



iOS PROGRAMMING

THE BIG NERD RANCH GUIDE
3RD EDITION

JOE CONWAY & AARON HILLEGASS

IOS PROGRAMMING

THE BIG NERD RANCH GUIDE

JOE CONWAY & AARON HILLEGASS



Big
nerd
ranch

iOS Programming: The Big Nerd Ranch Guide

by Joe Conway and Aaron Hillegass

Copyright © 2012 Big Nerd Ranch, Inc.

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, Inc.
1989 College Ave.
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, Inc.

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 0321821521
ISBN-13 978-0321821522

Third edition, second printing, August 2012

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 Scott Ritchie, Brian Hardy, Mikey Ward, Christian Keur, Alex Silverman, Owen Matthews, Brian Turner, Juan Pablo Claude, and Bolot Kerimbaev.
- Our tireless editor, Susan Loper, took our distracted mumblings and made them into readable prose.
- Our technical reviewers, Bill Monk and Jawwad Ahmad, helped us find and fix flaws.
- Ellie Volckhausen designed the cover. (The photo is of the bottom bracket of a bicycle frame.)
- Chris Loper at IntelligentEnglish.com designed and produced the print book and the EPUB and Kindle versions.
- The amazing team at Pearson Technology Group patiently guided us through the business end of book publishing.

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

Table of Contents

Introduction	xiii
Prerequisites	xiii
What's Changed in the Third Edition?	xiii
Our Teaching Philosophy	xiv
How To Use This Book	xiv
How This Book Is Organized	xv
Style Choices	xvii
Typographical Conventions	xvii
Necessary Hardware and Software	xviii
1. A Simple iOS Application	1
Creating an Xcode Project	2
Building Interfaces	5
Model-View-Controller	9
Declarations	11
Declaring instance variables	12
Declaring methods	12
Making Connections	13
Setting pointers	14
Setting targets and actions	15
Summary of connections	17
Implementing Methods	17
Build and Run on the Simulator	20
Deploying an Application	22
Application Icons	24
Launch Images	25
2. Objective-C	29
Objects	29
Using Instances	30
Creating objects	30
Sending messages	31
Destroying objects	32
Beginning RandomPossessions	33
Creating strings	36
Format strings	37
NSArray and NSMutableArray	37
Subclassing an Objective-C Class	38
Creating an NSObject subclass	39
Instance variables	42
Accessor methods	43
Instance methods	46
Initializers	47
Other initializers and the initializer chain	51
Using Initializers	52
Class methods	53
Testing your subclass	55

- Exceptions and Unrecognized Selectors 56
- Fast Enumeration 57
- Challenges 58
 - Bronze Challenge: Bug Finding 58
 - Silver Challenge: Another initializer 58
 - Gold Challenge: Another Class 58
 - Are You More Curious? 59
 - For the More Curious: Class Names 59
- 3. Managing Memory with ARC 61
 - The Heap 61
 - The Stack 62
 - Pointer Variables and Object Ownership 63
 - Memory Management 64
 - Using ARC for memory management 64
 - How objects lose owners 65
 - Strong and Weak References 67
 - Properties 72
 - Declaring properties 72
 - Synthesizing properties 74
 - Instance variables and properties 76
 - Copying 76
 - Dot Syntax 77
 - For the More Curious: Autorelease Pool and ARC History 78
- 4. Delegation and Core Location 81
 - Projects, Targets, and Frameworks 82
 - Core Location 83
 - Receiving updates from CLLocationManager 85
 - Delegation 87
 - Protocols 88
 - Delegation, controllers, and memory management 90
 - Using the Debugger 91
 - Using breakpoints 91
 - Diagnosing crashes and exceptions 95
 - Bronze Challenge: Distance Filter 97
 - Silver Challenge: Heading 97
 - For the More Curious: Build Phases, Compiler Errors, and Linker Errors 97
 - Preprocessing 98
 - Compiling 99
 - Linking 100
- 5. MapKit and Text Input 103
 - Object Diagrams 103
 - MapKit Framework 104
 - Interface Properties 105
 - Being a MapView Delegate 108
 - Using the documentation 110
 - Your own MKAnnotation 114
 - Tagging locations 118
 - Putting the pieces together 119

Bronze Challenge: Map Type	120
Silver Challenge: Changing the Map Type	120
Gold Challenge: Annotation Extras	120
6. Subclassing UIView and UIScrollView	121
Views and the View Hierarchy	123
Creating a Custom View	124
The drawRect: Method	128
Core Graphics	131
UIKit Drawing Additions	132
Redrawing Views	134
Motion Events	135
Using UIScrollView	137
Panning and paging	139
Zooming	139
Hiding the Status Bar	141
Bronze Challenge: Colors	142
Silver Challenge: Shapes	142
Gold Challenge: Another View and Curves	142
7. View Controllers	145
UIViewController	145
Creating HypnoTime	145
Subclassing UIViewController	146
Another UIViewController	150
UITabBarController	156
View Controller Lifecycle	161
Initializing view controllers	161
UIViewController and lazy loading	162
View Controller Subclasses and Templates	169
Bronze Challenge: Another Tab	169
Silver Challenge: Controller Logic	169
For the More Curious: The main Function and UIApplication	169
For the More Curious: Retina Display	170
8. Notification and Rotation	173
Notification Center	173
UIDevice Notifications	174
Autorotation	176
Setting autoresizing masks programmatically and bitwise operations	182
Forcing Landscape Mode	184
Bronze Challenge: Proximity Notifications	185
Silver Challenge: Programmatically Setting Autoresizing Masks	185
Gold Challenge: Overriding Autorotation	185
For the More Curious: Overriding Autorotation	185
9. UITableView and UITableViewController	187
Beginning the Homeowner Application	187
UITableViewController	189
Subclassing UITableViewController	189
UITableView's Data Source	191
Creating BNRItemStore	192

- Implementing data source methods 196
- UITableViewCells 198
 - Creating and retrieving UITableViewCells 199
 - Reusing UITableViewCells 201
- Code Snippet Library 202
- Bronze Challenge: Sections 205
- Silver Challenge: Constant Rows 205
- Gold Challenge: Customizing the Table 205
- 10. Editing UITableView 207
 - Editing Mode 207
 - Adding Rows 213
 - Deleting Rows 214
 - Moving Rows 215
 - Bronze Challenge: Renaming the Delete Button 217
 - Silver Challenge: Preventing Reordering 217
 - Gold Challenge: Really Preventing Reordering 217
- 11. UINavigationController 219
 - UINavigationController 220
 - An Additional UIViewController 223
 - Navigating with UINavigationController 229
 - Pushing view controllers 229
 - Passing data between view controllers 231
 - Appearing and disappearing views 232
 - UINavigationController 233
 - Bronze Challenge: Displaying a Number Pad 238
 - Silver Challenge: Dismissing a Number Pad 238
 - Gold Challenge: Pushing More View Controllers 238
- 12. Camera 239
 - Displaying Images and UIImageView 239
 - Taking pictures and UIImagePickerController 241
 - Creating BNRImageStore 248
 - NSDictionary 249
 - Creating and using keys 251
 - Core Foundation and toll-free bridging 253
 - Wrapping up BNRImageStore 254
 - Dismissing the keyboard 256
 - Bronze Challenge: Editing an Image 257
 - Silver Challenge: Removing an Image 257
 - Gold Challenge: Camera Overlay 257
 - For the More Curious: Recording Video 257
- 13. UIPopoverController and Modal View Controllers 261
 - Universalizing Homepwner 262
 - Determining device family 263
 - UIPopoverController 263
 - More Modal View Controllers 266
 - Dismissing modal view controllers 269
 - Modal view controller styles 270
 - Completion blocks 272

Modal view controller transitions	273
Bronze Challenge: Universalizing Whereami	274
Silver Challenge: Peeling Away the Layers	274
Gold Challenge: Popover Appearance	274
For the More Curious: View Controller Relationships	274
Parent-child relationships	274
Presenting-presenter relationships	275
Inter-family relationships	276
14. Saving, Loading, and Application States	279
Archiving	279
Application Sandbox	281
Constructing a file path	282
NSKeyedArchiver and NSKeyedUnarchiver	283
Application States and Transitions	286
Writing to the Filesystem with NSData	288
More on Low-Memory Warnings	291
Model-View-Controller-Store Design Pattern	292
Bronze Challenge: PNG	292
Silver Challenge: Archiving Whereami	292
For The More Curious: Application State Transitions	292
For the More Curious: Reading and Writing to the Filesystem	294
For the More Curious: The Application Bundle	296
15. Subclassing UITableViewCell	299
Creating HomepwnerItemCell	299
Configuring a UITableViewCell subclass's interface	301
Exposing the properties of HomepwnerItemCell	302
Using HomepwnerItemCell	303
Image Manipulation	304
Relaying Actions from UITableViewCells	308
Adding pointers to cell subclass	309
Relaying the message to the controller	310
Objective-C selector magic	310
Presenting the image in a popover controller	312
Bronze Challenge: Color Coding	315
Silver Challenge: Cell Base Class	315
Gold Challenge: Zooming	315
16. Core Data	317
Object-Relational Mapping	317
Moving Homepwner to Core Data	317
The model file	318
NSManagedObject and subclasses	322
Updating BNRItemStore	325
Adding BNRAssetTypes to Homepwner	330
More About SQL	335
Faults	336
Trade-offs of Persistence Mechanisms	338
Bronze Challenge: Assets on the iPad	339
Silver Challenge: New Asset Types	339

- Gold Challenge: Showing Assets of a Type 339
- 17. Localization 341
 - Internationalization Using NSLocale 342
 - Localizing Resources 343
 - NSLocalizedString and Strings Tables 346
 - Bronze Challenge: Another Localization 348
 - For the More Curious: NSBundle’s Role in Internationalization 348
- 18. NSUserDefaults 351
 - Updating Whereami 351
 - Using NSUserDefaults 353
 - Silver Challenge: Initial Location 355
 - Gold Challenge: Concise Coordinates 355
 - For the More Curious: The Settings Application 355
- 19. Touch Events and UIResponder 357
 - Touch Events 358
 - Creating the TouchTracker Application 359
 - Drawing with TouchDrawView 360
 - Turning Touches Into Lines 362
 - The Responder Chain 364
 - Bronze Challenge: Saving and Loading 365
 - Silver Challenge: Colors 365
 - Gold Challenge: Circles 365
 - For the More Curious: UIControl 365
- 20. UIGestureRecognizer and UINavigationController 367
 - UIGestureRecognizer Subclasses 368
 - Detecting Taps with UITapGestureRecognizer 368
 - UINavigationController 371
 - UILongPressGestureRecognizer 373
 - UIPanGestureRecognizer and Simultaneous Recognizers 374
 - For the More Curious: UINavigationController and UIResponderStandardEditActions 376
 - For the More Curious: More on UIGestureRecognizer 377
 - Bronze Challenge: Clearing Lines 378
 - Silver Challenge: Mysterious Lines 378
 - Gold Challenge: Speed and Size 378
 - Mega-Gold Challenge: Colors 378
- 21. Instruments 379
 - Static Analyzer 379
 - Instruments 381
 - Allocations Instrument 381
 - Time Profiler Instrument 387
 - Leaks Instrument 390
 - Xcode Schemes 392
 - Creating a new scheme 393
 - Build Settings 395
- 22. Core Animation Layer 399
 - Layers and Views 399
 - Creating a CALayer 400
 - Layer Content 403

Implicitly Animatable Properties	405
Bronze Challenge: Another Layer	406
Silver Challenge: Corner Radius	407
Gold Challenge: Shadowing	407
For the More Curious: Programmatically Generating Content	407
For the More Curious: Layers, Bitmaps, and Contexts	408
23. Controlling Animation with CAAAnimation	411
Animation Objects	411
Spinning with CABasicAnimation	414
Timing functions	417
Animation completion	418
Bouncing with a CAKeyframeAnimation	418
Bronze Challenge: More Animation	420
Silver Challenge: Even More Animation	420
Gold Challenge: Chaining Animations	420
For the More Curious: The Presentation Layer and the Model Layer	420
24. UIStoryboard	423
Creating a Storyboard	423
UITableViewController in Storyboards	427
Segues	430
More on Storyboards	434
25. Web Services and UIWebView	437
Web Services	438
Starting the Nerdfeed application	439
NSURL, NSMutableURLRequest, and NSURLConnection	440
Formatting URLs and requests	441
Working with NSURLConnection	441
Collecting XML data	442
Parsing XML with NSXMLParser	445
Constructing the tree of model objects	446
A quick tip on logging	455
UIWebView	456
For the More Curious: NSXMLParser	458
For the More Curious: The Request Body	459
For the More Curious: Credentials	460
Bronze Challenge: More Data	461
Silver Challenge: More UIWebView	461
26. UISplitViewController and NSRegularExpression	463
Splitting Up Nerdfeed	464
Master-Detail Communication	468
Displaying the Master View Controller in Portrait Mode	474
Universalizing Nerdfeed	477
NSRegularExpression	478
Constructing a pattern string	480
Bronze Challenge: Finding the Subforum	482
Silver Challenge: Swapping the Master Button	483
Silver Challenge: Processing the Reply	483
Gold Challenge: Showing Threads	483

