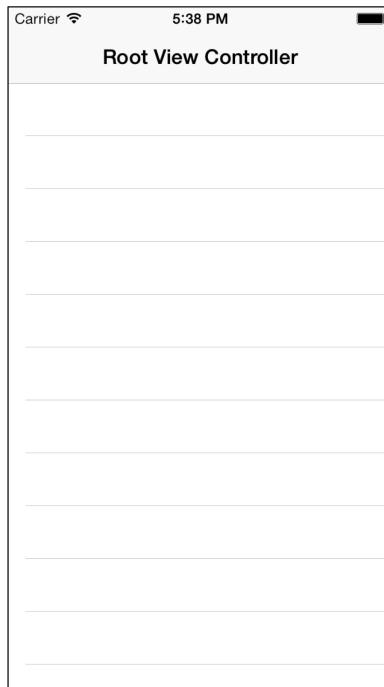
Since a storyboard file supplies the window for an application, the application delegate does not need to create a window.

In `BNRAppDelegate.m`, remove the code from `application:didFinishLaunchingWithOptions:` that creates the window.

```
- (BOOL)application:(UIApplication *)application  
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions  
{  
    self.window = [[UIWindow alloc] initWithFrame:[UIScreen mainScreen] bounds]];  
    // Override point for customization after application launch  
    self.window.backgroundColor = [UIColor whiteColor];  
    [self.window makeKeyAndVisible];  
    return YES;  
}
```

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 28.6). All of this came from the storyboard file – you did not have to write any code.

Figure 28.6 Initial Colorboard screen



UITableViewController in Storyboards

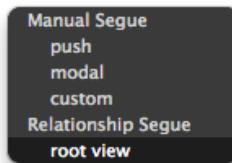
When using a **UITableViewController**, you typically implement the appropriate data source methods to return the content of each cell. This makes sense when you have dynamic content – like a list of items that may change – but it is a lot of work when you have a table whose content never changes. Storyboards allow you to add static content to a table view without having to implement the data source methods.

To see how easy this is, you are going to add a **UITableViewController** to the storyboard and give it static content. Apple frequently changes the file templates, so it is possible that your **UINavigationController**'s `rootViewController` is not already a **UITableViewController** instance. Either way (practice is good!), let's go through the steps of adding one.

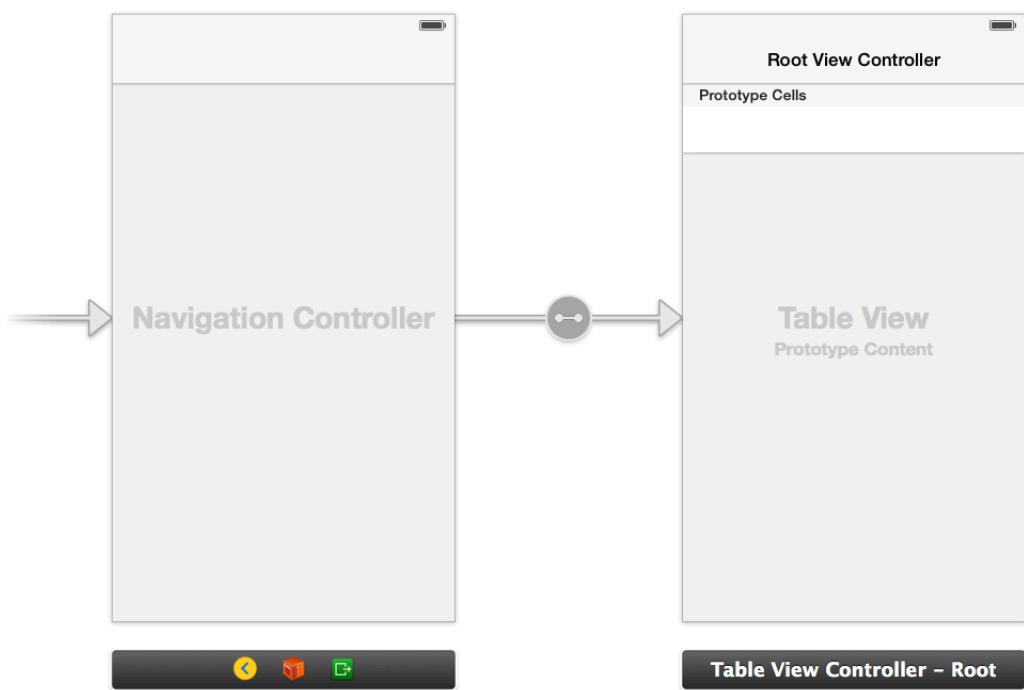
If there is already a second view controller in your Storyboard next to the navigation controller, select its black bar (the representation of the view controller itself), and delete it.

Next, drag a **UITableViewController** from the library onto the canvas. To set this table view controller as the root view controller of the navigation controller, Control-drag from the navigation controller's view to the table view controller's view. Let go, and from the black panel that appears, select `root view` (Figure 28.7). Remember that despite dragging between views, these properties are being set on the view controllers themselves.

Figure 28.7 Setting a relationship



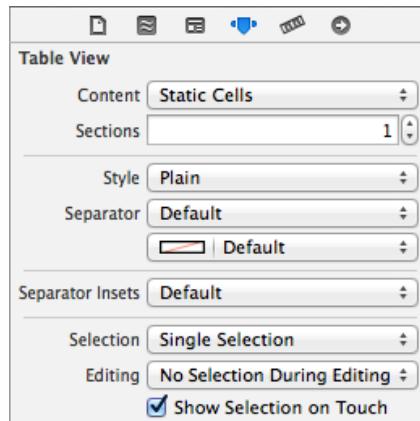
This establishes the **UITableViewController** as the root view controller of the **UINavigationController**. There will now be an arrow from the navigation controller to the table view controller. In the middle of this arrow is an icon that represents the type of relationship between the two view controllers (Figure 28.8).

Figure 28.8 **UINavigationController** and **UITableViewController**

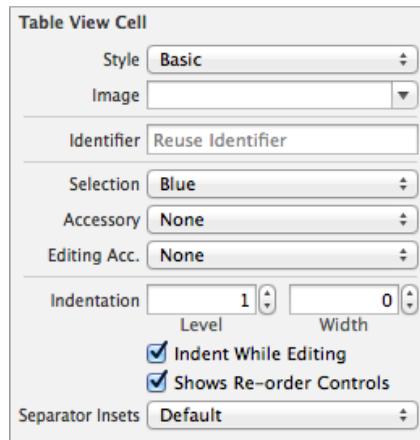
(Notice the zoom in and out controls in the bottom right corner of the canvas? You can zoom in and out to see more of the canvas. This is especially useful when you have a lot of view controllers. However, you cannot select the view objects when zoomed out.)

Next, select the Table View of the **UITableViewController**. In the attributes inspector, change the Content pop-up menu to Static Cells (Figure 28.9).

Figure 28.9 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 28.10).

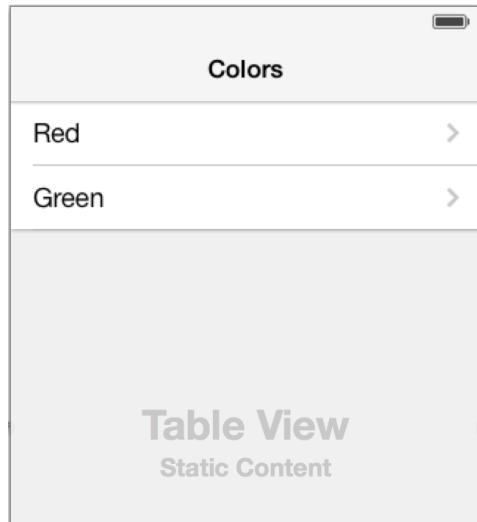
Figure 28.10 Basic **UITableViewCell**

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 28.11 shows the updated table view.

Figure 28.11 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.

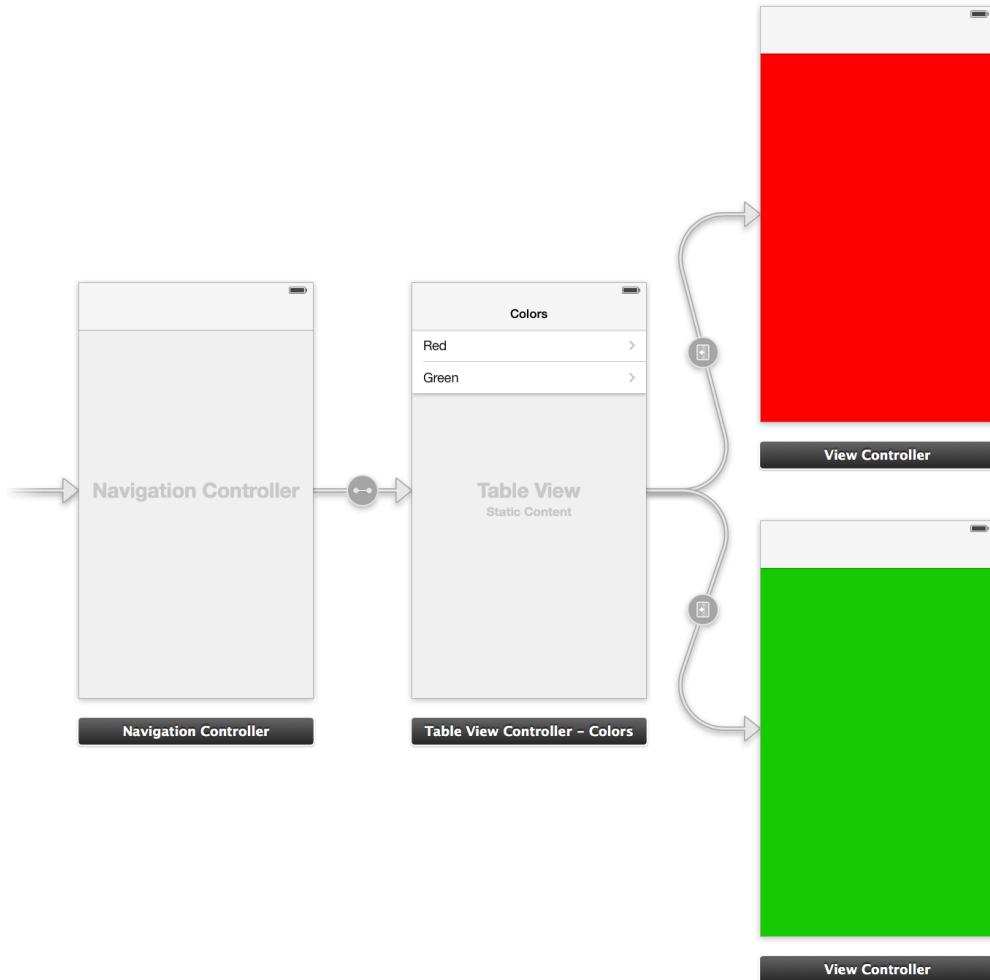
Let's start with two push segues. A push segue pushes a view controller onto the stack of a navigation controller. You will need to set up two more view controllers in your storyboard, one whose view's background is red, and the other, green. The segues will be between the table view controller and these 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 Push.

Then, select the Green cell and Control-drag to the other view controller. Your canvas should look like Figure 28.12.

Figure 28.12 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 push segues.

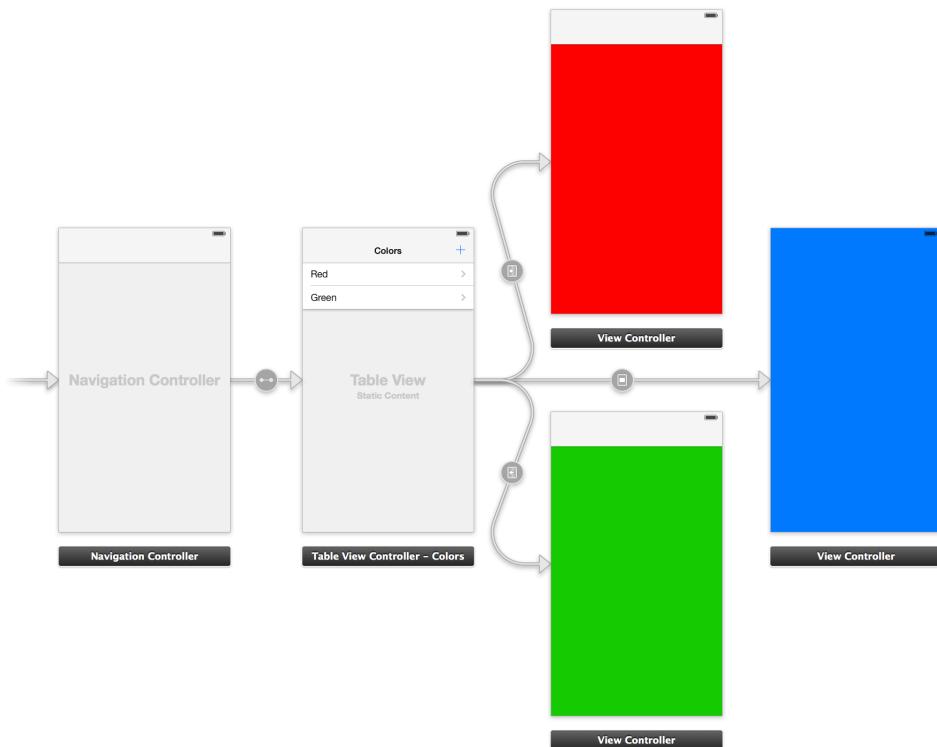
Build and run the application. Tap on each row, and you will be taken to the appropriate view controller. You can even move back in the navigation stack to the table view controller like you would expect. The best part about this? You have not written any code yet.

Note that push segues only work if the origin of the segue is inside a navigation controller. Fortunately, the origin of these segues is the table view controller, which meets this requirement.

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 Modal from the black panel. The storyboard canvas now looks like Figure 28.13. (Notice that the icon for the modal segue is different from the icon for the push segues.)

Figure 28.13 A modal segue

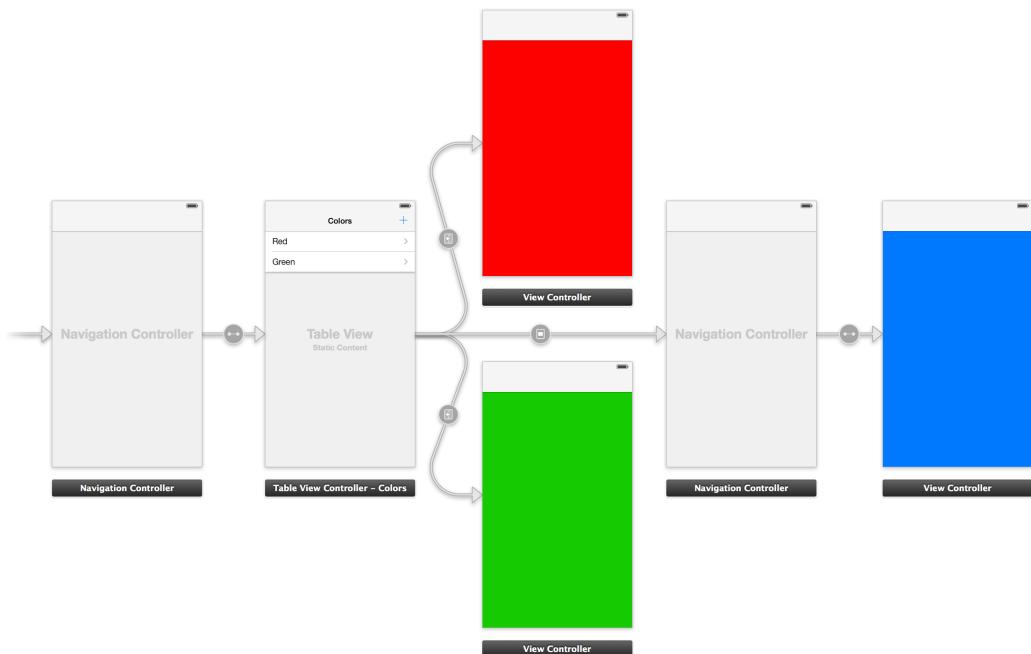


Build and run the application. Tap the bar button item, and a view controller with a blue view will slide onto the screen. All is well – except you cannot dismiss this view controller.

You will dismiss the view controller from a **UIBarButtonItem** on the navigation bar that says Done. Currently, the modal view controller is being presented by itself, so it has no navigation bar for the bar button item. To fix this, drag a **UINavigationController** onto the canvas and delete the **UITableView** (or whatever the second view controller was that Apple provided with the navigation controller).

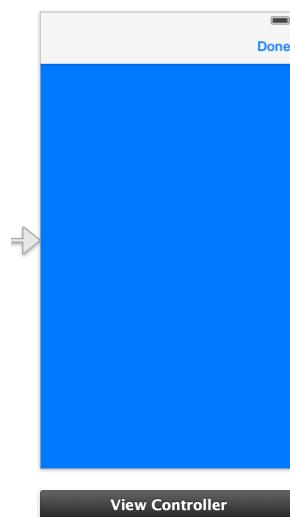
Delete the existing modal segue, and instead have the + item trigger a modal segue to the navigation controller. The existing blue view controller should be the root of the navigation controller. Your storyboard should now look like Figure 28.14.

Figure 28.14 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 28.15.

Figure 28.15 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 **BNRColorViewController**.

In **BNRColorViewController.h**, change the superclass to be **UIViewController**.

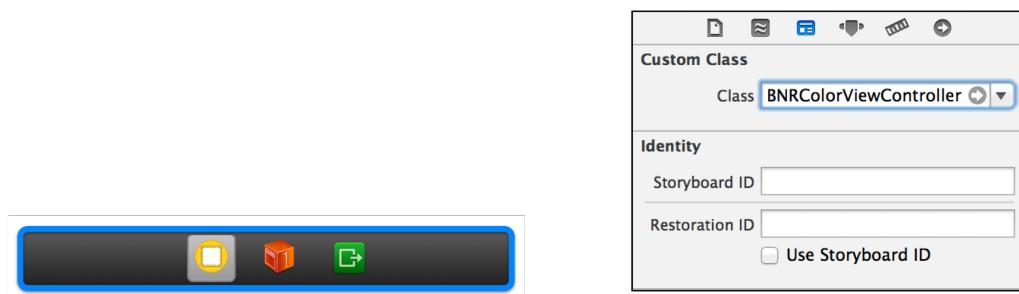
```
@interface BNRColorViewController : NSObject
@interface BNRColorViewController : UIViewController
@end
```

Then in **BNRColorViewController.m**, implement a method to dismiss itself.

```
- (IBAction)dismiss:(id)sender
{
    [self.presentingViewController dismissViewControllerAnimated:YES
                                                    completion:nil];
}
```

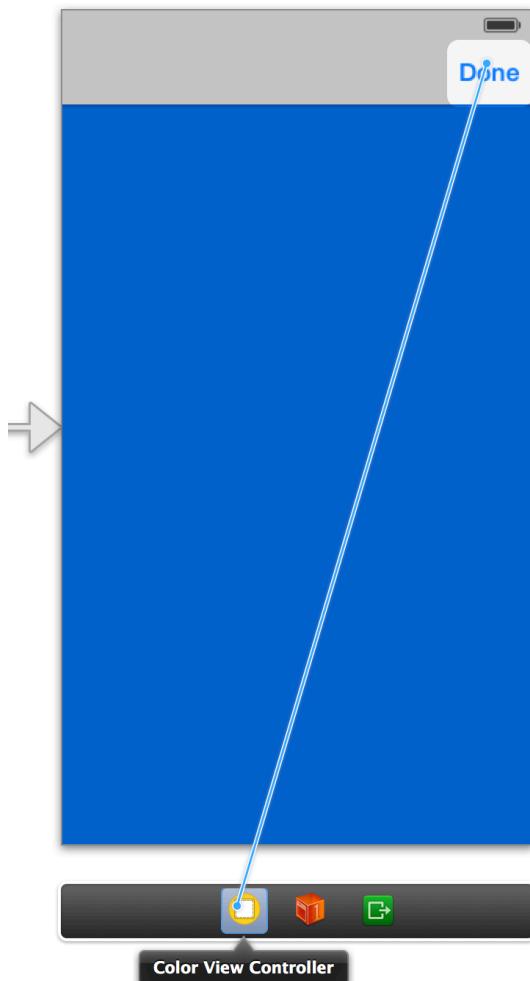
Open **Colorboard.storyboard** again. Select the black bar underneath the modally presented (blue) view controller. (This is called the *scene dock*.) In the identity inspector, change the Class to **BNRColorViewController** (Figure 28.16).

Figure 28.16 Changing view controller to **BNRColorViewController**



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 28.17).

Figure 28.17 Setting outlets and actions in a storyboard

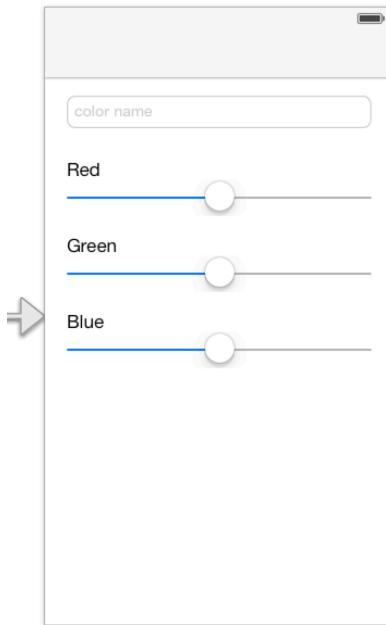


This button is now hooked up to send the message `dismiss:` to its `BNRColorViewController` whenever tapped. Build and run the application, present the `BNRColorViewController`, and then tap on the Done button. Voilà!

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 `Colorboard.storyboard`, add one `UITextField`, three `UILabel` objects, and three `UISlider` objects to the view of `BNRColorViewController` so it looks like Figure 28.18.

Figure 28.18 Configuring the view for **BNRColorViewController**

Let's have the background color of **BNRColorViewController**'s view match the slider values. In **BNRColorViewController.m**, add outlets to the text field and three sliders in the class extension.

```
@interface BNRColorViewController : UIViewController

@property (nonatomic, weak) IBOutlet UITextField *textField;
@property (nonatomic, weak) IBOutlet UISlider *redSlider;
@property (nonatomic, weak) IBOutlet UISlider *greenSlider;
@property (nonatomic, weak) IBOutlet UISlider *blueSlider;

@end

@implementation
```

All three sliders will trigger the same method when their value changes. Implement this method in **BNRColorViewController.m**.

```
- (IBAction)changeColor:(id)sender
{
    float red = self.redSlider.value;
    float green = self.greenSlider.value;
    float blue = self.blueSlider.value;
    UIColor *newColor = [UIColor colorWithRed:red
                                    green:green
                                      blue:blue
                                     alpha:1.0];
    self.view.backgroundColor = newColor;
}
```

Now open `Colorboard.storyboard` and connect the outlets from Color View Controller (the first icon in the scene dock below the `BNRColorViewController` view) to the text field and three sliders. Then 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.