- 27. Blocks 485
 - Blocks and Block Syntax 485
 - Declaring block variables 486
 - Defining block literals 487
 - Executing blocks 488
 - More notes about blocks 488
 - Basics of Using Blocks 489
 - Variable Capturing 492
 - Typical Block Usage 495
 - For the More Curious: The __block Modifier, Abbreviated Syntax, and Memory 496
 - For the More Curious: Pros and Cons of Callback Options 499
- 28. Model-View-Controller-Store 503
 - The Need for Stores 503
 - Creating BNRFeedStore 506
 - Using the Store 508
 - Building BNRFeedStore 511
 - Initiating the connection 511
 - Another request 517
 - JSON Serialization 520
 - More on Store Objects 525
 - Bronze Challenge: UI for Song Count 526
 - Mega-Gold Challenge: Another Web Service 526
 - For the More Curious: JSON Data 526
- 29. Advanced MVCS 529
 - Caching the RSS Feed 529
 - Advanced Caching 535
 - NSCopying 541
 - Finishing the BNR feed 543
 - Read and Unread Items 545
 - Other Benefits of Store Objects 550
 - Bronze Challenge: Pruning the Cache 551
 - Silver Challenge: Favorites 551
 - Gold Challenge: JSON Caching 551
 - For the More Curious: Designing a Store Object 552
 - Determining external sources 552
 - Determining singleton status 552
 - Determining how to deliver results 553
 - For the More Curious: Automatic Caching and Cache.db 553
- 30. iCloud 555
 - iCloud Requirements 555
 - Ubiquity Containers 556
 - Provisioning a Ubiquity Container 558
 - Core Data and iCloud 561
 - For the More Curious: iCloud Backups 566
- 31. Afterword 569
 - What to do next 569
 - Shameless plugs 569
- Index 571

Introduction

An aspiring iOS developer faces 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 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 won't 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's Changed in the Third Edition?

This edition assumes that the reader is using Xcode 4.3 and running applications on an iOS 5 device or simulator.

With iOS 5, automatic reference counting (ARC) is the default memory management for iOS. We've redone the memory management chapter to address ARC, and we use ARC throughout the book.

You'll find new chapters on using gesture recognizers, storyboards, **NSRegularExpression**, and iCloud. We've also added two chapters dedicated to the the Model-View-Controller-Store design pattern, which we use at Big Nerd Ranch and believe is well-suited for many iOS applications.

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

Our Teaching Philosophy

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

Here's what we've learned over the years of teaching iOS programming:

- We've learned what ideas people must have to get started programming, and we focus on that subset.
- We've learned that people learn best when these concepts are introduced *as they are needed*.
- We've learned that programming knowledge and experience grow best when they grow together.
- We've learned that “going through the motions” is much more important than it sounds. Many times we'll ask you to start typing in code before you understand it. We get that you may feel like a trained monkey typing in a bunch of code that you don't 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's 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's happening. However, there will be times when you'll have to take our word for it. (If you think this will bug you, keep reading – we've got some ideas that might help.) Don't get discouraged if you run across a concept that you don't understand right away. Remember that we're 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 aren't clear at the beginning will suddenly make sense when you implement them the first (or the twelfth) time.

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

- Take a deep breath and wait it out. We'll get there, and so will you.
- Check the index. We'll 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'll want plenty of practice using it. Consult it early and often.
- If it's Objective-C or object-oriented programming concepts that 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 won’t 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’d 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/iOSProgramming3ed.zip>.

There are two types of learning. When you learn about the Civil 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 Civil 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 don’t remember what they have learned.

How This Book Is Organized

In this book, each chapter addresses one or more ideas of iOS development followed by hands-on practice. For more coding practice, we issue challenges towards the end of each chapter. 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’ll 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 won’t create an iOS application in these two chapters, you will build and debug a tool called `RandomPossessions` to ground you in these concepts.

In Chapters 4 and 5, you will learn about the Core Location and MapKit frameworks and create a mapping application called `Whereami`. You will also get plenty of experience with the important design pattern of delegation as well as working with protocols, frameworks, object diagrams, the debugger, and the Apple documentation.

Chapters 6 and 7 focus on the iOS user interface with the `Hypnosister` and `HypnoTime` applications. You will get lots of practice working with views and view controllers as well as implementing panning, zooming, and navigating between screens using a tab bar.

Introduction

In Chapter 8, you will create a smaller application named `HeavyRotation` while learning about notifications and how to implement autorotation in an application. You will also use autoresizing to make `HeavyRotation` iPad-friendly.

Chapter 9 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 nine chapters total to complete.

In Chapters 9, 10, and 15, 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 11 builds on the navigation experience gained in Chapter 7. You will learn how to use `UINavigationController`, and you will give `Homepwner` a drill-down interface and a navigation bar.

In Chapter 12, you’ll learn how to take pictures with the camera and how to display and store images in `Homepwner`. You’ll use `NSDictionary` and `UIImagePickerController`.

In Chapter 13, you’ll learn about `UIPopoverController` for the iPad and modal view controllers. In addition, you will make `Homepwner` a universal application – an application that runs natively on both the iPhone and the iPad.

Chapter 14 delves into ways to save and load data. In particular, you will archive data in the `Homepwner` application using the `NSCoding` protocol. The chapter also explains the transitions between application states, such as active, background, and suspended.

Chapter 16 is an introduction to Core Data. You will change the `Homepwner` application to store and load its data using an `NSManagedObjectContext`.

Chapter 17 introduces the concepts and techniques of internationalization and localization. You will learn about `NSLocale`, strings tables, and `NSBundle` as you localize parts of `Homepwner`. This chapter will complete the `Homepwner` application.

In Chapter 18, you will use `NSUserDefaults` to save user preferences in a persistent manner.

In Chapters 19 and 20, you’ll create a drawing application named `TouchTracker` to learn about touch events. You’ll see how to add multi-touch capability and how to use `UIGestureRecognizer` to respond to particular gestures. You’ll also get experience with the first responder and responder chain concepts and more practice with `NSDictionary`.

In Chapter 21, you’ll learn how to use Instruments to optimize the performance of your applications. This chapter also includes explanations of Xcode schemes and the static analyzer.

Chapters 22 and 23 introduce layers and the Core Animation framework with a brief return to the `HypnoTime` application to implement animations. You will learn about implicit animations and animation objects, like `CABasicAnimation` and `CAKeyframeAnimation`.

Chapter 24 covers a new feature of iOS for building applications called storyboards. You’ll piece together an application using `UIStoryboard` and learn more about the pros and cons of using storyboards to construct your applications.

Chapter 25 ventures into the wide world of 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 26, 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 27 will teach you about the how and why of blocks – an increasingly important feature of the iOS SDK. You'll create a simple application to prepare for using blocks in **Nerdfeed** in the next chapter.

In Chapters 28 and 29, you will change the architecture of the **Nerdfeed** application so that it uses the Model-View-Controller-Store design pattern. You'll learn about request logic and how to best design an application that communicates with external sources of data.

In Chapter 30, you'll learn how to enable an application to use iCloud to synchronize and back up data across a user's iOS devices.

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:

- There is an alternative syntax for calling accessor methods known as *dot-notation*. In this book, we will explain dot-notation, but we will not use it. For us and for most beginners, dot-notation tends to obfuscate what is really happening.
- In our subclasses of **UIViewController**, we always change the designated initializer to **init**. It is our opinion that the creator of the instance should not need to know the name of the XIB file that the view controller uses, or even if it has a XIB file at all.
- We will always create view controllers programmatically. Some programmers will instantiate view controllers inside XIB files. We've found this practice leads to projects that are difficult to comprehend and debug.
- We will nearly always start a project with the simplest template project: the empty application. The boilerplate code in the other template projects doesn't follow the rules that precede this one, so we think they make a poor basis upon which to build.

We believe that following these rules makes our code easier to understand and easier to maintain. After you have worked through this book (where you *will* do it our way), you should try breaking the rules to see if we're wrong.

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 will be formatted the same for simplicity's sake. For example, "In the **loadView** method of the **RexViewController** 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'll 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 will be 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 QuizAppDelegate : NSObject <UIApplicationDelegate> {
    int currentQuestionIndex;

    // The model objects
    NSMutableArray *questions;
    NSMutableArray *answers;