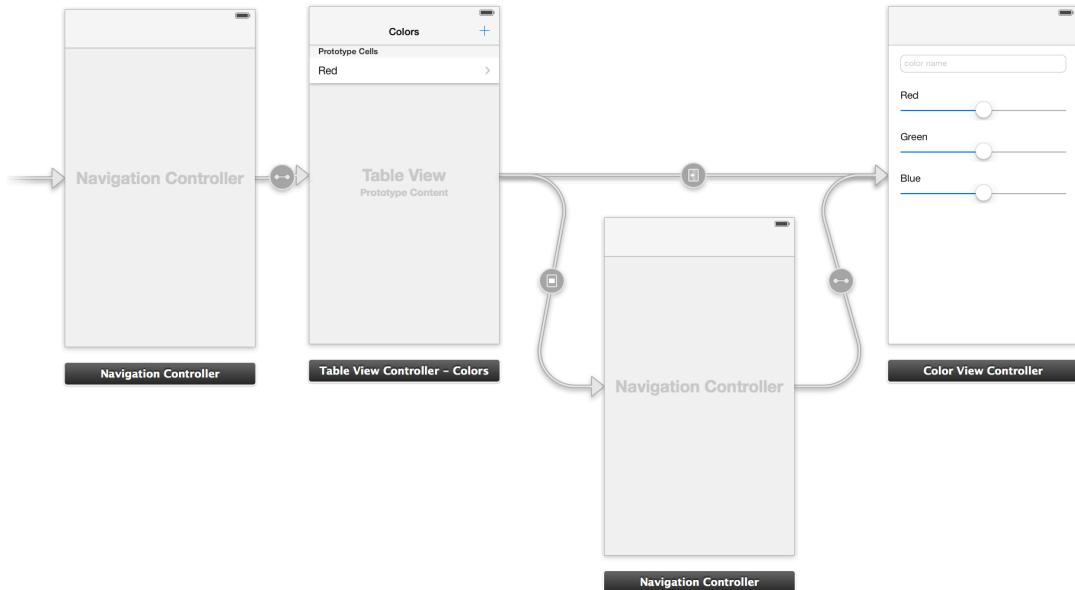
Passing Data Around

As we discussed in Chapter 10, 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 `BNRColorViewController` 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.

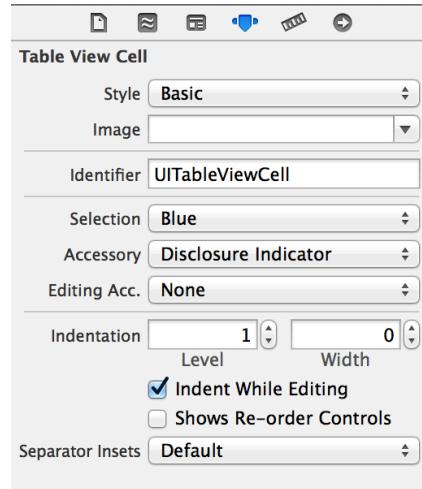
In `Colorboard.storyboard`, delete the Red and Green view controllers that are being pushed from the `UITableView`. Then select the table view and open the attributes inspector. Change its Content type to Dynamic Prototypes and delete the second `UITableViewCell`. The storyboard should look like Figure 28.19.

Figure 28.19 Dynamic prototypes storyboard



Then select the **UITableViewCell** and set its reuse identifier to **UITableViewCell** (Figure 28.20).

Figure 28.20 **UITableViewCell** reuse identifier



In order to supply this table view controller with data for its table view, you will need to create a new **UIViewController** subclass. Create a new **NSObject** subclass named **BNRPaletteViewController**.

In **BNRPaletteViewController.h**, change the superclass to be **UITableViewController**.

```
@interface BNRPaletteViewController : NSObject  
@interface BNRPaletteViewController : UITableViewController  
  
@end
```

In **BNRPaletteViewController.m**, import **BNRColorViewController.h** and add an **NSMutableArray** to the class extension.

```
#import "BNRPaletteViewController.h"  
#import "BNRColorViewController.h"  
  
@interface BNRPaletteViewController ()  
  
@property (nonatomic) NSMutableArray *colors;  
  
@end  
  
@implementation BNRPaletteViewController
```

Next, implement **viewWillAppear:** and the table view data source methods in **BNRPaletteViewController.m**.

```

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
    [self.tableView reloadData];
}

- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
{
    return [self.colors count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:@"UITableViewCell"
                                         forIndexPath:indexPath];

    return cell;
}

```

Next, create a new **NSObject** subclass named **BNRColorDescription** that will represent a user-defined color.

In **BNRColorDescription.h**, add properties for a **UIColor** and a name.

```

@interface BNRColorDescription : NSObject

@property (nonatomic) UIColor *color;
@property (nonatomic, copy) NSString *name;

@end

```

Then, in **BNRColorDescription.m**, override **init** to set default values for these properties.

```

@implementation BNRColorDescription

- (instancetype)init
{
    self = [super init];
    if (self) {
        _color = [UIColor colorWithRed:0
                                green:0
                                 blue:1
                                alpha:1];
        _name = @"Blue";
    }
    return self;
}

@end

```

To test whether the code works, let's add a new **BNRColorDescription** to the **colors** array of **BNRPaletteViewController**.

At the top of **BNRPaletteViewController.m**, import **BNRColorDescription.h**. Then override the **colors** accessor to lazily instantiate the array and add a new **BNRColorDescription** to the array.

```
#import "BNRPaletteViewController.h"
#import "BNRColorDescription.h"

@implementation BNRPaletteViewController

- (NSMutableArray *)colors
{
    if (!_colors) {
        _colors = [NSMutableArray array];

        BNRColorDescription *cd = [[BNRColorDescription alloc] init];
        [_colors addObject:cd];
    }
    return _colors;
}
```

Also, update the data source method in `BNRPaletteViewController.m` to display the name of the color.

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:@"UITableViewCell"
                                         forIndexPath:indexPath];

    BNRColorDescription *color = self.colors[indexPath.row];
    cell.textLabel.text = color.name;

    return cell;
}
```

Build and run the application. You should be able to drill down, but there are two problems. First, the “Blue” color is not being passed down to the `BNRColorDescription`. 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 `BNRColorViewController.h` 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. Do not forget to import `BNRColorDescription.h` at the top.

```
#import "BNRColorDescription.h"

@interface BNRColorViewController : UIViewController

@property (nonatomic) BOOL existingColor;
@property (nonatomic) BNRColorDescription *colorDescription;

@end
```

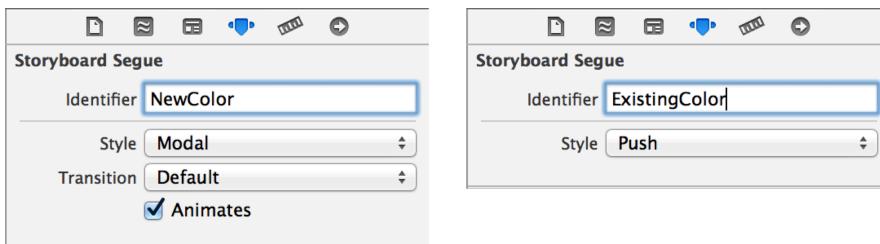
Whenever a segue is triggered on a view controller, it gets sent the message `prepareForSegue:sender:`. This method gives you both the `UIStoryboardSegue`, which gives you information about which segue is happening, and the `sender`, which is the object that triggered the segue (a `UIBarButtonItem` or 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 Colorboard.storyboard again. Select the modal segue and open its attribute inspector. For the identifier, type in NewColor. Next, select the push segue and give it the identifier ExistingColor. The attributes inspector for both segues is shown in Figure 28.21.

Figure 28.21 Segue identifiers



With your segues identified, you can now pass your color objects around. Open BNRPaletteViewController.m and implement `prepareForSegue:sender:`:

```
- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    if ([segue.identifier isEqualToString:@"NewColor"]) {

        // If we are adding a new color, create an instance
        // and add it to the colors array
        BNRCOLORDescription *color = [[BNRCOLORDescription alloc] init];
        [self.colors addObject:color];

        // Then use the segue to set the color on the view controller
        UINavigationController *nc =
            (UINavigationController *)segue.destinationViewController;
        BNRCOLORViewController *mvc =
            (BNRCOLORViewController *)[nc topViewController];
        mvc.colorDescription = color;
    }
    else if ([segue.identifier isEqualToString:@"ExistingColor"]) {

        // For the push segue, the sender is the UITableViewCell
        NSIndexPath *ip = [self.tableView indexPathForCell:sender];
        BNRCOLORDescription *color = self.colors[ip.row];

        // Set the color, and also tell the view controller that this
        // is an existing color
        BNRCOLORViewController *cvc =
            (BNRCOLORViewController *)segue.destinationViewController;
        cvc.colorDescription = color;
        cvc.existingColor = YES;
    }
}
```

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 `BNRCOLORDescription` and give it to the `BNRCOLORViewController`.

If you tap an existing color, the “ExistingColor” segue is triggered. Notice that when a **UITableViewCell** triggers a segue, it is sent as the `sender` argument, and you can use that to determine which index path was selected. The color that was tapped is then passed to the **BNRColorViewController**.

(Why is the `destinationViewController` for “NewColor” a **UINavigationController** when it is a **BNRColorViewController** 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 **BNRColorViewController**: 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 **BNRColorViewController** goes away (either by dismissing the modal view controller or popping from the navigation controller stack).

Open `BNRColorViewController.m` and override `viewWillAppear:` to get rid of the Done button if `existingColor` is YES.

```
- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];

    // Remove the 'Done' button if this is an existing color
    if (self.existingColor) {
        self.navigationItem.rightBarButtonItem = nil;
    }
}
```

Then, still in `BNRColorViewController.m`, override `viewDidLoad` to set the initial background color, slider values, and color name.

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    UIColor *color = self.colorDescription.color;

    // Get the RGB values out of the UIColor object
    float red, green, blue;
    [color getRed:&red
              green:&green
                blue:&blue
               alpha:nil];

    // Set the initial slider values
    self.redSlider.value = red;
    self.greenSlider.value = green;
    self.blueSlider.value = blue;

    // Set the background color and text field value
    self.view.backgroundColor = color;
    self.textField.text = self.colorDescription.name;
}
```

Finally, save the values when the view is disappearing.

```
- (void)viewWillDisappear:(BOOL)animated
{
    [super viewWillDisappear:animated];
    self.colorDescription.name = self.textField.text;
    self.colorDescription.color = self.view.backgroundColor;
}
```

Build and run the application and the colors should display and save correctly.

More on Storyboards

In this exercise, you created a storyboard, set up a few view controllers, laid out their interfaces, and created some segues between them. This is the basic idea behind storyboards, and while there are a few more flavors of segues and types of view controllers you can set up, you get the idea. A storyboard replaces lines of code.