    // The view objects
    IBOutlet UILabel *questionField;
    IBOutlet UILabel *answerField;
    UIWindow *window;
}
```

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 can't 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's okay when you're starting with iOS. In these first chapters, you'll be focused on learning the fundamentals of the iOS SDK, and these are the same across iOS devices. Later in the book, we'll 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.

3

Managing Memory with ARC

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

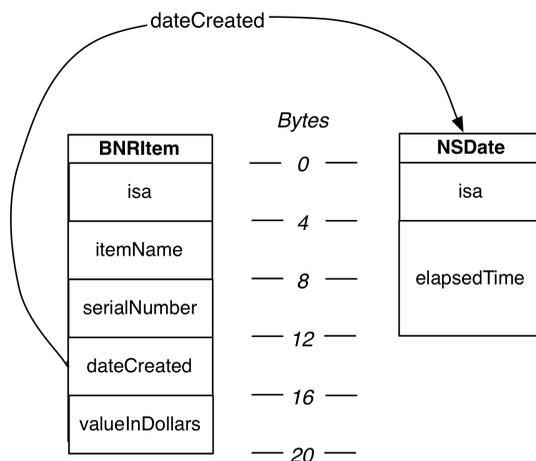
The Heap

All Objective-C objects are stored in a part of memory called *the heap*. When we send an `alloc` message to a class, a chunk of memory is allocated from the heap. This chunk includes space for the object's instance variables.

For example, consider an instance of `NSDate`, which represents a specific point in time. An `NSDate` has two instance variables: a `double` that stores the number of seconds since a fixed reference point in time and the `isa` pointer, which every object inherits from `NSObject`. A `double` is eight bytes, and a pointer is 4 bytes, so each time `alloc` is sent to the `NSDate` class, 12 bytes is allocated from the heap for a new `NSDate` object.

Consider another example: `BNRItem`. A `BNRItem` has five instance variables: four pointers (`isa`, `itemName`, `serialNumber`, and `dateCreated`) and an `int` (`valueInDollars`). The amount of memory needed for an `int` is four bytes, so the total size of a `BNRItem` is 20 bytes (Figure 3.1).

Figure 3.1 Byte count of `BNRItem` and `NSDate` instances



Notice in Figure 3.1 that the **NSDate** object does not live inside the **BNRItem**. Objects never live inside one another; they exist separately on the heap. Instead, objects keep references to other objects as needed. These references are the pointer instance variables of an object. Thus, when a **BNRItem**'s `dateCreated` instance variable is set, the *address* of the **NSDate** instance is stored in the **BNRItem**, not the **NSDate** itself. So, if the **NSDate** was 10, 20, or even 1000 bytes, it wouldn't affect the size of the **BNRItem**.)

The Stack

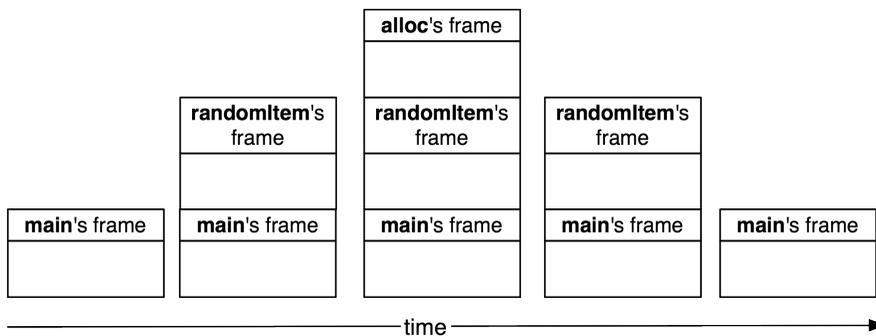
There is another part of memory called the *stack* that is separate from the heap. The reason for the names heap and stack has to do with how we visualize them. The heap is a giant heaping mess of objects, and we use pointers to remember where those objects are stored within the heap. The stack, on the other hand, can be visualized as a physical stack of *frames*.

When a method (or function) is executed, it allocates a chunk of memory from the stack. This chunk of memory is called a frame, and it 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 the **main** function, the frame for **main** is put at the bottom of the stack. When **main** calls another method (or function), the frame for that method is added to the top of the stack. Of course, that method could call another method, and so on, until we have a towering stack of frames. Then, as each method or function finishes, its frame is "popped off" the stack and destroyed. If the method is called again, a brand new frame will be allocated and put on the stack.

For example, in your **RandomPossessions** application, the **main** function runs **BNRItem**'s **randomItem** method, which in turn runs **BNRItem**'s **alloc** method. The stack would look like Figure 3.2. Notice that **main**'s frame stays alive while the other methods are executing because it has not yet finished executing.

Figure 3.2 Stack growing and shrinking



Recall that the **randomItem** method runs inside of a loop in the **main** function. So with every iteration, the stack grows and shrinks as frames are pushed on and popped off the stack.

Pointer Variables and Object Ownership

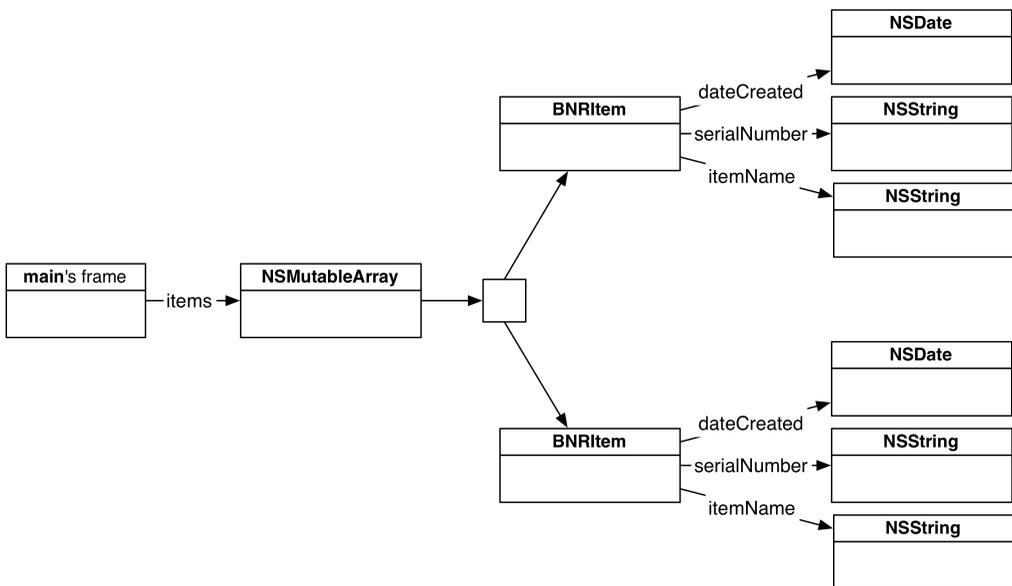
Pointer variables convey *ownership* of the objects that they point to.

- When a method (or function) has a local variable that points to an object, that method 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 RandomPossessions application as a whole. Or, better yet, reopen RandomPossessions.xcodeproj and have another look at the code in main.m. In this application, an instance of **NSMutableArray** is created in the **main** function, and then 10 **BNRItem** instances are added to the array.

Figure 3.3 shows the objects in RandomPossessions and the pointers that reference them.

Figure 3.3 Objects and pointers in RandomPossessions



The **NSMutableArray** is pointed to by the local variable **items** within the **main** function, so the **main** function owns the **NSMutableArray**. Each **BNRItem** instance owns the objects pointed to by its instance variables.

In addition, the **NSMutableArray** owns the **BNRItems**. Recall that a collection object, like an **NSMutableArray**, holds pointers to objects instead of actually containing them. These pointers convey ownership: an array owns the objects it points to.

The relationship between pointers and object ownership is important for understanding memory management in iOS.

Memory Management

If heap memory were infinite, we could create all the objects we needed 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. So it is important to destroy objects that are no longer needed to free up and reuse heap memory. On the other hand, it is critical *not* to destroy objects that *are* still needed.

The idea of object ownership helps us determine whether an object should be destroyed.

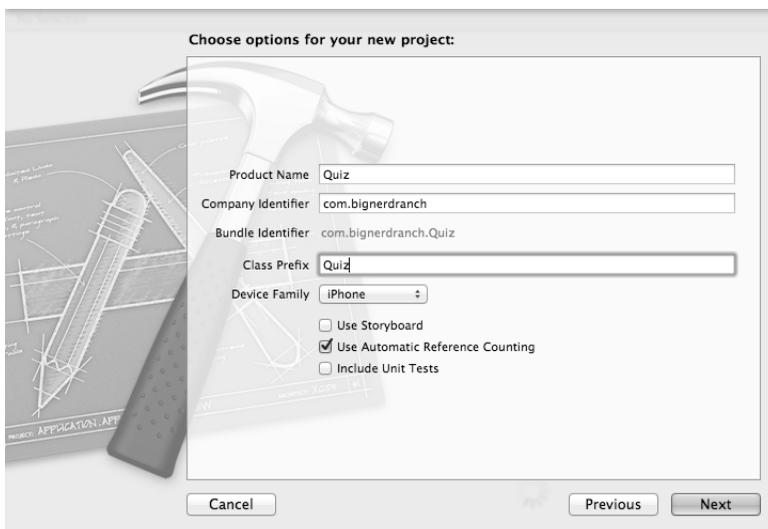
- *An object with no owners should be destroyed.* An ownerless object cannot be sent messages and is isolated and useless to the application. Keeping it around wastes precious memory. This is called a *memory leak*.
- *An object with at least one owner must not be destroyed.* If an object is destroyed but another object or method still has a pointer to it (or, more accurately, a pointer to where it used to live), then you have a very dangerous situation: sending a message to an object that no longer exists will crash your application. This is called *premature deallocation*.

Using ARC for memory management

The good news is that you don't need to keep track of who owns whom and what pointers still exist. Instead, your application's memory management is handled for you by *automatic reference counting*, or ARC.

In both projects you've built in Xcode so far, you've made sure to Use Automatic Reference Counting when creating the project (Figure 3.4). This won't change; all of your projects in this book will use ARC for managing your application's memory.

Figure 3.4 Naming a new project



(If the Use Automatic Reference Counting box in Figure 3.4 was unchecked, the application would use *manual reference counting* instead, which was the only type of memory management available before

iOS 5. For more information about manual reference counting and **retain** and **release** messages, see the For the More Curious section at the end of this chapter.)

ARC can be relied on to manage your application’s memory automatically for the most part, but it’s important to understand the concepts behind it to know how to step in when you need to. So let’s return to the idea of object ownership.

How objects lose owners

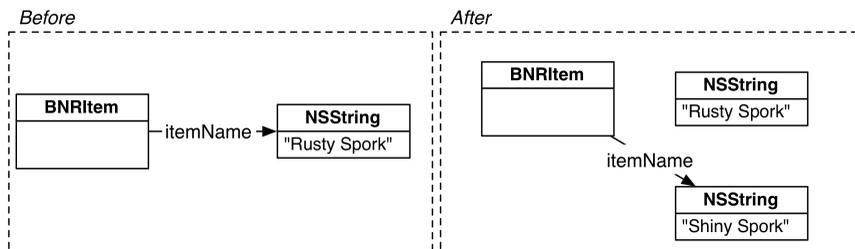
We know that an object is safe to destroy – and should be destroyed – when it no longer has any owners. So how does an object lose an owner?

- 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`.
- A variable that points to the object is itself destroyed.

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

Why might a pointer change the object it points to? Imagine a **BNRItem**. The **NSString** that its `itemName` instance variable points to reads “Rusty Spork.” If we polished the rust off of that spork, it would become a shiny spork, and we’d want to change the `itemName` to point at a different **NSString** (Figure 3.5).

Figure 3.5 Changing a pointer



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.

Why would you set a pointer to `nil`? Remember that setting a pointer to `nil` represents the absence of an object. For example, say you have a **BNRItem** that represents a television. Then, someone scratches off the television’s serial number. You would then set its `serialNumber` instance variable to `nil`. The **NSString** that `serialNumber` used to point to loses an owner.

When a pointer variable itself is destroyed, the object that the variable was pointing at loses an owner. At what point a pointer variable will get destroyed depends on whether it is a local variable or an instance variable.

Recall that instance variables live in the heap as part of an object. When an object gets destroyed, its instance variables are also destroyed, and any object that was pointed to by one of those instance variables loses an owner.

Local variables live in the method’s frame. When a method finishes executing and its frame is popped off the stack, any object that was pointed to by one of these local variables loses an owner.

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

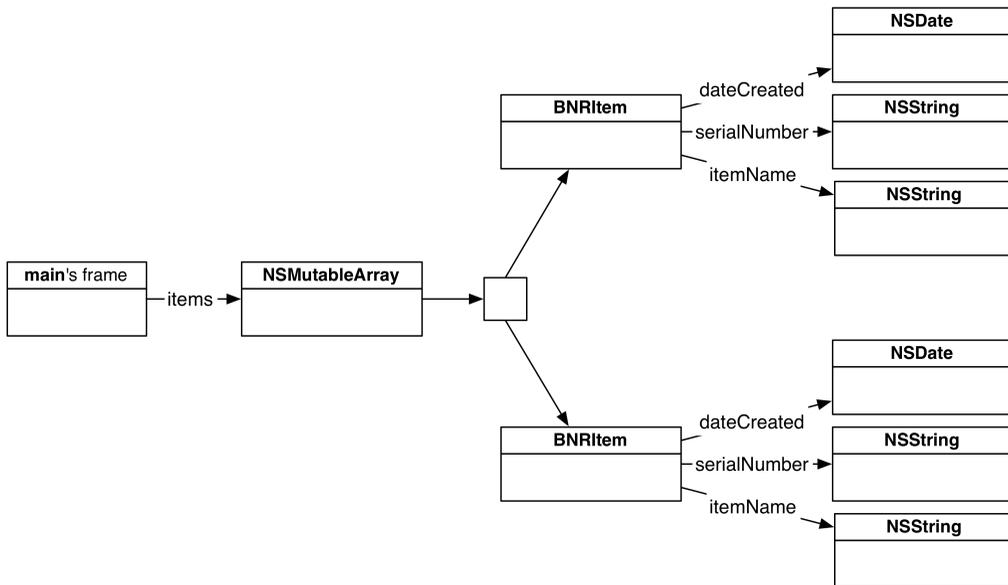
```
[items removeObject:p]; // object pointed to by p loses an owner
```

Keep in mind that losing an owner doesn’t necessarily mean that the object gets destroyed; if there is still another pointer to the object somewhere, then that object will continue to exist. However, when an object loses its last owner, it means certain and appropriate death.

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.

We have an example of this in **RandomPossessions**. Take another look at the object diagram of this application.

Figure 3.6 Objects and pointers in **RandomPossessions**



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

But it doesn’t stop there. When the **NSMutableArray** is destroyed, all of its pointers to **BNRItems** are destroyed. Once these variables are gone, no one owns any of the **BNRItems**, so they are all destroyed.

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

Let's add some code so that we can see this destruction as it happens. **NSObject** implements a **dealloc** method, which is sent to an object when it is about to be destroyed. We can override this method in **BNRItem** to print something to the console when a **BNRItem** is destroyed. In `RandomPossessions.xcodeproj`, 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 **BNRItems** 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 simply by setting `items` to `nil`. That's the power of ARC.

Strong and Weak References

So far, we've said that anytime a pointer variable stores the address of an object, that object has an owner and will stay alive. This is known as a *strong reference*. However, a variable can optionally *not* take ownership of an object 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 an unusual situation called a *retain cycle*. A retain cycle occurs when two or more objects have strong references to each other. This is bad news. When two objects own each other, they will 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 by virtue of those two strong references.

Thus, a retain 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 retain cycle in `RandomPossessions` to see how this works. First, we'll give **BNRItem** instances the ability to hold another **BNRItem** (so we can represent things like backpacks and purses). In addition, a **BNRItem** will know which **BNRItem** holds it. In `BNRItem.h`, add two instance variables and accessors