For example, a push segues replace this code:

```
- (void)tableView:(UITableView *)tableView
didSelectRowAtIndexPath:(NSIndexPath *)ip
{
    UIViewController *vc = [[UIViewController alloc] init];
    [self.navigationController pushViewController:vc];
}
```

While this seems nice, storyboarding, in our opinion, is not very useful. Let's go through the pros and cons. First, the pros:

- Storyboards can be used to easily show off the flow of an application to a client or a colleague.
- Storyboards remove some simple code from your source files.
- Tables with static content are easy to create.
- Prototype cells replace the need to create separate XIBs for custom table view cells.
- Storyboards sure do look pretty.

The cons, unfortunately, outweigh the pros:

- Storyboards are difficult to work with in a team. Typically, a team of iOS programmers breaks up the work by having each member focus on a particular view controller. With a storyboard, everyone has to do their work in the same storyboard file. This can quickly lead to clutter and difficulties with version control.
- Storyboards disrupt the flow of programming. Let's say you are writing a view controller and adding the code for a button that presents a view controller modally. You can do that pretty easily in code – `alloc` and `init` the view controller, and send `presentViewController:animated:completion:` to `self`. With storyboards, you have to load up the storyboard file, drag some stuff onto the canvas, set the Class in the identity inspector, connect the segue, and then configure the segue.

- Storyboards sacrifice flexibility and control for ease of use. The work required to add advanced functionality to the basic functionality of a storyboard is often more than the work required to put together the advanced and basic functionality in code.
- Storyboards always create new view controller instances. Each time you perform a segue, a new instance of the destination view controller is created. Sometimes, though, you would like to keep a view controller around instead of destroying it each time it disappears off the screen. Storyboarding does not allow you to do this.

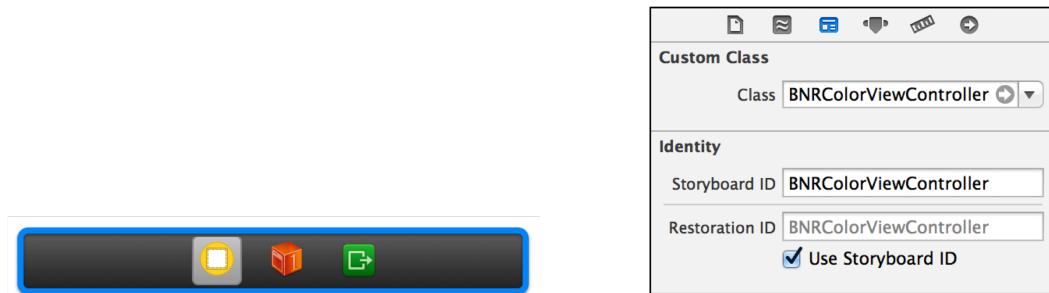
Overall, storyboards make easy code easier and difficult code more difficult. We do not use them in this book, and we do not typically use them when writing our own applications. However, Apple seems to be pushing them harder in each release of Xcode, so you might one day decide that a particular application would benefit from storyboarding.

For the More Curious: State Restoration

We discussed how to work with the state restoration system in Chapter 24, but the Homepwner application did not use storyboards. Let's take a look at how state restoration works when using storyboards.

Storyboards handle a lot of the boilerplate state restoration code for you. Within a storyboard file, each view controller's restoration identifier can be set. Typically, the restoration identifier is set to be the same as the storyboard identifier (which you did not need to use in this chapter). For state restoration to work properly, you will want each view controller to have both a storyboard identifier and a restoration identifier (Figure 28.22).

Figure 28.22 Restoration identifier



Since state restoration is opt-in, you will still need to override the two application delegate methods to tell the system you want states to be saved and restored.

```
- (BOOL)application:(UIApplication *)application
shouldSaveApplicationState:(NSCoder *)coder
{
    return YES;
}

- (BOOL)application:(UIApplication *)application
shouldRestoreApplicationState:(NSCoder *)coder
{
    return YES;
}
```

Finally, your **UIViewController** subclasses will each need to implement the **UIViewControllerRestoration** protocol method to return an instance of the appropriate view controller. Since you are working with storyboards, you will let the storyboard instantiate the view controller for you. Here is an example:

```
+ (UIViewController *)viewControllerWithRestorationIdentifierPath:(NSArray *)path
                                                       coder:(NSCoder *)coder
{
    BNRCOLORViewController *vc = nil;

    UIStoryboard *storyboard =
        [coder decodeObjectForKey:UIStateRestorationViewControllerStoryboardKey];
    if (storyboard)
    {
        vc = (BNRCOLORViewController *)[storyboard
                                         instantiateViewControllerWithIdentifier:@"BNRCOLORViewController"];
        vc.restorationIdentifier = [identifierComponents lastObject];
        vc.restorationClass = [BNRCOLORViewController class];
    }
    return vc;
}
```

The **NSCoder** automatically encodes a reference to the storyboard that you can pull out using the **UIStateRestorationViewControllerStoryboardKey** key. You can then use the storyboard to instantiate the appropriate view controller, passing in the correct storyboard identifier.

Other than that, the rest of state restoration implementation is the same. If view controllers need to save out any information, then they will need to implement the **encodeRestorableStateWithCoder:** and **decodeRestorableStateWithCoder:** methods.

Whether you are using storyboards or not, state restoration is easy to implement and provides a better experience for your users. Be a stylish developer; implement state restoration in your applications today!

This page intentionally left blank

29

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 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.

This page intentionally left blank

Index

Symbols

#import, 44, 64
#pragma mark, 231, 232
%@ token, 38
.h files, 41
.m files, 41, 43
.xcassets (asset catalog), 25
.xcdatamodeld (data model file), 432
@prefix
 creating arrays with, 56
 creating dictionaries with, 224
 creating strings with, 36
 Objective-C keywords, 41
@autoreleasepool, 84
@class, 167
@end, 41
@implementation, 43
@import, 64
@interface
 in header files, 41, 114
@optional, 148
@property, 75
@protocol, 148
@selector(), 207
@synthesize, 81
__cmd, 361
__weak, 73

A

accessor methods, 42-47
 (see also properties)
 customizing, 81
dot syntax and, 46, 47
importance of using, 48
naming conventions for, 139
 properties and, 75-78, 168
accessory indicator (**UITableViewCell**), 170
action methods, 16-18
 connecting in XIB file, 214-216
 and **UIControl**, 246, 247
active state, 354
addKeyframeWithRelativeStartTime:relati...
...veDuration:animations:, 495
addObject:, 36

addSubview:, 90, 93
alignment rectangles, 286, 287
alloc, 30, 31
Allocations instrument, 266-271
ambiguous layout, 299-303, 307
analyzing (code), 276-278, 280
animateKeyframesWithDuration:delay:opti...
...ons:animations:completion:, 495
animateWithDuration:animations:, 492-494
animateWithDuration:delay:options:anim...
...tions:completion:, 493
animations
 basic, 492-494
 with blocks, 493-498
 keyframe, 494-496
 spring, 497, 498
 timing functions, 493, 494
anti-aliasing, 140
API Reference, 97, 98, 346
app delegate, 22
App ID, 24
application bundle
 explained, 365-367
 and internationalization, 480
 mainBundle, 183
 NSBundle, 183
application delegates, 156
application sandbox, 348-350, 365
application states, 353-356, 361, 362
application:didFinishLaunchingWithOptions:,
156
application:shouldRestoreApplicationState:,
456
application:shouldSaveApplicationState:,
456
applicationDidBecomeActive:, 355, 361
applicationDidEnterBackground:, 351, 355,
362
applications
 (see also application bundle, debugging,
 projects, universal applications)
build settings for, 279-281
building, 22, 476
cleaning, 476
data storage, 348, 349, 453
deploying, 23-25
directories in, 348-350
icons for, 25-27

- launch images for, 27
- multiple threads in, 412
- optimizing CPU usage, 271-274
- profiling, 266, 267
- restoring state, 455-466
- running on iPad, 3
- running on simulator, 22
- targets and, 278
- thread safety, 337, 338
- universalizing, 283, 284
- applicationWillEnterForeground:**, 355, 361
- applicationWillResignActive:**, 355, 361
- ARC (Automatic Reference Counting)
 - (see also memory management)
 - benefits of, 69, 70
 - history of, 83
 - vs. manual reference counting, 83
 - overview, 66
 - and strong reference cycles, 70-74
- archiveRootObject:toFile:**, 350
- archiving
 - vs. Core Data, 431
 - described, 345
 - implementing, 345-348
 - with **NSKeyedArchiver**, 350-353
 - thumbnail images, 380
 - when to use, 453
 - and XIB files, 348
- arguments, 31, 32
- arrays
 - allowable contents, 58
 - basics of, 36, 37
 - defined, 34
 - vs. dictionaries, 222
 - fast enumeration of, 37
 - literal access, 59
 - literal creation, 56
 - and memory management, 68
 - and object ownership, 67, 68
 - writing to filesystem, 364
- asset catalogs (Xcode), 25
- assistant editor (Xcode), 197-202, 214-216
- atomic**, 78
- attributes (Core Data), 432-435
- attributes inspector, 13
- authentication challenge, 416
- Auto Layout
 - (see also constraints, Interface Builder)
- alignment rectangles, 286, 287
- ambiguous layout, 299-303, 307
- autoresizing masks and, 318, 319
- debugging, 298-307
- and Dynamic Type, 393
- layout attributes, 286, 287
- misplaced views, 304
- missing constraints, 299-303, 307
- placeholder constraints, 399, 401
- purpose, 285
- unsatisfiable constraints, 303
- auto-completion (in Xcode), 20, 21, 174-176
- _autolayoutTrace**, 307
- autorelease**, 83
- autorelease pool, 83
- autoresizing masks, 309, 318, 319
- autorotation, 321, 323, 324, 326
- availableMediaTypesForSourceType:**, 233
- awakeFromInsert**, 439
- awakeFromNib**, 398

B

- background state, 354-356, 361, 362
- backgroundColor** (**UIView**), 90, 101
- Base internationalization, 473
- baselines, 286
- basic animations, 492-494
- becomeFirstResponder**, 144
- binary numbers, 339, 340
- bitmasks, 339, 340
- bitwise operators, 340
- blocks, 382, 383
 - animation and, 493-498
 - completion, 335-337
 - variable capturing, 385, 386
- bounds, 94
- braces
 - dictionary syntax, 224
 - instance variable declarations, 41
- brackets
 - array syntax, 56, 59
 - arrays and, 59
 - messages and, 31
- breakpoint navigator, 155
- breakpoints, 152, 155
- build configurations, 280
- build settings, 279-281

bundles
application (see application bundle)
identifiers for, 24
NSBundle, 365, 480
settings, 483, 487-489

C

CALayer, 87
callbacks, 360
(see also delegation, notifications, target-action pairs)
camera
(see also images)
recording video, 232-234
taking pictures, 213-220
cancelsTouchesInView, 259
canPerformAction:withSender:, 260
canvas (Interface Builder), 9
cells (see **UITableViewCell**)
CGContextRef, 107, 378
CGPoint, 89
CGRect, 89-91
CGRectMake(), 90
CGSize, 89, 378
class extensions, 114
(see also header files)
class methods, 47, 55-57
classes
(see also *individual class names*)
copying files, 164
creating, 6-8, 39-41
files for, 41, 120
header files of, 41
hierarchy of, 35, 36, 41
inheritance of, 41
overview, 29, 30
prefixes for, 63
reusing, 120, 164
subclasses, 35, 36
subclassing, 38-58, 114
superclasses, 35, 36, 41, 52
visibility, 114
`_cmd`, 361
Cocoa Touch, 29
code paths, 276
code snippet library, 174-176
code-completion (in Xcode), 20, 21, 174-176

compile-time errors, 60
completion blocks, 335-337
concurrency, 412
connections (in Interface Builder), 14-18
connections inspector, 18
console (Xcode), 38
constraints (Auto Layout)
adding programmatically, 312, 313
align, 294, 295
creating in Interface Builder, 287-298
creating programmatically, 310-318
creating with VFL, 310-312
creating without VFL, 316-318
debugging, 298-307
deleting, 292
intrinsic content size, 315, 316
missing, 299-303, 307
nearest neighbor and, 287
overview, 287
pin, 289-294
placeholder, 399, 401
priorities, 298, 315
unsatisfiable, 303
constraintsWithVisualFormat:options:metrics:views:, 311, 312
constraintWithItem:attribute:relatedBy: toItem:attribute:multiplier:constant:, 316-318
content compression resistance priority, 315
content hugging priority, 315
contentMode (UIImageView), 212, 213
contentView (UITableViewCell), 171, 369, 370
contentViewController (UIPopoverController), 326
control events, 246
controller objects, 5, 361
convenience methods, 57
copy (property attribute), 80, 81
copying files, 164
copying objects, 80, 81
Core Data
vs. archiving, 431
attributes, 432-435
entities, 432-437, 444-449
faults, 450-452
fetch requests, 442, 443, 452
fetched property, 452
lazy fetching, 450

logging SQL commands, 449
model configurations, 452
model file, 431-437, 439
NSManagedObjectContext, 439-443
NSManagedObjectModel, 439-441
NSPersistentStoreCoordinator, 439-441
as ORM, 431, 432
relationships, 435-437, 450-452
and SQLite, 431, 439-441, 449
subclassed **NSManagedObject**, 437-439
transforming values, 434
versioning, 452, 453
when to use, 431, 453
Core Graphics, 106-108
Core Graphics (framework), 378
count (NSArray), 37
credentials (web services), 416, 417
curly braces
 dictionary syntax, 224
 instance variable declarations, 41
currentLocale, 470

D

dangling pointers, 67
data model file (Core Data), 431-437, 439
data model inspector, 434
data source methods, 169
data storage
 (see also archiving, Core Data)
for application data, 348, 349
binary, 357, 363
choosing, 453
with I/O functions, 363
for images, 378-380
with **NSData**, 356
dataSource (UITableView), 161, 164-170
dealloc, 69
debug area (Xcode), 38
debug gauges, 263-265
debug navigator, 152
debugger bar, 154
debugging
 (see also debugging tools, exceptions)
Auto Layout, 298-307
exceptions, 60-62
NSError, 363, 364
stack trace, 152
stepping through methods, 154, 155
debugging tools
 Allocations instrument, 266-271
 breakpoints, 152, 155
 debug gauges, 263-265
 debug navigator, 152
 debugger, 151-156
 generation analysis, 270, 271
 Instruments, 265-276
 issue navigator, 23
 stack trace, 152, 153
 static analyzer, 276-278, 280
 Time Profiler, 271-274
 variables view, 153, 154
declarations
 instance variable, 41
 method, 42, 49, 50, 55-57
 property, 75-78
 protocol, 149
decodeRestorableStateWithCoder:, 464
definesPresentationContext, 344
delegation, 148, 149
deleteRowsAtIndexPaths:withRowAnimation:, 188
description (NSObject), 38, 48
designated initializers, 49-54
detail view controllers, 422
developer certificates, 24
developer documentation, 97, 98, 105, 346
device orientation, 321, 323, 324, 326
device type
 determining at run time, 324
 setting, 283
devices
 checking for camera, 217-219
 deploying to, 23, 24
 display resolution, 91
 provisioning, 23-25
 Retina display, 25, 27, 140, 141
dictionaries
 (see also JSON data)
literal creation (@{...}) and access, 224
memory management of, 359
using, 222-226
writing to filesystem, 364
didRotateFromInterfaceOrientation:, 326
directories
 application, 348-350

Documents, 348
Library/Caches, 349
Library/Preferences, 349
lproj, 473, 480
temporary, 349
dismissPopoverAnimated:, 329
dismissViewControllerAnimated:completion:,
335
dispatch_async(), 413
dispatch_once(), 337, 338
display resolution, 91
dock (Interface Builder), 8
documentation, developer, 97, 98, 105, 346
Documents directory, 348
dot syntax, 46, 47
drawing (see views)
drawRect:, 94-105
 and run loop, 112, 113
 and **UITableViewCell**, 369
drill-down interface
 with **UINavigationController**, 191
 with **UISplitViewController**, 421
Dynamic Type, 389-401

E

editButtonItem, 208
editing (**UITableView**,
UITableViewController), 179, 184
editor area (Xcode), 8
encodeInt:forKey:, 346
encodeObject:forKey:, 346
encodeRestorableStateWithCoder:, 463, 464
encodeWithCoder:, 345-347, 350
@end, 41
endEditing:, 205, 229
entities (Core Data), 432-437, 444-449
errors
 compile-time, 60
 and **NSError**, 363, 364
 run-time, 60-62
event loop, 112, 113
events
 callbacks and, 360
 control, 246
 first responder and, 144
 motion, 144
 run loop and, 112, 113

touch (see touch events)
exceptions
 breakpoint for, 155, 156
 diagnosing in debugger, 155, 156
 explained, 60-62
 internal inconsistency, 186
 in Objective-C, 61
 throwing, 331
 unrecognized selector, 61
 using **NSException**, 331
exerciseAmbiguousLayout (UIView), 301, 303

F

fast enumeration, 37, 50, 58
faults, 450-452
fetch requests, 442, 443, 452
fetched property, 452
file inspector, 473
file paths, retrieving, 349, 350
File's Owner, 127-130
files
 copying, 120
 header (.h), 41
 implementation (.m), 41, 43
 importing, 43
 including, 43
filteredArrayUsingPredicate:, 443
first responder
 becoming, 144
 and nil-targeted actions, 247
 for non-touch events, 144
 resigning, 205, 229
 and responder chain, 245, 246
 text fields and, 144
 and **UIMenuController**, 255
FirstResponder, 144
font preferences, 389-393, 395, 396
format strings, 38
forward declarations, 167
frame (in stack), 65
frame (UIView), 89-91, 94
frameworks
 Core Data (see Core Data)
 Core Graphics, 106-108, 378
 Foundation, 63
 importing, 64
 linking, 64

- MobileCoreServices, 234
prefixes and, 63
functions, 30
(see also *individual function names*)
- G**
- generation analysis, 270, 271
genstrings, 478
gestures
(see also **UIGestureRecognizer**,
UIScrollView)
long press, 256, 257
panning, 115, 256-260
taps, 250-256
getter methods, 42-44
graphics context, 107
GUIDs, 225
- H**
- .h files (see header files)
hasAmbiguousLayout (**UIView**), 301
header files
vs. class extensions, 114
description, 41
importing, 43-45, 64
order of declarations in, 56
precompiled, 64
visibility, 114
header view (**UITableView**), 179-184
heap memory, 65, 66
heapshots, 270, 271
hierarchies
class, 35, 36, 41
view, 86-94
Homepwner application
adding an image store, 220
adding Auto Layout constraints, 285-308, 310-318
adding drill-down interface, 191-210
adding Dynamic Type, 389-401
adding item images, 211-230
adding item store, 164-168
adding modal presentation, 329-335
adding preferences to, 484-489
adding state restoration, 455-466
customizing cells, 369-385
enabling editing, 179-190
- localizing, 470-480
moving to Core Data, 431
object diagrams, 164, 194
reusing **BNRItem** class, 164
storing images, 356-358
universalizing, 283, 284
- HTTP protocol, 417-419
- HypnoNerd application
adding a local notification, 135, 136
adding second view controller, 124-130
adding tab bar controller, 130-134
creating, 120-123
- Hypnosister application
creating **BNRHypnosisView**, 88, 89
handling a touch event, 112
object diagram, 114
scrolling, 114-118
- I**
- I/O functions, 363
IBAction, 16, 17, 214-216
IBOutlet, 14, 15, 200, 201
ibtool, 481, 482
icons
(see also images)
application, 25-27
asset catalogs for, 25
camera, 213
id, 50
identity inspector, 129
ignoreSnapshotOnNextApplicationLaunch, 468
image picker (see **UIImagePickerController**)
imageNamed:, 141
imagePickerController: didFinishPicking...MediaWithInfo:, 218
imagePickerControllerDidCancel:, 218
images
(see also camera, icons, **UIImageView**)
archiving, 378-380
caching, 356-358
creating thumbnail, 377-380
displaying in **UIImageView**, 212, 213
manipulating in offscreen contexts, 377-380
for Retina display, 140
storing, 220-225
wrapping in **NSData**, 356
imageWithContentsOfFile:, 357

@implementation, 43
implementation files, 41, 43
#import, 44
importing files, 43, 44, 64
inactive state, 354
including files, 43
inheritance, single, 38, 41
init
 alloc and, 31
 overriding, 53
 overview, 49-54
initialize, 483
initializers, 49-54
 (see also **init**)
designated, 49-54
disallowing calls to, 331
 and singletons, 165-168
initWithCoder:, 345, 347, 348
initWithContentsOfFile:, 363, 364
initWithFrame:, 89
initWithStyle:, 162
insertObjectAtIndex:, 36, 59
inspectors (Xcode)
 attribute, 13
 connections, 18
 data model, 434
 file, 473
 identity, 129
 size, 315
instance methods, 47
instance variables
 (see also pointers, properties)
accessor methods for, 42
declaring, 41, 42
description, 30
explained, 42
in memory, 66
visibility, 114
 and weak references, 74
instances, 30, 31
instancetype, 50
Instruments, 265-276
integerForKey:, 485
@interface
 in class extensions, 114
 in header files, 41
Interface Builder
 (see also Xcode)

canvas, 8
connecting objects, 14-18
connecting with source files, 197-202, 372
creating objects in, 10-13
dock, 8
editing XIB files, 8-18
explained, 8
making connections in, 197-202
placeholders in, 20
and properties, 372
setting outlets in, 15, 16, 200
setting target-action in, 17, 18
simulated metrics, 199
when to use, 124
interface files (see header files)
interface orientation, 321, 323, 324, 326
interfaceOrientation, 326
internal inconsistency exception, 186
internationalization, 469-472, 480
 (see also localization)
intrinsic content size, 315, 316
iOS simulator
 killing apps in, 455
 low-memory warnings and, 360
 multiple touches in, 157
 rotating in, 322
 running applications on, 22
 sandbox location, 351
 saving images to, 219
 viewing application bundle in, 365
iPad
 (see also devices)
 application icons for, 25
 launch images for, 27
 running iPhone applications on, 3
isa pointer, 60
isEqual:, 187
isSourceTypeAvailable:, 217
issue navigator, 23