```
@interface BNRItem : NSObject
{
    NSString *itemName;
    NSString *serialNumber;
    int valueInDollars;
    NSDate *dateCreated;
```

```
    BNRItem *containedItem;
    BNRItem *container;
}

+ (id)randomItem;

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

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

- (void)setContainer:(BNRItem *)i;
- (BNRItem *)container;
```

Implement the accessors 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
    [i setContainer:self];
}

- (BNRItem *)containedItem
{
    return containedItem;
}

- (void)setContainer:(BNRItem *)i
{
    container = i;
}

- (BNRItem *)container
{
    return container;
}
```

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

```
#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 *p = [BNRItem randomItem];
            [items addObject:p];
        }

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

```

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

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

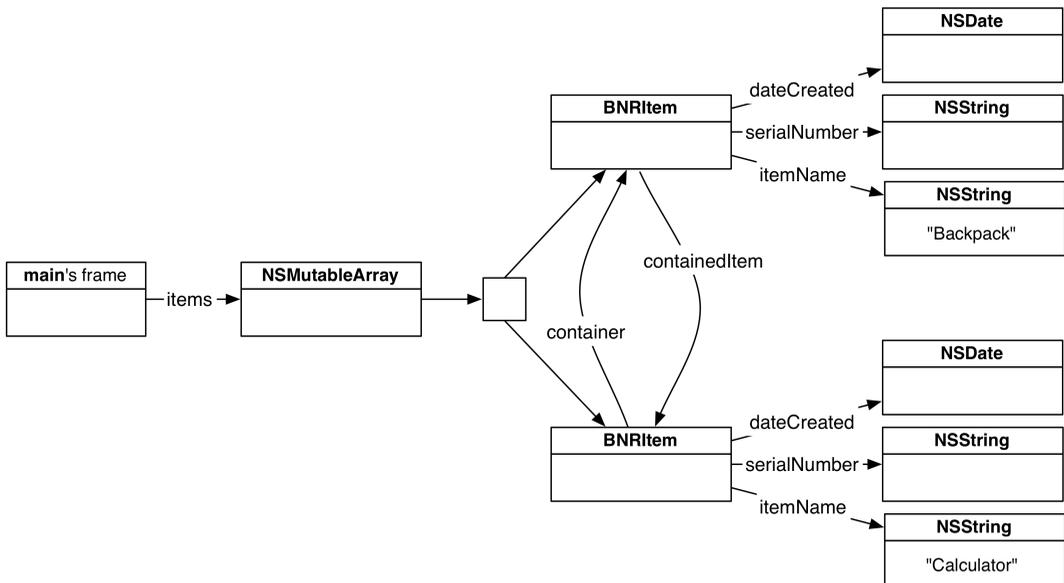
[backpack setContainedItem:calculator];

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

```

Here's what the application looks like now:

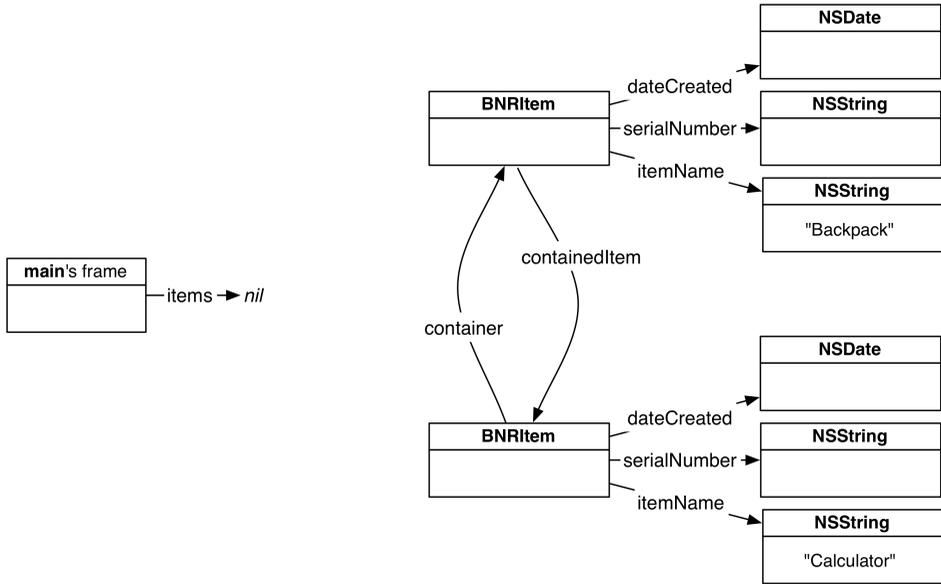
Figure 3.7 RandomPossessions with retain cycle



Per our understanding of memory management so far, both **BNRItems** should be destroyed along with their instance variables when `items` is set to `nil`. Build and run the application. Notice that the console does not report that these objects have been destroyed.

This is a retain cycle: the backpack and the calculator have strong references to one another, so there is no way to destroy these objects. Figure 3.8 shows the objects in the application that are still taking up memory once `items` has been set to `nil`.

Figure 3.8 A retain cycle!



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

To fix this problem, one of the pointers between the **BNRItems** 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 retain cycle, the backpack is the parent, and the calculator is the child. Thus, the backpack can keep its strong reference to the calculator (*containedItem*), but the calculator's reference to the backpack (*container*) should be weak.

To declare a variable as a weak reference, we 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.

Every retain cycle 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 retain cycle.

A child holding a strong reference to its *parent's* parent also causes a retain 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.

It's good to understand and look out for retain cycles, but keep in mind that they are quite rare. Also, Xcode has a Leaks tool to help you find them. We'll see how to use this tool in Chapter 21.

An interesting property of weak references is that they know when the object they reference is destroyed. Thus, if the backpack is destroyed, the calculator automatically sets its container instance variable to `nil`. In `main.m`, make the following changes to see this happen.

```
NSMutableArray *items = [[NSMutableArray alloc] init];

BNRItem *backpack = [[BNRItem alloc] init];
[backpack setItemName:@"Backpack"];
{items addObject:backpack};

BNRItem *calculator = [[BNRItem alloc] init];
[calculator setItemName:@"Calculator"];
{items addObject:calculator};

[backpack setContainedItem:calculator];

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

backpack = nil;

NSLog(@"Container: %@", [calculator container]);

calculator = nil;
```

Build and run the application. Notice that after the backpack is destroyed, the calculator reports that it has no container without any additional work on our part.

A variable can also be declared using the `__unsafe_unretained` attribute. Like a weak reference, an unsafe unretained reference does not take ownership of the object it points to. Unlike a weak reference, an unsafe unretained reference is not automatically set to `nil` when the object it points to is destroyed. This makes unsafe unretained variables, well, unsafe. To see an example, change `container` to be unsafe unretained in `BNRItem.h`.

```
__strong __unsafe_unretained BNRItem *container;
```

Build and run the application. It will most likely crash. The reason? When the calculator was asked for its container within the `NSLog` function call, it obligingly returned its value – the address in memory where the non-existent backpack used to live. Sending a message to a non-existent object resulted in a crash. Oops.

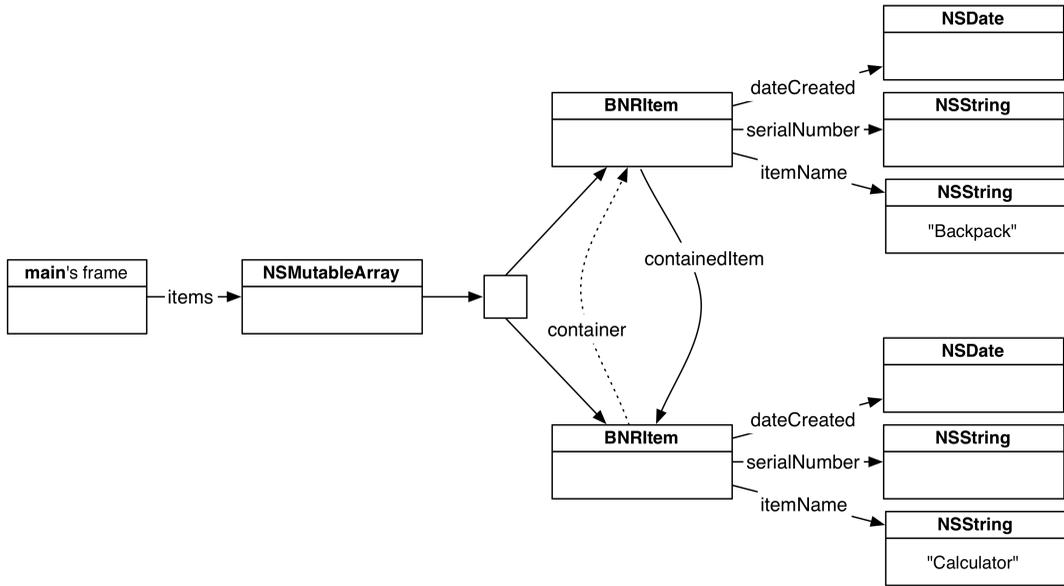
As a novice iOS programmer, you won't use `__unsafe_unretained`. As an experienced programmer, you probably won't use it, either. It exists primarily for backwards compatibility: applications prior to iOS 5 could not use weak references, so to have similar behavior, they must use `__unsafe_unretained`.

Be safe. Change this variable back to `__weak`.

```
__unsafe_unretained __weak BNRItem *container;
```

Here's the current diagram of `RandomPossessions`. Notice that the arrow representing the container pointer variable is now a dotted line. A dotted line denotes a weak (or unsafe unretained reference). Strong references are always solid lines.

Figure 3.9 RandomPossessions with retain cycle avoided



Properties

Each time we’ve added an instance variable to **BNRItem**, we’ve declared and implemented a pair of accessor methods. Now we’re going to see how to use *properties* instead. Properties are a convenient alternative to writing out accessors for instance variables – one that saves a lot of typing and makes your class files much clearer to read.

Declaring properties

A property is declared in the interface of a class where methods are declared. A property declaration has the following form:

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

When you declare a property, you are implicitly declaring a setter and a getter for the instance variable of the same name. So the above line of code is equivalent to the following:

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

Each property has a set of attributes that describe the behavior of the accessor methods. The attributes are declared in parentheses after the @property directive. Here is an example:

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

There are three property attributes. Each attribute has two or three options, one of which is the default and does not have to explicitly declared.

The first attribute of a property has two options: `nonatomic` or `atomic`. This attribute has to do with multi-threaded applications and is outside the scope of this book. Most Objective-C programmers typically use `nonatomic`: we do at Big Nerd Ranch, and so does Apple. In this book, we'll use `nonatomic` for all properties.

Let's change `BNRItem` to use properties instead of accessor methods. In `BNRItem.h`, replace all of your accessor methods with properties that are `nonatomic`.

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

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

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

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

--(NSDate *)dateCreated;

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

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

@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
```

Unfortunately, `nonatomic` is not the default option, so you will always need to explicitly declare your properties to be `nonatomic`.

The second attribute of a property is either `readwrite` or `readonly`. A `readwrite` property declares both a setter and getter, and a `readonly` property just declares a getter. The default option for this attribute is `readwrite`. This is what we want for all of `BNRItem`'s properties with the exception of `dateCreated`, which should be `readonly`. In `BNRItem.h`, declare `dateCreated` as a `readonly` property so that no setter method is declared for this instance variable.

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

The final attribute of a property describes its memory management. The default option depends on the type of the property. A property whose type is not a pointer to an object, like `int`, does not need memory management and thus defaults to `assign`. `BNRItem` only has one property that is not a pointer to an object, `valueInDollars`. For pointers to objects, like `NSString *`, this attribute defaults to `strong`. `BNRItem` has five object pointer properties: four of these will use `strong`, and the `container` property will use `weak` to avoid a retain cycle. With pointers to objects, it is good to be explicit and use the `strong` property to avoid confusion. In `BNRItem.h`, update the property declarations as shown.

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

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

Build and run the application. You should see the exact same behavior as the last time you ran it. The only difference is that `BNRIItem.h` is much cleaner.