J

JSON data, 409-412

K

key-value coding (KVC), 138, 139
key-value pairs, 222-225
keyboard

- appearance, 145, 146
 - dismissing, 228
 - number pad, 210
 - keyframe animations, 494-496
 - keys (in dictionaries), 222-228
 - KUTTypeImage**, 233, 234
 - KUTTypeMovie**, 233, 234
 - KVC (key-value coding), 138, 139
- L**
- labels (in message names), 32
 - language settings, 469, 476
 - launch images, 27, 455, 468
 - layers (of views), 87, 88
 - layout attributes, 286, 287
 - lazy loading, 121, 136, 137
 - Leaks instrument, 274-276
 - leaks, memory, 67, 70
 - libraries
 - (see also frameworks)
 - code snippet, 174-176
 - object, 10
 - Library/Caches** directory, 349
 - Library/Preferences** directory, 349
 - line numbers, showing, 277
 - loadView**, 121, 122, 184
 - local notifications, 135
 - local variables, 65, 66
 - Localizable.strings**, 478
 - localization
 - Base internationalization and, 473
 - using ibtool, 481, 482
 - internationalization, 469-472, 480
 - lproj directories, 473, 480
 - NSBundle**, 480
 - of preferences, 489
 - resources, 473-476
 - strings tables, 477-480
 - user settings for, 469, 476
 - XIB files, 473-476
 - localizedDescription**, 363
 - locationInView:**, 254
 - low-memory warnings, 220
 - lproj directories, 473, 480
- M**
- .m files, 41, 43
 - mach_msg_trap()**, 273
 - macros, preprocessor, 281
 - main bundle, 183 (see application bundle)
 - main thread, 412
 - main()**, 36, 156
 - main.m**, 36
 - mainBundle**, 183, 367
 - manual reference counting, 83
 - master view controllers, 422, 425
 - mediaTypes**, 232
 - memory, 65, 66
 - memory management
 - with ARC, 66
 - arrays, 68
 - dangling pointers, 67
 - dictionaries, 225, 359
 - Leaks instrument, 274-276
 - need for, 66
 - and object ownership, 67-70
 - optimizing with Allocations instrument, 266-271
 - pointers, 67-70
 - premature deallocation, 67
 - and properties, 79
 - and strong reference cycles, 70-74
 - strong and weak references, 70-74
 - UITableViewCell**, 173
 - memory warnings, 220
 - menus (**UIMenuController**), 254-256, 260
 - messages, 31-33
 - (see also methods)
 - methods
 - (see also *individual method names*)
 - accessor, 42-47
 - action, 16-18, 246, 247
 - class, 47, 55-57
 - convenience, 57
 - data source, 169
 - declaring, 42, 49, 50, 55
 - defined, 30
 - designated initializer, 49-54
 - vs. functions, 30
 - implementing, 43, 44, 49, 51
 - initializer, 49-54
 - instance, 47
 - vs. messages, 32
 - names of, 32

overriding, 48, 49, 52-54
stepping through, 154, 155
minimumPressDuration, 256
misplaced views, 304
missing constraints, 299-303, 307
MobileCoreServices, 234
.mobileprovision files, 24
modal view controllers
 defined, 218
 dismissing, 332-334
 and non-disappearing parent view, 335
 relationships of, 342
 in storyboards, 507-511
 styles of, 334
 transitions for, 337
modalPresentationStyle, 334, 344
modalTransitionStyle, 337
modalViewControllerAnimated, 332-334
model file (Core Data), 431-437, 439
model objects, 5
Model-View-Controller (MVC), 4-6, 361
Model-View-Controller-Store (MVCS), 361
motion effects, 151
motion events, 144
multi-threading, 337, 338, 412
multi-touch, enabling, 241
multipleTouchEnabled (**UIView**), 241
MVC (Model-View-Controller), 4-6, 361
MVCS (Model-View-Controller-Store), 361

N

namespaces, 63
naming conventions
 accessor methods, 43, 139
 cell reuse identifiers, 173
 class prefixes, 63
 initializer methods, 49
 XIB files, 135
navigation controllers (see
UINavigationController)
navigationController, 202, 341
navigationItem (**UIViewController**), 206
navigators (Xcode)
 breakpoints, 155
 debug, 152
 defined, 4
 issue, 23
project, 3, 4
nearest neighbor, 287
Nerdfeed application
 adding **UIWebView**, 413-415
 fetching data, 406-408
 using **UISplitViewController**, 422-424
nested message sends, 31
nextResponder, 245
NIB files
 (see also XIB files)
 and archiving, 348
 awakeFromNib, 398
 explained, 13
 key-value coding and, 138
 loading, 126
 loading manually, 373
 XIB files and, 13
nibWithNibName:bundle:, 373
nil
 and arrays, 59
 returned from initializer, 52
 sending messages to, 32
 setting pointers to, 32
 -targeted actions, 247
 as zero pointer, 32
nonatomic, 78
notifications (**NSNotificationCenter**), 358-360
 of low-memory warnings, 358
 settings change, 489
notifications, local, 135
notifications, push, 135
NSArray
 (see also arrays)
 basics, 36, 37
 count, 37
 details, 58-60
 literal creation (@[...]), 56
 objectAtIndex:, 59
NSBundle, 183, 480
NSCoder, 345, 347
 and state restoration, 463, 464
NSCoding protocol, 345-348
NSData, 59, 356, 380, 483
NSDate, 204, 364
NSDateFormatter, 204, 470
NSDictionary, 222-225
 (see also dictionaries)
NSError, 363, 364

NSEException, 331
NSEXpression, 452
NSFetchRequest, 442, 443, 452
NSGlobalDomain, 486
NSIndexPath, 172, 188
NSInteger, 170
NSJSONSerialization, 410
NSKeyedArchiver, 350-353
NSKeyedUnarchiver, 352
NSLayoutConstraint, 311
NSLocale, 470
NSLocalizedString(), 477, 479
NSLog(), 38
NSManagedObject, 437-439, 452
NSManagedObjectContext, 439-443, 452
NSManagedObjectModel, 439-441
NSMutableArray
 (see also arrays)
 basics, 36
 details, 58-60
 insertObject:atIndex:, 59
 removeObject:, 187
 removeObjectIdenticalTo:, 187
 replaceObjectAtIndex:withObject:, 59
NSMutableDictionary, 221-225
 (see also dictionaries)
NSNotificationCenter, 358-360
NSNull, 59
NSNumber, 59, 364
NSObject, 35, 36, 38-41
 dealloc, 69
 description, 38, 48
NSPersistentStoreCoordinator, 439-441
NSPredicate, 442
NSSearchPathDirectory, 349
NSSearchPathForDirectoriesInDomains, 349
 NSSortOrdering, 452
NSString
 basics, 36
 creating, 36, 57
 internationalizing, 477
 literal creation (@"..."), 36
 localizing, 481, 482
 NSLog() and, 38
 property list serializable, 364
 stringWithFormat:, 57
 using tokens with, 38
 writing to filesystem, 357-363
NSStringFromSelector, 361
NSTemporaryDirectory, 349
NSUInteger, 170
NSURL, 406-408
NSURLCredential, 417
NSURLRequest, 406-408, 418, 419
NSURLSession, 407-409, 416
NSURLSessionAuthChallengeUseCredential, 417
NSURLSessionConfiguration, 408
NSURLSessionDataDelegate (protocol), 416
NSURLSessionDataTask, 408, 409, 411-413
NSURLSessionTask, 407, 417
NSUserDefaults, 349, 483
NSUserDefaultsDidChangeNotification, 489
NSUUID, 225, 226
NSNumber, 59
NSNumberTransformer, 434
number pad, 210

O

objc_msgSend(), 274
object library, 10, 11
Object-Relational Mapping (ORM), 431
objectAtIndex: (NSArray), 59
objectForKey: (NSDictionary), 222-225
Objective-C
 @ prefix, 41
 basics, 29-62
 keywords, 41
 message names, 32
 method names, 32
 naming conventions, 43, 49
 single inheritance in, 41
objects
 (see also classes, memory management)
 allocation, 66
 copying, 80, 81
 independence of, 42
 in memory, 66
 overview, 29-31
 ownership of, 66-70
 property list serializable, 364
 size of, 66
offscreen contexts, 377, 378
optional methods (in protocols), 148
Organizer window (Xcode), 24

orientation (UIDevice**)**, 321
ORM (Object-Relational Mapping), 431
outlets
 connecting with source files, 372
 declared as weak, 130
 defined, 14
 setting, 14-16, 197
overriding methods, 48, 49, 52-54

P

parallax, 151
parentViewController, 332-334
pathForResource:type:, 480
pixels, 91
placeholder objects, 128
placeholders (in code), 20, 176
pointers
 in arrays, 35
 in Interface Builder (see outlets)
 and memory management, 67-70
 overview, 30, 31
 setting in XIB files, 15, 16
 setting to nil, 32
 as strong references, 70
 syntax of, 42
 as weak references, 70, 73, 74
points (vs. pixels), 91
popover controllers, 326-328, 425
popoverControllerDidDismissPopover:, 328
#pragma mark, 231, 232
precompiled header files, 64
predicates (fetch requests), 442
predicateWithFormat:, 442
preferences
 (see also Dynamic Type, localization)
 available defaults, 486
 font, 389-393, 395, 396
 global constants for, 484
 localizing, 489
 location of, 483
 reading, 485
 registering defaults, 484
 settings bundle and, 487-489
 updating, 485-489
preferredContentSizeCategory (UIApplication**)**, 395
preferredFontForTextStyle: (UIFont**)**, 391
 premature deallocation, 67
prepareForSegue:sender:, 516
 preprocessor macros, 281
presentedViewController, 342
presentingViewController, 333, 342
presentViewController:animated:completion:, 218, 335
 products, 278
 profiling (applications), 266, 267
 project and targets list (Xcode), 278
 project navigator, 3, 4
projects
 build settings for, 279-281
 cleaning and building, 476
 copying files to, 164
 creating, 2-4
 defined, 278
 target settings in, 365
 templates for, 2
properties
 accessor methods and, 168
 atomic, 78
 attributes of, 78-81
 copy, 80, 81
 creating from XIB file, 372
 creating in Interface Builder, 372
 custom accessors for, 81
 declaring, 75-78
 without instance variables, 82
 memory management of, 79
 nonatomic, 78
 overriding accessors, 81
 readonly, 79
 readwrite, 79
 strong, 79
 synthesizing, 81, 168
 visibility, 114
 weak, 79
property list serializable objects, 364
protocols
 declaring, 149
 delegate, 148, 149
 described, 148
 NSCoding, 345-348
 NSURLSessionDataDelegate, 416
 optional vs. required methods, 148
 structure of, 148
UIApplicationDelegate, 355