Synthesizing properties

In addition to using a property to declare accessor methods, you can *synthesize* a property to generate the code for the accessor methods in the implementation file. Right now, `BNRIItem.m` defines the accessor methods declared by each property. For example, the property `itemName` declares two accessor methods, `itemName` and `setItemName:`, and these are defined in `BNRIItem.m` like so:

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

- (NSString *)itemName
{
    return itemName;
}
```

When you synthesize a property, you don't have to type out the accessor definitions. You can synthesize a property by using the `@synthesize` directive in the implementation file. In `BNRIItem.m`, add a `synthesize` statement for `itemName` and delete the implementations of `setItemName:` and `itemName`.

```
@implementation BNRIItem
@synthesize itemName;

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

You can synthesize properties in the same `synthesize` statement or split them up into multiple statements. In `BNRIItem.m`, synthesize the rest of the instance variables and delete the rest of the accessor implementations.

```
@implementation
@synthesize itemName;
@synthesize containedItem, container, serialNumber, valueInDollars,
    dateCreated;

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

```

{
    return serialNumber;
}

-(void)setValueInDollars:(int)i
{
    valueInDollars = i;
}

-(int)valueInDollars
{
    return valueInDollars;
}

-(NSDate *)dateCreated
{
    return dateCreated;
}

-(void)setContainedItem:(BNRItem *)i
{
    containedItem = i;

    // When given an item to contain, the contained
    // item will be given a pointer to its container
    [i setContainer:self];
}

-(BNRItem *)containedItem
{
    return containedItem;
}

-(void)setContainer:(BNRItem *)i
{
    container = i;
}

-(BNRItem *)container
{
    return container;
}

```

Usually, synthesized accessors work fine, but sometimes you need an accessor method to do some additional work. This is the case for **setContainedItem:**. Here is our original implementation:

```

- (void)setContainedItem:(BNRItem *)i
{
    containedItem = i;
    [i setContainer:self];
}

```

The synthesized setter won't include the second line establishing the reciprocal relationship between the container and the containedItem. Its implementation just looks like this:

```

- (void)setContainedItem:(BNRItem *)i
{
    containedItem = i;
}

```

Because we need this setter to do additional work, we cannot rely on the synthesized method and must write the implementation ourselves. Fortunately, writing our own implementation does not conflict

with synthesizing the property. Any implementation we add will override the synthesized version. In `BNRItem.m`, add back the implementation of `setContainedItem:`.

```
- (void)setContainedItem:(BNRItem *)i
{
    containedItem = i;
    [i setContainer:self];
}
```

Build and run the application again. It should work the same as always, but your code is much cleaner.

Synthesizing a property that you declared in the header file is optional, but typical. The only reason not to synthesize a property is if both the getter and the setter methods have additional behavior you need to implement.

Instance variables and properties

With properties, we can go even one step further in code clarity. By default, a synthesized property will access the instance variable of the same name. For example, the `itemName` property accesses the `itemName` instance variable: the `itemName` method returns the value of the `itemName` instance variable, and the `setItemName:` method changes the `itemName` instance variable.

If there is no instance variable that matches the name of a synthesized property, one is automatically created. So declaring an instance variable *and* synthesizing a property is redundant. In `BNRItem.h`, remove all of the instance variables as well as the curly brackets.

```
@interface BNRItem : NSObject
{
    NSString *itemName;
    NSString *serialNumber;
    int valueInDollars;
    NSDate *dateCreated;

    BNRItem *containedItem;
    __weak BNRItem *container;
}
```

Build and run the application. Notice there are no errors and everything works fine. All of the instance variables (like `itemName` and `dateCreated`) still exist even though we no longer explicitly declare them.

Copying

There is one more change we need to make to our properties – specifically, the two properties that point to instances of `NSString`.

In general, when you have a property that points to an instance of a class that has a mutable subclass (like `NSString` or `NSArray`), it is safer to make a copy of the object to point to rather than pointing to an existing object that could have other owners.

For instance, imagine if a `BNRItem` was initialized so that its `itemName` pointed to an instance of `NSMutableString`.

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

This code is valid because an **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 **BNRItem** that also points to it.

You may be wondering why this is a real problem. In your application, you're not going to change this string unless you mean to. However, when you write classes for others to use, you can't be sure how they will use your classes, and you have to program defensively.

In this case, the defense is to declare this property using the memory management attribute `copy` instead of `strong`. In `BNRItem.h`, change the `itemName` and `serialNumber` properties to `copy`.

```
@property (nonatomic, copy) NSString *itemName;
@property (nonatomic, copy) NSString *serialNumber;
```

Now the generated setter method for the synthesized `itemName` property looks like this:

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

Instead of setting `itemName` to point to the **NSString** object pointed to by `str`, this setter sends the message `copy` to that **NSString**. The `copy` method returns a new **NSString** object (not an **NSMutableString**) that has the same values as the original string, and `itemName` is set to point at the new string. 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 doesn't gain or lose an owner, and none of its data changes.

Dot Syntax

We should mention an alternative syntax for sending accessor messages to an object called dot syntax:

```
// Following two lines are exactly equivalent
int value = [item valueInDollars];
int value = item.valueInDollars;

// Following two lines are exactly equivalent
[item setValueInDollars:5];
item.valueInDollars = 5;
```

We have reservations about Objective-C newcomers using dot syntax. We think it hides the fact that you are actually sending a message and can be confusing. Once you are comfortable with Objective-C, it is totally okay to use dot syntax. But while you are learning, it's better to use square brackets to make sure you understand what is really going on.

This book will always use square brackets for sending accessor messages.

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 was a guaranteed 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'll see more about the static analyzer in Chapter 21, 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 memory management problems became a thing of the past.

(Some people have an irrational fear of letting the compiler do their work for them and say they prefer manual memory management for this reason. If someone says something like that to you, open up one of their .m files, go to the Product menu, and select Generate Assembly File from the Generate Output menu item. Tell them if they don't trust the compiler, then they should be writing the assembly code they see in front of them.)

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 after using it within the method in which it was returned.

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