- UIDataSourceModelAssociation, 466
UIGestureRecognizerDelegate, 257
UIImagePickerControllerDelegate, 218-220
UINavigationControllerDelegate, 219
UIPopoverControllerDelegate, 327
UIResponderStandardEditActions, 260
UISplitViewControllerDelegate, 426
UITableViewDataSource, 161, 169-171, 188
UITableViewDelegate, 161
UITextFieldDelegate, 148, 228
UIViewControllerRestoration, 461
provisioning profiles, 24
push notifications, 135
pushViewController:animated:, 202, 203
- Q**
- Quartz (see Core Graphics)
Quick Help, 346
Quiz application, 2-27
- R**
- RandomItems application
 creating, 33-38
 creating **BNRItem** class, 39-58
readonly, 79
readwrite, 79
receiver, 31
reference pages, 97, 98, 346
region settings, 469
relationships (Core Data), 435-437, 450-452
release, 83
removeObject:, 187
removeObjectIdenticalTo:, 187
reordering controls, 189
replaceObjectAtIndex:withObject:, 59
required methods (in protocols), 148
requireGestureRecognizerToFail:, 260
resignFirstResponder, 144, 150, 205
resources
 asset catalogs for, 25
 defined, 25, 365
 localizing, 473-476
responder chain, 245, 246
responders (see first responder, **UIResponder**)
respondsToSelector:, 148
restoration classes, 456
restoration identifiers, 456-459
- retain**, 83
Retina display, 25, 27, 140, 141
reuseIdentifier (**UITableViewCell**), 173
reusing
 classes, 164
 table view cells, 173, 174
Root.plist, 488, 489
rootViewController
 (**UINavigationController**), 192-195
rootViewController (**UIWindow**), 122
rotation, handling, 321, 323, 324, 326
rows (**UITableView**)
 adding, 185, 186
 deleting, 187, 188
 moving, 188-190
run loop, 112, 113, 156
run-time errors, 60-62
- S**
- Sample Code (documentation), 105
sandbox, application, 348-350, 365
schemes, 22, 24
scrolling, 114-117
SDK Guides (documentation), 105
sections (**UITableView**), 170, 179
SEL, 207
selector, 31, 207
self, 51, 57
sendAction:to:from:forEvent:, 247
sendActionsForControlEvents:, 247
setEditing:animated:, 184, 208
setNeedsDisplay (**UIView**), 113
setNeedsDisplayInRect (**UIView**), 113
setObject:ForKey:, 222-225
setPagingEnabled:, 118
setStroke (**UIColor**), 101
setter methods, 42-44
settings (see preferences)
Settings application, 349
settings bundle, 483, 487-489
setValue:ForKey:, 138
simulated metrics, 199
simulator
 killing apps in, 455
 low-memory warnings and, 360
 multiple touches in, 157
 rotating in, 322

running applications on, 22
sandbox location, 351
saving images to, 219
viewing application bundle in, 365
single inheritance, 38, 41
singletons
 implementing, 165-168
 thread-safe, 337, 338
size inspector, 315
sort descriptors (**NSFetchRequest**), 442
sourceType (UIImagePickerController), 216, 217
split view controllers (see **UISplitViewController**)
splitViewController, 341, 424
spring animations, 497, 498
SQL, 449
SQLite, 431, 439-441, 449
square brackets
 array syntax, 56, 59
 arrays and, 59
 messages and, 31
SSL (Secure Sockets Layer), 416, 417
stack (memory), 65, 153
stack trace, 152, 153
standardUserDefaults, 483
state restoration
 and application life cycle, 459, 460
 controlling snapshots, 467
 explained, 455, 456
 and **NSCoder**, 463
 opting in to, 456
 restoration identifiers, 456-459
 with storyboards, 520
UIViewControllerRestoration protocol, 461
 for views, 465
states, application, 353-356
static analyzer, 276-278, 280
static tables, 503-506
static variables, 166
store objects, 361
storyboards
 creating, 499-503
 pros and cons, 519, 520
 segues, 506-519
 state restoration and, 520
 static tables in, 503-506
 vs. XIB files, 499
strings (see **NSString**)
strings tables, 477-480
stringWithFormat:, 57
strong, 79
strong reference cycles, 70-74
 finding with Leaks instrument, 274-276
structures (C), 29, 30
subclasses, 35, 36
subclassing, 38-58, 114
 (see also overriding methods)
 method return types, 50
 use of **self**, 57
subviews, 86
super, 52
superclasses, 35, 36, 41, 52
Superview, 91
supportedInterfaceOrientations, 323
suspended state, 354, 355

T

tab bar controllers (see **UITabBarController**)
tab bar items, 132-134
tabBarController, 341
table view cells (see **UITableViewCell**)
table view controllers (see **UITableViewController**)
table views (see **UITableView**)
tables (databases), 431
tableView, 186
tableView:cellForRowAtIndexPath:, 170-174
tableView:commitEditingStyle:forRowAtIndexPath:, 188
tableView:didSelectRowAtIndexPath:, 204
tableView:moveRowAtIndexPath:toIndexPath:, 188, 189
tableView:numberOfRowsInSection:, 170
target-action pairs
 defined, 16-18
 setting programmatically, 207
 and **UIControl**, 246, 247
 and **UIGestureRecognizer**, 250
targets
 build settings for, 279-281, 365
 defined, 278
templates (Xcode), xvii, 2
text styles, 390
textFieldShouldReturn:, 146, 228

threads, 337, 338, 412
thumbnail images, creating, 377-380
Time Profiler instrument, 271-274
timing functions, 493, 494
tmp directory, 349
toggleEditingStyle:, 184
tokens, 38
topViewController (UINavigationController), 193
touch events
 (see also **UIGestureRecognizer**)
 basics of, 235, 236
 enabling multi-touch, 241-245
 and responder chain, 245, 246
 and target-action pairs, 246, 247
 and **UIButton**, 246, 247
touchesBegan:withEvent:, 235, 236
touchesCancelled:withEvent:, 236
touchesEnded:withEvent:, 236
touchesMoved:withEvent:, 235, 236
TouchTracker application
 drawing lines, 237-245
 recognizing gestures, 249-260
transformable attributes (Core Data), 434
translationInView:, 258

U

UI thread, 412
UIAlertView, 363
UIApplication
 and events, 236
 and **main()**, 156
 and responder chain, 245, 247
UIApplicationDelegate, 355
UIApplicationDidBecomeActiveNotification, 467
UIApplicationDidReceiveMemoryWarningNotification, 358
UIApplicationWillResignActiveNotification, 467
UIBarButtonItem, 207-209, 213-216, 229
UIBezierPath, 96-105
UICollectionView, 387, 388
UIColor, 91, 100, 101
UIContentSizeCategoryDidChangeNotification, 392
UIButton, 229, 246, 247
UIControlEventTouchUpInside, 246, 247
UIDataSourceModelAssociation (protocol), 466
UIFont, 391
UIGestureRecognizer
 action messages of, 250, 256
 cancelsTouchesInView, 259
 chaining recognizers, 260
 delaying touches, 260
 described, 249
 detecting taps, 250-256
 enabling simultaneous recognizers, 257
 implementing multiple, 252-254, 257-260
 intercepting touches from view, 250, 257, 259
 locationInView:, 254
 long press, 256, 257
 panning, 256-260
 state (property), 256, 258, 261
 subclasses, 250, 261
 subclassing, 261
 translationInView:, 258
 and **UIResponder** methods, 259
UIGestureRecognizerDelegate, 257
UIGraphics functions, 378
UIGraphicsBeginImageContextWithOptions, 378
UIGraphicsEndImageContext, 378
UIGraphicsGetImageFromCurrentImageContext, 378
UIImage, 356
 (see also images, **UIImageView**)
UIImageJPEGRepresentation, 356
UIImagePickerController
 instantiating, 216-218
 on iPad, 326
 presenting, 218-220
 recording video with, 232-234
 in **UIPopoverController**, 326
UIImagePickerControllerDelegate, 218-220
UIImageView, 212, 213
UIInterpolatingMotionEffect, 492
UILocalNotification, 135
UILongPressGestureRecognizer, 256, 257
UIMenuController, 254-256, 260
UIModalPresentationCurrentContext, 344
UIModalPresentationFormSheet, 334
UIModalPresentationPageSheet, 334
UIModalTransitionStyleCoverVertical, 337
UIModalTransitionStyleCrossDissolve, 337

UIModalTransitionStyleFlipHorizontal, 337
UIModalTransitionStylePartialCurl, 337
UINavigationBar, 193, 195-209
UINavigationController
 (see also view controllers)
 adding view controllers to, 202, 203, 205
 described, 192-196
 instantiating, 195
 managing view controller stack, 192
 navigationController, 341
 pushViewController:animated:, 202, 203
 rootViewController, 192-194
 in storyboards, 506, 507
 topViewController, 192-194
 and **UINavigationBar**, 206-209
 view, 193
 viewControllers, 193
 viewWillAppear:, 205
 viewWillDisappear:, 205
UINavigationControllerDelegate, 219
UINavigationItem, 206-209
UINib, 373
UIPanGestureRecognizer, 256-260
UIPopoverController, 326-328, 425
UIPopoverControllerDelegate, 327
UIResponder
 described, 144
 menu actions, 260
 and responder chain, 245, 246
 and touch events, 235
UIResponderStandardEditActions (protocol), 260
UIScrollView, 114-117
UISplitViewController
 illegal on iPhone, 422
 master and detail view controllers, 422-425
 overview, 422-424
 in portrait mode, 425-427
 splitViewController, 341
UISplitViewControllerDelegate, 426
UIStoryboard, 499-520
UIStoryboardSegue, 506-519
UITabBarController
 implementing, 130-134
 tabBarController, 341
 vs. **UINavigationController**, 191
 view, 131
UITabBarItem, 132-134
UITableView, 159-161
 (see also **UITableViewCell**,
 UITableViewcontroller)
 adding rows to, 185, 186
 deleting rows from, 187, 188
 editing mode of, 179, 184, 185, 208, 370
 editing property, 179, 184, 185
 footer view, 179
 header view, 179-184
 moving rows in, 188-190
 populating, 164-173
 sections, 170, 179
 view, 163
UITableViewCell
 adding images to, 377-380
 cell styles, 171
 contentView, 171, 369, 370
 creating interface with XIB file, 371, 372
 editing styles, 188
 relaying actions from, 380-385
 retrieving instances of, 171-173
 reusing instances of, 173, 174
 subclassing, 369-374
 subviews, 170, 171
 UITableViewCellStyle, 171
UITableViewCellEditingStyleDelete, 188
UITableViewcontroller
 (see also **UITableView**)
 adding rows, 185, 186
 creating in storyboard, 503-506
 creating static tables, 503-506
 data source methods, 169
 dataSource, 164-170
 deleting rows, 187, 188
 described, 161
 designated initializer, 162
 editing property, 184
 initWithStyle:, 162
 moving rows, 188-190
 returning cells, 171-174
 subclassing, 161-163
 tableView, 186
 UITableViewStyleGrouped, 162
 UITableViewStylePlain, 162
UITableViewDataSource (protocol), 161, 169-171, 188
UITableViewDelegate, 161
UITapGestureRecognizer, 250-256