```
@autoreleasepool {  
    // Get a BNRItem back from a method that created it, method doesn't say alloc/copy  
    BNRItem *item = [BNRItem someItem];  
} // item loses an owner and is destroyed because nothing else took ownership of it
```

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

Index

Symbols

#import, 55
%@ prefix, 37
.h files, 41
.m files, 41, 44
@ prefix
 creating strings with, 36
 and Objective-C keywords, 42
@autoreleasepool, 79
@class, 195
@end, 42
@implementation, 44
@interface, 42
@optional, 89
@private, 543
@property, 72
@protected, 543
@protocol, 88
@public, 543
@selector(), 235
@synthesize, 74
^, 486
_cmd, 292
__block, 497, 498
__bridge, 253, 254
__unsafe_unretained, 71
__weak, 70

A

accessor methods, 43-46, 72
 (see also properties)
accessory indicator (**UITableViewCell**), 198
action methods, 15-17
 connecting in XIB file, 242-244
 and **UIControl**, 365-366
active state, 287
actor objects, 512, 513
addAnimation:forKey:, 417
addObject:, 35, 38
addSubview:, 125, 128
alloc, 30-31, 193
Allocations instrument, 381-387
allocWithZone:, 193
analyzing (code), 379-381

angled brackets, 120
animation transactions, 406
animationDidStop:finished:, 418
animations, 411
 (see also **CALayer**, layers)
 CABasicAnimation, 412-413, 415-417, 420
 CAKeyframeAnimation, 413, 418-420
 choosing, 415
 classes of, 399, 411-414
 and data types, 413
 identity matrices in, 420
 implicit, 405-406
 key paths of, 411, 415
 keyframes in, 414-416
 keys for, 417
 reusing, 417
 timing functions of, 417-418
animationWithKeyPath:, 417
anonymous functions (see blocks)
anti-aliasing, 170
API Reference, 110-114
APIs, 399
 (see also frameworks)
 Core Animation, 399, 403, 412
 Core Foundation, 252, 253-254
App ID, 23
Apple documentation, 110-114
application bundle, 282, 296-298, 348
application delegates, 11, 170
application dock, 287
application domain, 354
application sandbox, 281-283, 296
application states, 286-288, 292-293
application:didFinishLaunchingWithOptions:, 123, 170
applicationDidBecomeActive:, 293
applicationDidEnterBackground:, 284, 288, 293
applications, 20
 (see also debugging, universal applications)
 adding frameworks to, 82
 allowing orientations, 184
 build settings for, 395-397
 building, 20, 45, 82, 97-101, 346
 cleaning, 346
 data storage, 281-282, 338
 deploying, 22-23
 directories in, 281-283

- entitlements of, 556
- icons for, 24-25
- launch images for, 25
- optimizing CPU usage, 387-390
- profiling, 381-383
- running on iPad, 3
- running on simulator, 20
- templates for, 82
- testing offline, 551
- user preferences in, 351-356
- applicationWillEnterForeground:**, 293
- applicationWillResignActive:**, 293
- ARC (Automatic Reference Counting), 64
 - (see also memory management)
 - and blocks, 498
 - and `__bridge`, 253, 254
 - and Core Foundation objects, 253, 254
 - history of, 78
 - vs. manual reference counting, 78
 - overview, 64
 - and retain cycles, 67
- archiveRootObject:toFile:**, 283
- archiving
 - vs. Core Data, 317, 317
 - described, 279
 - implementing, 279-281
 - with **NSKeyedArchiver**, 283-286
 - thumbnail images, 307
 - when to use, 338
 - and XIB files, 281
- arguments, 31-32
- arrays
 - copying, 542
 - vs. dictionaries, 249
 - fast enumeration of, 57
 - and memory management, 66
 - and object ownership, 63, 66
 - overview, 37-38
 - writing to filesystem, 295
- assistant editor, 224-229, 242-244
- atomic, 73
- attributes (Core Data), 318, 320-320, 324
- attributes inspector, 8, 108-108
- auto-completion (in Xcode), 18, 202-204, 511
- automatic reference counting (see ARC)
- autorelease**, 78
- autorelease pool, 78
- autoresize masks, 179-180, 182-184

- autorotation, 176-186, 466
- availableMediaTypesForSourceType:**, 258
- awakeFromFetch**, 324
- awakeFromInsert**, 325

B

- background state, 287-288, 287, 292-293
- `backgroundColor`, 130
- becomeFirstResponder**, 135
- binary numbers, 183, 184
- bitmap contexts, 408, 408
- bitwise operators, 183, 184
- `__block`, 497
- blocks
 - as approach to callbacks, 495, 500
 - and code-completion, 511
 - capturing variables in, 492-495, 497
 - completion, 272-273, 495
 - copying, 498
 - creating, 487, 489
 - location in memory, 498
 - nesting, 532
 - in operation queue, 494, 534
 - as properties, 492
 - reasons to use, 492, 494, 495, 500
 - scope of, 492, 495
 - syntax, 485-489
 - timing of execution, 533
 - variables for, 486-489, 511
- BNRConnection**, 512-517
- bounds, 129, 131
- brackets, 31, 42, 120
- breakpoint navigator, 94
- breakpoints, 91-91, 94
- `__bridge`, 253, 254
- build configurations, 396
- build phases, 97-101
- build settings, 395-397
- bundles
 - application, 282, 296-298, 348
 - identifier, 23
 - NSBundle**, 348

C

- CAAnimation**, 399, 411
 - (see also animations)
- CAAnimationGroup**, 414

CABasicAnimation, 412-413, 415-417
Cache.db, 553
caching data, 529-545, 553
CAKeyframeAnimation, 413, 418-420
CALayer, 411
 (see also animations, layers)
 addAnimation:forKey:, 417
 animatable properties of, 405-406
 bitmap context for, 408
 contents, 403
 creating, 400-402
 delegate, 407
 described, 399
 designated initializer, 401
 drawInContext:, 407-408
 presentationLayer, 420
 properties, 402-406, 411-412, 415
 setPosition:, 405
 subclassing, 407
 superlayer, 404
 zPosition, 404-405
callbacks, 87, 87, 499, 499, 500
CAMediaTimingFunction, 417
camera, 259
 (see also images)
 recording video, 257-259
 taking pictures, 241-248
cancelsTouchesInView, 376
canPerformAction:withSender:, 376
canvas area, 6
CAPropertyAnimation, 411
capture groups (regular expressions), 481-482
CATransaction, 406
CATransform3DIdentity, 420
CATransition, 414
cells (see **UITableViewCell**)
CFRelease, 253
CFStringRef, 253
CFUUIDRef, 251-254
CGBitmapContextCreate, 409
CGContext, 129
CGContextRef, 129-134, 131, 132
 drawing to, 306
 and layers, 408-409
CGImage, 403
CGPoint, 126, 412-413
CGRect, 126
CGSize, 126, 306
class methods, 53-54
classes, 29
 (see also *individual class names*)
 copying files, 191
 creating, 39
 declaring, 41-42
 inheritance of, 38, 42
 overview, 29-30
 prefixes for, 59
 reusing, 191
 subclassing, 38-56
 superclasses, 38, 42, 50
CLLocation, 85-86
CLLocationManager, 84-87
CLLocationManagerDelegate, 88
closures (see blocks)
_cmd, 292
Cocoa Touch, 82
code snippet library, 202-204
code-completion (in Xcode), 18, 202-204, 511
compile-time errors, 99-99
compiling, 98, 99
completion blocks, 272-273, 495
connection:didFailWithError:, 443
connection:didReceiveData:, 443-443
connectionDidFinishLoading:, 443-445
connections inspector, 17
console, 21, 36, 37
const, 563
containsObject:, 537
contentMode (**UIImageView**), 240-241
contentView (**UITableViewCell**), 199-199, 299-300
contentViewController
(**UIPopoverController**), 264
contexts, drawing, 128-134
controller objects, 103-104
 (see also view controllers)
 defined, 10
 as delegates, 90
 lifespan of, 90
 and store objects, 292
convenience methods, 53-54
coordinate (MKAnnotation), 113, 116, 116
copy, 77, 498, 541
copying arrays, 542
copying blocks, 498
copying files, 191

copying objects, 76-77, 541
copyWithZone:, 542
 Core Animation, 399, 399, 403, 412
 (see also animations, **CALayer**)
 Core Data
 vs. archiving, 317, 317
 attributes, 318, 320-320
 entities, 318-322, 330-335
 faults, 336-337
 fetch requests, 327-329, 338
 fetched property, 338
 and iCloud, 561-566
 lazy fetching, 336
 logging SQL commands, 335
 model file, 318-322, 325
NSManagedObjectContext, 325-329
NSManagedObjectModel, 325-327
NSPersistentStoreCoordinator, 325-327
 as ORM, 317-318
 relationships, 320-322, 336-337
 and SQLite, 317, 325-327, 335
 subclassing **NSManagedObject**, 322-325
 transient attributes, 324
 versioning, 338, 338
 when to use, 317, 338, 529
 Core Foundation, 252, 253-254
 Core Graphics (framework), 128, 131, 306-306,
 403-403
 Core Location (framework), 83-87
count (NSArray), 38
 curly brackets, 42
currentDevice, 263
currentLocale, 342

D

data source methods, 196
 data storage, 281
 (see also archiving, Core Data)
 for application data, 281-282
 binary, 289, 294
 caching, 529-545
 choosing, 338
 with I/O functions, 294
 for images, 305-308
 with **NSData**, 288, 305-305
 dataSource (**UITableView**), 189, 191-198, 196
dealloc, 67, 90

debug area, 21
 debug navigator, 91
 debugger bar, 93
 debugging, 91
 (see also debugging tools, exceptions)
 categorizing log statements, 455
 compile-time errors, 99-99
 creating schemes for, 393-395
 exceptions, 56-57
 linker errors, 100-101
NSError, 294-295
 stack trace, 91, 95
 stepping through methods, 93-94
 testing offline behavior, 551
 debugging tools
 Allocations instrument, 381-387
 breakpoints, 91-91, 94
 debug navigator, 91
 debugger, 91-97
 heapshots, 387
 Instruments, 381-391
 issue navigator, 21
 log navigator, 36
 stack trace, 91, 92, 95
 static analyzer, 379-381
 Time Profiler, 387-390
 variables view, 92, 93
 declarations
 class, 41-42
 instance variable, 42
 method, 44, 48-48, 53-54
 protocol, 89
definesPresentationContext, 278
 delegate (property), 85, 85, 90, 90, 474
 (see also delegation)
 delegation
 as approach to callbacks, 499, 499, 500
 and controller objects, 90
 creating a delegate protocol, 469-474
 delegate, 85, 90, 474
 and layers, 407
 and memory management, 90
 overview, 87-88
 protocols used for, 88-90
 swapping delegates, 447
 vs. target-action pairs, 87
deleteRowsAtIndexPaths:withRowAnimation:,
 215

description (NSObject), 37, 39, 46
designated initializers, 48-52
detail view controllers, 464
developer certificates, 22
devices
 checking for camera, 245-246
 deploying to, 22, 23
 determining type of, 263
 orientation of, 174, 178
 provisioning, 22-23, 23
 Retina display, 24, 26, 170-171
dictionaries
 described, 249-251
 memory management of, 291
 using, 251-254
 writing to filesystem, 295
directories
 application, 281-283
 Documents, 282, 564, 567
 Library/Caches, 282, 567
 Library/Preferences, 282
 lproj, 343, 348
 temporary, 282
dismissPopoverAnimated:completion:, 266
dismissViewControllerAnimated:completion:,
271, 272
dock
 for applications, 287
 in XIB editor area, 5
documentation, using, 110-114
Documents directory, 282, 564, 567
domain, application, 354
domain, registration, 354
dot syntax, xvii, 77
dot-notation, xvii, 77
drawing contexts, 128-134
drawing operations, 131
drawInRect:withFont:, 133
drawLayer:inContext: (CALayer), 407-408
drawRect:, 128-131, 134
 and Core Graphics, 131, 408
 and run loop, 134
 and **UITableViewCell**, 299
drill-down interface
 with **UINavigationController**, 219
 with **UISplitViewController**, 463

E
editButtonItem, 237
editing (**UIView**,
UITableViewController), 207, 212
editor area, 5
encodeInt:forKey:, 280
encodeObject:forKey:, 280
encodeWithCoder:, 279, 279-280, 284
@end, 42
endEditing:, 233, 256
entities (Core Data), 318-322, 330-335
entitlements, 556
errors
 compile-time, 99-99
 connection, 443
 linker, 100-101
 and **NSError**, 294-295
 run-time, 56-57
event loop, 134
exceptions
 diagnosing in debugger, 95-97
 explained, 56-57
 internal inconsistency, 213
 throwing, 267
 unknown key, 229
 unrecognized selector, 56, 57, 311
 using **NSException**, 267
explicit layers, 400, 407
 (see also **CALayer**)
extern, 563

F
fast enumeration, 57
faults, 336-337
fetch requests, 327-329, 338
fetched property, 338
file inspector, 343
file paths, retrieving, 282-283
File's Owner, 151-156, 152
files
 header (.h), 41, 116
 implementation (.m), 41, 44, 116
 importing, 44, 83, 120
 including, 44
 intermediate, 98
 library, 100
 object, 99

filteredArrayUsingPredicate:, 329

first responder

and motion events, 135

and nil-targeted actions, 366

overview, 118-119

resigning, 233, 256

and responder chain, 364-365

and **UIMenuController**, 372

format string, 37

forward declarations, 195

frame (in stack), 62

frame (**UIView**), 126, 129, 131

frameworks

adding, 82-83, 399

class name prefixes of, 59

Cocoa Touch, 82, 82

Core Data (see Core Data)

Core Foundation, 252, 253-254

Core Graphics, 128, 131, 306, 403-403

Core Location, 83

header files, 83

importing, 83, 120

and linker errors, 100

MapKit, 103, 104

MobileCoreServices, 259

QuartzCore, 399, 403

UIKit, 403

functions

callback, 87, 87

vs. methods, 30

timing (animation), 417-418

G

genstrings, 347

gestures, 367

(see also **UIGestureRecognizer**,

UIScrollView)

long press, 373-374

panning, 137, 373, 374-376

pinching, 141

taps, 368-372

getter methods, 43-45

graphic contexts, 408

GUIDs, 251

H

.h files (see header files)

header files

description, 41

for frameworks, 83

importing, 44, 55-56

order of declarations in, 53

shortcut to implementation files, 116

header view (**UITableView**), 207-211

heap memory, 61, 62, 64

heapshots, 387

HeavyRotation application

implementing autorotation, 177-181

making universal, 181-182

registering and receiving notifications,
175-175

hierarchies

class, 38

layer, 399, 400

view, 124-126, 145, 148

Homepwner application

adding an image store, 248

adding drill-down interface, 219-238, 221

adding item images, 239-257

adding item store, 192-195

adding modal presentation, 266-272

customizing cells, 299-315

enabling editing, 207-217

localizing, 342-348

moving to Core Data, 317

object diagrams, 192, 221

reusing **BNRItem** class, 191

storing images, 288-291

universalizing, 262-263

HTTP protocol, 459-460

Hypnosister application

creating **HypnosisView**, 124-126

detecting shakes, 135-137

hiding status bar, 141

object diagram, 137

scrolling and zooming, 137-141

HypnoTime application

adding animation, 400-406, 414-420

adding second view controller, 150-156

adding tab bar controller, 156-161

creating, 145-150

I

I/O functions, 294

IBAction, 13, 15-16, 242-244
IBOutlet, 13-15, 226-229
ibtool, 344-346
iCloud
 backing up data with, 566
 controlling synchronization, 564, 567
 and Core Data, 561-566
 document storage, 555
 key-value storage, 555
 managing stored data, 565
 purpose, 555
 requirements, 555
 synchronizing with, 556
 and ubiquity containers, 556-561, 562, 564
 user accounts, 555, 555
icons
 application, 24-25
 camera, 241
 tab bar, 160-161
id, 48
identity matrices, 420
image picker (see **UIImagePickerController**)
imageName:, 171
imagePickerController:
didFinishPickingMediaWithInfo:, 245
imagePickerControllerDidCancel:, 245
images, 259
 (see also camera, **UIImage**, **UIImageView**)
 archiving, 305-305, 307
 caching, 288-291
 creating thumbnail, 304-308
 displaying in **UIImageView**, 239-241
 manipulating in offscreen contexts, 304-308
 for Retina display, 170
 storing, 248-251
 wrapping in **NSData**, 289
imageWithContentsOfFile:, 290
@implementation, 44
implementation files, 41, 44, 116
implicit animations, 405-406
implicit layers, 399, 403
#import, 55
importing files, 44, 55, 83, 120
inactive state, 287
including files, 44
incomplete implementation warnings, 96
Info.plist, 184
inheritance, single, 38, 42
init
 and
 alloc, 30-31
 overview, 47-52
 for view controllers, xvii
initialize, 354
initializers, 47-52
 disallowing calls to, 267
 and singletons, 192-195
initWithCGImage:, 403, 409
initWithCoder:, 279, 280-281
initWithContentsOfFile:, 294-295
initWithFrame:, 126, 401, 403
initWithNibName:bundle:, 161
insertObjectAtIndex:, 35, 38
inspectors
 attributes, 8, 108-108
 connections, 17
 file, 343
 overview, 7
 size, 179-180
instance variables, 72
 (see also properties)
 accessing with **->**, 543
 accessor methods for, 43
 and weak references, 71
 customizing in attributes inspector, 8
 declaring, 42-42
 description, 30
 in memory, 61
 memory management of, 66
 datatypes of, 42-43
 private, 543
 and properties, 76
 protected, 543
 public, 543
 visibility of, 543
instances, 30-31
instantiateWithOwner:options:, 304
Instruments, 381-391
 and schemes, 393-395
@interface, 42
interface files (see header files)
intermediate files, 98
internal inconsistency exception, 213
internationalization, 341-343, 348
 (see also localization)
iOS SDK documentation, 110-114

iPad, 3
(see also devices)
application icons for, 24
launch images for, 26
orientations, 178
running iPhone applications on, 3
XIB files for, 262

isa pointer, 48

isEqual:, 214, 537

isSourceTypeAvailable:, 245

issue navigator, 21, 99

J

JSON, 520-524, 526-527

K

kCAMediaTimingFunctionEaseInEaseOut, 418

key paths (animation), 411-415

key-value pairs, 250-251

in `Info.plist`, 184

keyboard

dismissing, 256

number pad, 238

setting properties of, 108

keyframes (animation), 414-416

keys

for animations, 417

in dictionaries, 250-255

keywords, 42

`KUTTypeImage`, 258, 259

`KUTTypeMovie`, 258, 259

L

labels (in message names), 32

landscape mode

allowing rotation to, 177

forcing rotation to, 184

and split view controllers, 466

language settings, 341, 346

launch images, 25-27

layers, 399

(see also **CALayer**)

compositing, 400

and delegation, 407

drawing programmatically, 407-409

explicit, 400, 407

hierarchy of, 400, 404-405

implicit, 399, 403

model vs. presentation, 420

size and position of, 402-403

and views, 399-400

Leaks instrument, 390-391

leaks, memory, 64

libraries

code snippet, 202-204

object, 7

system, 100, 100, 120

(see also frameworks)

library files, 100

Library/Caches directory, 282, 567

Library/Preferences directory, 282

linker errors, 100-101

LoadNibNamed:owner:options:, 304

loadView, 148, 150, 163, 211, 229

local variables, 62, 63, 66

`Localizable.strings`, 347

localization

adding, 343

and `ibtool`, 344-346

internationalization, 341-343, 348

`lproj` directories, 343, 348

NSBundle, 348

resources, 343-346

strings tables, 346-348

user settings for, 341, 346

XIB files, 343-346

localizedDescription, 294

location finding, 84-87

location services, 83, 83

(see also Core Location)

locationInView:, 370

locationManager:didFailWithError:, 86

locationManager:didUpdateToLocation:

fromLocation:, 85

log navigator, 36

low-memory warnings, 162, 248, 291

`lproj` directories, 343, 348

M

.m files, 41, 44

mach_msg_trap, 389

macros, preprocessor, 396-397

main, 34, 169

main bundle, 282, 296-298, 348

main operation queue, 494
mainBundle, 210, 297
makeKeyAndVisible, 123
 manual reference counting, 78
 map views, 109
 (see also **MKAnnotation**, **MKMapView**)
 changing type of, 351, 352
 zooming, 109-114
 MapKit (framework), 103, 104
 mapType (**MKMapView**), 351, 352
mapView:didUpdateUserLocation:, 112, 112
 masks, autoresize, 179-180, 182-184
 master view controllers, 464, 474
matchesInString:options:range:, 479
 mediaTypes, 257
 memory, 61, 62, 62, 64
 memory leaks, 64
 memory management
 with ARC, 64
 arrays, 66
 avoiding retain cycles, 70
 and controller objects, 90
 and copying, 77
 and delegates, 90
 dictionaries, 251, 291
 leaks, 64, 67
 (see also retain cycles)
 and Leaks instrument, 390-391
 need for, 64
 notifications, 174
 and object ownership, 65-67
 optimizing with Allocations instrument,
 381-387
 pointers, 65-67
 premature deallocation, 64
 properties, 73
 strong and weak references, 67-72
 UITableViewCell, 201
 memory warnings, 248, 291
 menus (**UIMenuController**), 371-372, 376
 messages, 31, 31-33
 (see also methods)
 metacharacters (regular expressions), 480
 methods, 30
 (see also *individual method names*)
 accessor, 43-46
 action, 15-17, 365-366
 arguments of, 48
 class, 53-54
 data source, 196
 declaring, 44, 47, 48-48, 53
 defined, 30
 designated initializer, 48-52
 implementing, 44-45, 47, 49
 initializer, 47-52
 vs. messages, 32
 names of, 32
 overriding, 46-47, 50-52
 parameters of, 48
 protocol, 89
 stepping through, 93-94
 minimumPressDuration, 373
MKAnnotation protocol, 114-117
MKAnnotationView, 104, 114
MKCoordinateRegion, 113, 113, 114
MKMapView, 104-105, 108-114, 112, 351
MKMapViewDelegate, 109-112
MKUserLocation, 113
 MobileCoreServices, 259
 .mobileprovision files, 23-23
 modal view controllers
 defined, 246
 dismissing, 269-270
 and non-disappearing parent view, 271
 relationships of, 275
 in storyboards, 431-434
 styles of, 270
 transitions for, 273
 modalPresentationStyle, 270, 278
 modalTransitionStyle, 273
 modalViewController, 269-270
 model file (Core Data), 318-322, 325
 model layer, 420
 model objects, 9, 103-104, 524
 Model-View-Controller (MVC), 9-11, 104-104,
 292
 Model-View-Controller-Store (MVCS)
 benefits of, 550-551
 vs. MVC, 292, 503-506
 when to use, 503
 motion events, 135-137
motionBegan:withEvent:, 136
motionCancelled:withEvent:, 136
motionEnded:withEvent:, 136
 multi-touch, 361
 MVC (Model-View-Controller), 9-11, 104, 292

MVCS (see Model-View-Controller-Store)

N

namespaces, 59

naming conventions

- accessor methods, 43
- cell reuse identifiers, 201
- class prefixes, 59
- delegate protocols, 89
- initializer methods, 47

navigation controllers (see

UINavigationController)

navigationController, 230, 275

navigationItem (**UIViewController**), 233

navigators

- breakpoint, 94
- debug, 91
- issue, 21, 99
- keyboard shortcuts for, 21
- log, 36
- project, 3-5

Nerdfeed application

- adding delegate protocol, 469-474
- adding iCloud support, 555-565
- adding **UIWebView**, 456-458
- caching RSS feeds, 529-545
- categorizing log statements, 455
- converting to MVCS, 506-525
- fetching data, 440-442
- indicating read items, 545-550
- parsing data, 444-455
- showing only post titles, 478-482
- using **UISplitViewController**, 464-466

nested message sends, 31

nextResponder, 364

NIB files, 5, 5, 304

(see also XIB files)

nibName:bundle:, 303

nil

- and arrays, 38
- as notification wildcard, 174
- returned from initializer, 50
- sending messages to, 32
- setting pointers to, 32
- targeted actions, 366
- as zero pointer, 32

nonatomic, 73

.nosync, 564

notifications

- as approach to callbacks, 499, 499, 500
- described, 173-176
- of low-memory warnings, 291
- posting, 563-564

NSArray, 37, 37-38, 537

(see also arrays)

NSBundle, 210, 348

NSCocoaErrorDomain, 294

NSCoder, 279, 281

NSCoding protocol, 279-281, 305

NSComparisonResult, 538

NSCopying, 541

NSData, 288, 305-305, 307

NSDate, 155, 232, 295, 323

NSDateFormatter, 155, 232, 342

NSDictionary, 249-251, 249

(see also dictionaries)

NSError, 294-295

NSException, 267

NSExpression, 338

NSFetchRequest, 327-329, 338

NSFileCoordinator, 555

NSFileManager, 561

NSFilePresenter protocol, 555

NSIndexPath, 200, 215

NSJSONSerialization, 521-524, 526

NSKeyedArchiver, 283-286

NSKeyedUnarchiver, 285

NSLocale, 342

NSLocalizedString, 346, 348

NSLog, 37

NSManagedObject, 322-325, 338

NSManagedObjectContext, 325-329, 338

NSManagedObjectModel, 325-327

NSMetadataQuery, 555

NSMutableArray, 35, 37-38, 37

(see also arrays)

removeObject:, 214

removeObjectIdenticalTo:, 214

sortUsingComparator:, 538

NSMutableDictionary, 249-251, 249

(see also dictionaries)

NSNotification, 173-174

NSNotificationCenter, 173-176

NSNull, 38

NSNumber, 295, 413

NSObject, 38-42
NSOperationQueue, 494
NSPersistentStoreCoordinator, 325-327
NSPredicate, 328
NSRegularExpression, 479-482
NSSearchPathDirectory, 282
NSSearchPathForDirectoriesInDomains, 282
NSSortOrdering, 338
NSString
 collecting from XIB , 344
 creating, 54
 creating with @, 36
 description, 39
 drawInRect:withFont:, 133
 internationalizing, 346
 as key path, 411
 literal, 36
 localizing, 344
 printing to console, 37
 property list serializable, 295
 stringWithFormat:, 54
 using tokens with, 37
 writing to filesystem, 289-294
NSStringFromSelector, 292
NSTemporaryDirectory, 282
NSTextCheckingResult, 479
NSTimeInterval, 323
NSUbiquitousKeyValueStore, 555
NSURL, 440-442
NSURLAuthenticationChallenge, 460
NSURLAuthenticationChallengeSender, 460
NSURLCache, 554
NSURLConnection, 441, 441-444, 459
NSURLCredential, 460-461
NSURLIsExcludedFromBackupKey, 567
NSURLRequest, 440-442, 459-460
NSUserDefaults, 282, 353-356
NSUserDefaultsDidChangeNotification, 356
NSValue, 362-364, 413
NSXMLParser, 444-455, 458
NSXMLParserDelegate, 446, 447, 449, 450, 458
number pad, 238

O

objc_msgSend, 390
object diagrams, 103-104
object files, 99

object library, 7
Object-Relational Mapping (ORM), 317
objectAtIndex:, 38
objectForKey:, 250-251
Objective-C
 basics, 29-57
 keywords, 42
 message names, 32, 32
 method names, 32
 naming conventions, 43, 47
 single inheritance in, 42
objects, 29
 (see also classes, memory management)
 allocation, 61
 copying, 76-77
 independence of, 42
 in memory, 61
 overview, 29-31
 ownership of, 63-63, 65-67
 property list serializable, 295
 size of, 61
offscreen contexts, 304-306
OmniGraffle, 103
operators (regular expressions), 480
optional methods (protocols), 89
Organizer window, 23
orientation
 and autorotation, 176-185
 on iPad, 178, 184, 263
 landscape mode, 184
 and split view controllers, 466
 UIDevice constants for, 174
orientationChanged:, 175
ORM (Object-Relational Mapping), 317
outlets, 13-17, 224
overriding methods, 46-47, 50-52

P

parentViewController, 269-270
parsing XML, 444-455
pathForResource ofType:, 349
paths (Core Graphics), 131
pattern strings (regular expressions), 479, 480-481
performSelector:withObject:, 310
placeholder objects, 152
placeholders (in code), 18, 19, 204

- pointers
 - in arrays, 37
 - and memory management, 65-67
 - overview, 30-31
 - setting in XIB files, 14-15
 - setting to nil, 32
 - as strong references, 67
 - syntax of, 42
 - as weak references, 67, 67
 - popover controllers, 264-265, 474
 - popoverControllerDidDismissPopover:**, 265
 - predicates (fetch requests), 328
 - predicateWithFormat:**, 328
 - preferences, 351-356
 - premature deallocation, 64
 - preprocessing, 98
 - preprocessor macros, 396-397
 - presentation layer, 420
 - presentedViewController, 275
 - presentingViewController, 269, 270, 275
 - presentViewController:animated:completion:**, 246, 272
 - products, 82, 392
 - profiling (applications), 381-382