UITextField
as first responder, 228, 247
setting attributes of, 210
UITextInputTraits, 145, 146
UITextFieldDelegate, 148, 228
UITextInputTraits (UITextField), 145, 146
UIToolbar, 207, 213
UITouch, 236, 240-245
UIUserInterfaceIdiomPad, 324
UIUserInterfaceIdiomPhone, 324
UIView
(see also **UIViewController**, views)
backgroundColor, 90, 101
bounds, 94
defined, 86
drawRect:, 94-105, 112, 113
endEditing:, 205
exerciseAmbiguousLayout, 301, 303
frame, 89-91, 94
hasAmbiguousLayout, 301
instantiating, 89
setNeedsDisplay, 113
setNeedsDisplayInRect:, 113
subclassing, 88-94
superview, 91
UIViewController
(see also **UIView**, view controllers)
definesPresentationContext, 344
didRotateFromInterfaceOrientation:, 326
instantiating, xvii
interfaceOrientation, 326
loadView, 121, 122, 184
modalTransitionStyle, 337
modalViewController, 332-334
navigationController, 202
navigationItem, 206
parentViewController, 332-334
presentingViewController, 333
splitViewController, 424
supportedInterfaceOrientations, 323
tabBarItem, 132
view, 121, 128, 137, 245
viewControllers, 341
viewDidLayoutSubviews, 301
viewDidLoad, 137
viewWillAppear:, 137, 220
willAnimateRotationToInterfaceOrientation...
...ation:duration:, 325
and XIB files, xvii
UIViewControllerRestoration (protocol), 461
UIWebView, 413-415
UIWindow, 86
and responder chain, 245
rootViewController, 122
unarchiveObjectWithFile:, 352
universal applications
accommodating device differences, 422, 429
creating, 283, 284
defined, 283
setting device family, 428
using iPad-only classes, 424
unrecognized selector, 61
unsatisfiable constraints, 303
URLs, 407
(see also **NSURL**)
user interface
(see also Auto Layout, views)
drill-down, 191, 421
handling rotation, 321, 323, 324, 326
keyboard, 228
orientation of, 321, 323, 324, 326
scrolling, 114-117
user settings, 483 (see preferences)
userInterfaceIdiom, 324
utility area, 174
utility area (Xcode), 10
UUIDs, 225

V

valueForKey:, 138
variables
(see also instance variables, local variables, pointers, properties)
local, 65, 66
static, 166
variables view, 153, 154
VFL (Visual Formal Language), 310, 311
video recording, 232-234
view (UIViewController), 121, 137
view controllers
(see also **UIViewController**, views)
adding to navigation controller, 202, 203
adding to popover controller, 327
adding to split view controller, 422-424
creating in a storyboard, 499-520

defined, 119
detail, 422
families of, 341, 342
lazy loading of views, 121, 136, 137
loading views, 128
master, 422, 425
modal, 218
passing data between, 203, 204
presenting, 130
relationships between, 340-344
reloading subviews, 220
role in application, 119
and state restoration, 456-466
and view hierarchy, 121
view hierarchy, 86-94, 121
view controllers, 341
viewControllers (UINavigationController), 193
viewControllerWithRestorationIdentifier..._Path:coder:, 462
viewDidLayoutSubviews (UIViewController), 301
viewDidLoad, 137
views
(see also Auto Layout, touch events, **UIView**, view controllers)
adding to window, 86, 121
animating, 491-498
creating custom, 88-94
defined, 86
drawing shapes, 96-105
drawing to screen, 87, 88, 94, 112, 113
in hierarchy, 86-88
layers and, 87, 88
lazy loading of, 121, 136
loading, 128
modal presentation of, 218
in Model-View-Controller, 4
redrawing, 112, 113
rendering, 87, 88, 94, 112, 113
resizing, 212, 213
and run loop, 112, 113
scrolling, 114-117
size and position of, 89-91, 94
and state restoration, 465
and subviews, 86-94
viewWillAppear:, 137, 204, 205, 220
viewWillDisappear:, 205

visibility, 114
Visual Formal Language (VFL), 310, 311

W

weak, 79
weak references, 70, 73, 74, 79
web services
authentication, 416, 417
credentials, 416, 417
for data storage, 453
and HTTP protocol, 417-419
implementing, 406-413
with JSON data, 409-412
NSURLSession, 408, 409
overview, 404
requesting data from, 406-409
SSL (Secure Sockets Layer), 416, 417
willAnimateRotationToInterfaceOrientation..._on:duration:, 325
workspaces (Xcode), 3
writeToFile:atomically:, 356
writeToFile:atomically:encoding:error:, 363

X

.xcassets (asset catalog), 25
.xcdatamodeld (data model file), 432
Xcode
(see also debugging tools, Instruments, Interface Builder, projects, iOS simulator)
API Reference, 97, 98, 346
application templates, 2
asset catalogs, 25
assistant editor, 197-202, 214-216
attributes inspector, 13
build settings, 279-281
building interfaces, 8-18
canvas, 9
code snippet library, 174-176
code-completion, 20, 21, 174-176
console, 38
creating classes, 6-8
creating projects in, 2-4
data model inspector, 434
debug area, 38
debug gauges, 263-265
debugger, 151-156

- documentation browser, 97, 98
 - editor area, 8
 - file inspector, 473
 - identity inspector, 129
 - inspectors, 10
 - issue navigator, 23
 - keyboard shortcuts, 202
 - library, 10
 - line numbers in, 277
 - navigator area, 4
 - navigators, 3
 - object library, 10, 11
 - Organizer window, 24
 - placeholders in, 176
 - products, 278
 - profiling applications in, 266, 267
 - project and targets list, 278
 - project navigator, 4
 - projects, 278
 - Quick Help, 346
 - schemes, 22, 24
 - size inspector, 315
 - static analyzer, 276-278, 280
 - tabs, 202
 - targets, 278
 - templates, 88
 - utility area, 10, 174
 - versions, 2
 - workspaces, 3
- XIB files
- (see also Interface Builder, NIB files)
 - alternate, 308
 - and archiving, 348
 - bad connections in, 201
 - Base internationalization and, 473
 - basics, 8
 - connecting with source files, 197-202, 214-216, 372
 - creating properties from, 372
 - defined, 8
 - editing, 8-18
 - File's Owner, 127-130
 - `~ipad` and `~iphone`, 308
 - loading manually, 183
 - localizing, 473-476
 - making connections in, 214-216
 - naming, 135
 - vs. NIB files, 13
- placeholders in, 128
- and properties, 372
- simulated metrics, 199
- vs. storyboards, 499
- when to use, 124
- XML property lists, 364

Tabs

- ⌘T New Tab
- ⌘W Close Tab
- ⇧⌘} Next Tab
- ⇧⌘{ Previous Tab

Run

- ⌘R ▶ Run
- ⇧⌘B ⌘ Analyze
- ⌘U ⌘ Test
- ⌘. ■ Stop
- ⌘I ⌘ Profile
- ⌘< Edit Scheme...

Window

- ⇧⌘2 Open Organizer

Editor Mode

- ⌘↔ Standard
- ⌥⌘↔ Assistant
- ⇧⌥⌘↔ Version

View

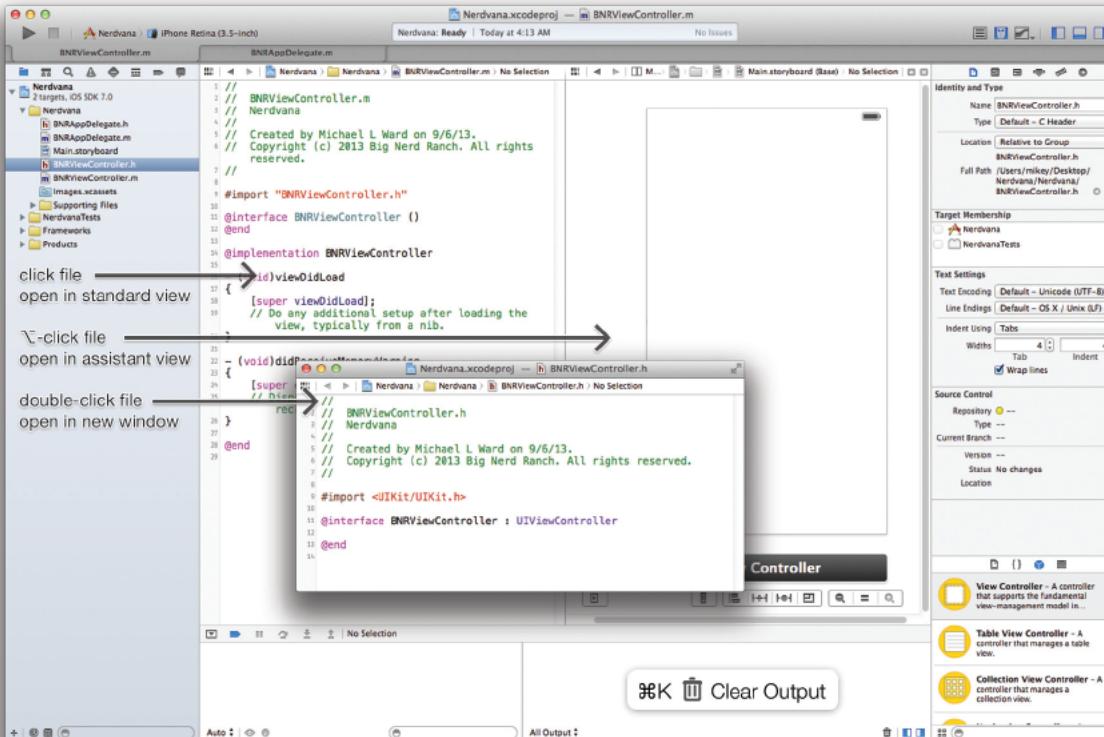
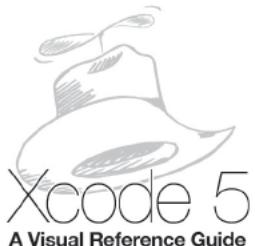
- ⌘0 Navigator
- ⇧⌘Y Debug
- ⌥⌘0 Utilities

Navigators

- ⌘1 Project
- ⌘2 Symbol
- ⌘3 Search
- ⌘4 Issue
- ⌘5 Tests
- ⌘6 Debug
- ⌘7 Breakpoint
- ⌘8 Log

Debug

- ⌘Y ⚡ Breakpoints
- ⌃⌘Y ⚡ Pause
- F6 ⚡ Step Over
- F7 ⚡ Step Into
- F8 ⚡ Step Out



Focus

- ⌃⌘. Move Focus To Next Area
- ⌃⌘> Move Focus To Previous Area



KEY

- ⌘ command
- ⇧ shift
- ⌥ option
- ⌃ control
- ↵ return

Xcode 5 Text Commands

KEY

⌘ command
⌃ shift

⌥ option
⌅ control
⌿ escape

Basics

⌘C Copy
⌘X Cut
⌘V Paste
⌘Z Undo

Find

⌘F Find...
⌃⌘F Find & Replace...
⌘G Find Next
⌃⌘G Find Previous

Line (or selection)

⌘] →Shift Right
⌘[Shift Left ⌂←
⌅| Re-Indent
⌘/ //Comment

Jump to...

→ Next Character
⌅→ Next Subword
⌃→ Next Word
⌘→ End of Line

```
return sampleClass;  
return sampleClass;  
return sampleClass;  
return sampleClass;
```

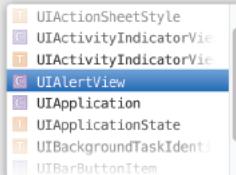


⌅⌘↑

Toggle to Counterpart

Hold ⌄ to **Highlight** Use ← to move backwards

⌿ Show Completion List
⌅. Next Completion
⌅> Previous Completion



⌘+click OR ⌅⌘D
Definition in Standard Window

⌃⌘+click OR ⌅⌃⌘D
Definition in Assistant Window

⌘+double-click
Definition in New Window

⌅/ Next Placeholder
⌅? Previous Placeholder

initWithTitle:(NSString *) message:(NSString *)
initWithTitle:(NSString *) message:(NSString *)



Big
nerd
ranch

Training • Books • Consulting

www.bignerdranch.com

(404) 478-9005