 - project navigator, 3-5
 - projects
 - adding frameworks to, 399
 - build settings for, 395-397
 - cleaning and building, 346
 - copying files to, 191
 - creating new, 2, 3
 - defined, 82
 - diagram of, 392
 - target settings in, 296
 - properties, 72-74
 - atomic, 73
 - attributes of, 72
 - block, 492
 - copy, 77
 - creating from XIB file, 302, 302, 302
 - declaring, 77
 - and instance variables, 76
 - memory management of, 73
 - nonatomic, 73
 - overriding accessors, 75
 - in protocols, 117
 - readonly, 73
 - readwrite, 73
 - strong, 73
 - synthesizing, 74-76
 - weak, 73
 - property list serializable objects, 295
 - protocols
 - CLLocationManagerDelegate, 88
 - creating new, 469-474
 - declaring, 89
 - delegate, 88-90, 109, 469-474
 - described, 88-89
 - implementation of, 117
 - MKAnnotation, 114-117
 - MKMapViewDelegate, 109-112
 - NSCoding, 279-281, 305
 - NSCopying, 541
 - NSURLAuthenticationChallengeSender, 460
 - NSXMLParserDelegate, 446, 447, 449, 450
 - optional methods in, 89
 - properties in, 117
 - required methods in, 89
 - structure of, 88
 - UIApplicationDelegate, 288
 - UIGestureRecognizerDelegate, 374
 - UIImagePickerControllerDelegate, 245, 247-248
 - UINavigationControllerDelegate, 247
 - UIPopoverControllerDelegate, 264
 - UIResponderStandardEditActions, 376
 - UIScrollViewDelegate, 139
 - UISplitViewControllerDelegate, 475-477
 - UITableViewDataSource, 189, 196-198, 199, 215, 215
 - UITableViewDelegate, 189, 211
 - UITextFieldDelegate, 256
 - UITextInputTraits, 108
 - provisioning profiles, 23-23
 - proximity monitoring, 185
 - pushViewController:animated:**, 230-231
- ## Q
- Quartz (see Core Graphics)
 - QuartzCore, 399, 403
 - Quiz application, 2-27
 - quotation marks, 120
- ## R
- RandomPossessions application

creating, 33-37
creating **BNRItem** class, 39-56
readonly, 73
readwrite, 73
receiver, 31
registerDefaults:, 354
registration domain, 354
regular expressions, 478-482
relationships (Core Data), 320-322, 336-337
release, 78
removeObject:, 214
removeObjectIdenticalTo:, 214
removeObserver:, 174
reordering controls, 216
required methods (protocols), 89
requireGestureRecognizerToFail:, 377
resignFirstResponder, 118, 233
resizing views, 182-184
resources
 defined, 24, 296
 localizing, 343-346
responder chain, 364-365
responders (see first responder, **UIResponder**)
respondsToSelector:, 89, 311
retain, 78
retain cycles
 explained, 67-70
 finding with Leaks instrument, 390-391
Retina display, 24, 26, 170-171
reuseIdentifier (UITableViewCell), 201
reusing
 animation objects, 417
 classes, 191
 table view cells, 201-202
rootViewController
(UINavigationController), 220-222
rootViewController (UIWindow), 149
rotation, 176-185, 466
rows (**UITableView**)
 adding, 213-213
 deleting, 214-215
 moving, 215-217
run loop, 134, 134, 169, 494
run-time errors, 56-57

S

sandbox, application, 281-283, 296

scheme editor, 392
schemes, 23, 392-395
screenshots, taking, 26
scrolling, 137-139
sections (of **UITableView**), 198, 207-207
SEL, 235
selector, 31, 31, 235
self, 49-50, 54
sendAction:to:from:forEvent:, 366
sendActionsForControlEvents:, 366
setAutoresizingMask:, 182-184
setEditing:animated:, 212, 237
setMultipleTouchEnabled:, 361
setNeedsDisplay, 134, 408
setObject:forKey:, 250-251
setPagingEnabled:, 139
setPosition:, 405
setProximityMonitoringEnabled:, 185
setRegion:animated:, 114
setStatusBarHidden:withAnimation:, 141
setter methods, 43-45
setText:, 134
Settings application, 282, 355
settings, user, 351-356
shakes, detecting, 135-137
shouldAutorotateToInterfaceOrientation:,
177, 177, 184, 184, 466
showsUserLocation, 108
simulator
 for Retina display, 171
 running applications on, 20
 sandbox location, 285
 saving images to, 246
 simulating low-memory warnings, 162
 simulating shakes, 136
 simulating two fingers, 141
 viewing application bundle in, 296
single inheritance, 38, 42
singletons, implementing, 192-195
size inspector, 179-180
sort descriptors (**NSFetchRequest**), 327
sortedArrayUsingComparator:, 496
sortUsingComparator:, 538
sourceType (**UIImagePickerController**),
244-245
split view controllers (see
UISplitViewController)
splitViewController, 275, 467

SQL, 335
SQLite, 317, 325-327, 335, 561
square brackets, 31
stack (memory), 62, 92
stack trace, 91, 92, 95
standardUserDefaults, 353, 354
states, application, 286-288
static analyzer, 379-381
static tables, 427-430
static variables, 193
status bar, hiding, 141
store objects
 asynchronous, 525-525, 553
 benefits of, 505, 550-551
 and caching data, 530
 creating, 506-517
 described, 292
 designing, 552-553
 as singletons, 552
 synchronous, 553
storeCachedResponse:forRequest:, 554
storyboards
 creating, 423-427
 pros and cons, 434-435
 segues, 430-434
 static tables in, 427-430
 vs. XIB files, 423
strings (see **NSString**)
strings tables, 346-348
stringWithFormat:, 54
strong, 73
strong references, 67, 70, 73
structures, 29-30
 -> operator, 543
subclassing, 38-56, 50
 (see also overriding methods)
 as approach to callbacks, 501
 and method return type, 48
 use of **self**, 54
super, 50-50
superclasses, 38, 42, 50
superlayer, 404
superview, 126
suspended state, 287, 288
syntax errors, 99-99
system libraries, 100, 120

T

tab bar controllers (see **UITabBarController**)
tab bar items, 158-161
tabBarController, 275
table view cells (see **UITableViewCell**)
table view controllers (see **UITableViewController**)
UITableView, 213
tableView:cellForRowAtIndexPath:, 197, 199-202
tableView:commitEditingStyle:forRowAtIndexPath:, 215
tableView:didSelectRowAtIndexPath:, 232
tableView:heightForHeaderInSection:, 211
tableView:moveRowAtIndexPath:toIndexPath:, 215, 216
tableView:numberOfRowsInSection:, 197-198
tableView:viewForHeaderInSection:, 211
target-action pairs
 as approach to callbacks, 499, 499, 500
 defined, 15-17
 vs. delegation, 87
 setting programmatically, 235
 and **UIControl**, 365-366
 and **UIGestureRecognizer**, 368
targets
 adding frameworks to, 82
 build settings for, 395-397
 building, 97-101
 defined, 82
 and schemes, 392-395
 settings of, 296
templates
 application, 82
 class, 125, 169
 reasons to avoid, xvii, 169
 view controller, 169
textFieldShouldReturn:, 104, 118-119, 256
thumbnail images, creating, 304-308
Time Profiler instrument, 387-390
timing functions (animation), 417-418
tmp directory, 282
toggleEditingMode:, 212
tokens (in **NSString**), 37
toll-free bridging, 252-254

topViewController (UINavigationController), 220

touch events

- and animation, 405-406
- basics of, 358-359
- enabling multi-touch, 361
- handling interruptions, 363-364
- keeping track of, 362-362
- and responder chain, 364-365
- and target-action pairs, 365-366
- and **UIControl**, 365-366

touchesBegan:

withEvent:, 358, 358, 362, 405

touchesCancelled:

withEvent:, 358, 363

touchesEnded:

withEvent:, 358, 358, 363

touchesMoved:

withEvent:, 358, 358, 363, 405

TouchTracker application

- drawing lines, 359-364
- recognizing gestures, 367-377

transient attributes (Core Data), 324

translationInView:, 375

typecasting, 252

U

ubiquity containers, 556-561, 562, 564

UIActivityIndicatorView, 104, 108

UIAlertView, 295

UIApplication

- and events, 358
- and **main**, 169
- and responder chain, 364, 366

UIApplicationDelegate, 288

UIApplicationDidReceiveMemoryWarning Notification, 291

UIBarButtonItem, 235-237, 241-244, 256

UIButton, 308

UIColor, 132

UIControl, 256-257, 365-366

UIControlEventTouchUpInside, 365, 365

UIDevice

- currentDevice**, 263
- determining device type, 263
- notifications from, 174

UIDocument, 555

UIEventSubtypeMotionShake, 137

UIGestureRecognizer

- action messages of, 368, 373
- cancelTouchesInView**, 376
- chaining recognizers, 377
- delaying touches, 377
- described, 367
- detecting taps, 368-372
- enabling simultaneous recognizers, 374, 374
- intercepting touches from view, 368, 374, 376
- locationInView:**, 370
- long press, 373-374
- panning, 373, 374-376
- states, 373, 375, 377
- subclasses, 368, 378
- subclassing, 378
- translationInView:** (**UIPanGestureRecognizer**), 375
- and **UIResponder** methods, 376

UIGestureRecognizerDelegate protocol, 374

UIGraphics functions, 306-306, 408-409

UIGraphicsBeginImageContextWithOptions, 306

UIGraphicsEndImageContext, 306

UIGraphicsGetImageFromCurrentImageContext, 306

UIImage

- and **CGContextRef**, 408
- and **CGImage**, 403
- wrapping in **NSData**, 289, 305-306

UIImageJPEGRepresentation, 289

UIImagePickerController

- instantiating, 244-245
- on iPad, 264
- presenting, 246-248
- recording video with, 257-259
- in **UIPopoverController**, 264

UIImagePickerControllerDelegate, 245, 247-248

UIImageView

- aspect fit, 178
- described, 239-241

UIKeyboardDidShowNotification, 174

UIKit, 403

UILabel, 134

UILongPressGestureRecognizer, 373-374

UIMenuController, 371-372, 376

- UIModalPresentationCurrentContext, 278
- UIModalPresentationFormSheet, 270
- UIModalPresentationPageSheet, 270
- UIModalTransitionStyleFlipHorizontal, 273
- UINavigationController**, 220, 222-237
 - UINavigationController**, 220
 - (see also view controllers)
 - adding view controllers to, 230-231, 232
 - described, 220-223
 - instantiating, 222
 - managing view controller stack, 220
 - navigationController, 275
 - pushViewController:animated:**, 230-231
 - rootViewController, 220-221
 - in storyboards, 430, 431
 - topViewController, 220-221
 - and **UINavigationController**, 233-237
 - view, 220
 - viewControllers, 220
 - viewWillAppear:**, 232
 - viewWillDisappear:**, 232
- UINavigationControllerDelegate, 247
- UINavigationControllerItem**, 233-237
- UINib**, 304
- UIPanGestureRecognizer**, 373, 374-376
- UIPopoverController**, 264-265, 474
- UIPopoverControllerDelegate, 264
- UIResponder**
 - becomeFirstResponder**, 135
 - described, 118
 - menu actions, 376
 - and motion events, 135-137
 - and responder chain, 364-365
 - and touch events, 358
- UIResponderStandardEditActions protocol, 376
- UIScrollView**
 - scrolling, 137-139
 - zooming, 139-141
- UIScrollViewDelegate, 139
- UISegmentedControl**, 351
- UISplitViewController**
 - autorotating, 466-467
 - illegal on iPhone, 464
 - master and detail view controllers, 464-469
 - overview, 464-466
 - in portrait mode, 474-477
 - splitViewController, 275
- UISplitViewControllerDelegate, 475-477
- UIStoryboard**, 423-435
- UIStoryboardSegue**, 430
- UITabBarController**, 156-161, 219, 220
 - tabBarController, 275
 - view, 157
- UITabBarItem**, 158-161
- UITableView**, 187-189
 - (see also **UITableViewCell**, **UITableViewController**)
 - adding rows to, 213-213
 - deleting rows from, 214-215
 - editing mode of, 207, 212-212, 237, 300
 - editing property, 207, 212-212
 - footer view, 207
 - header view, 207-211
 - moving rows in, 215-217
 - populating, 191-201
 - sections, 198, 207-207
 - view, 191
- UITableViewCell**
 - adding images to, 304-308
 - cell styles, 199-199
 - contentView, 199-199, 299-300
 - creating interface with XIB file, 301-303
 - editing styles, 215
 - relaying actions from, 308-315
 - retrieving instances of, 199-201
 - reusing instances of, 201-202, 303
 - subclassing, 299-304
 - subviews, 198-199
- UITableViewCellEditingStyleDelete, 215
- UITableViewController**, 189
 - (see also **UITableView**)
 - adding rows, 213-213
 - creating in storyboard, 427-430
 - creating static tables, 427-430
 - data source methods, 196
 - dataSource, 191-198
 - deleting rows, 214-215
 - described, 189-190
 - editing property, 212
 - moving rows, 215-217
 - returning cells, 199-202
 - subclassing, 189-191
 - tableView**, 213
- UITableViewDataSource, 189, 196-198, 199, 215, 215
- UITableViewDelegate, 189, 211

UITapGestureRecognizer, 368-372

UITextField

- described, 104
- as first responder, 118-119, 118, 256, 366
- and keyboard, 118
- setting attributes of, 108-108, 238

UITextFieldDelegate, 256

UITextFieldTraits, 108

UITextView, 118

UIToolbar, 235, 241

UITouch, 358-362

UIUserInterfaceIdiomPad, 263

UIUserInterfaceIdiomPhone, 263

UIView, 121

- (see also **UIViewController**, views)
- autoresizingMask**, 182
- backgroundColor**, 130
- bounds, 129, 131
- defined, 1, 121-123
- drawRect:**, 128-131, 134
- endEditing:**, 233
- frame, 126, 129, 131
- instantiating, 126
- layer, 400
- setNeedsDisplay**, 134
- size and position, 402
- subclassing, 124-128
- superview, 126

UIViewController, 145

- (see also view controllers)
- defined, 145
- definesPresentationContext**, 278
- initWithNibName:bundle:**, 161
- instantiating, xvii
- loadView**, 148, 150, 163, 211, 229
- modalTransitionStyle**, 273
- modalViewController**, 269-270
- navigationController**, 230
- navigationItem**, 233
- parentViewController**, 269-270
- presentingViewController**, 269, 270
- splitViewController**, 467
- subclassing, 146
- template, 169
- view, 145, 152, 364
- viewControllers, 274
- viewDidLoad**, 162
- viewDidUnload**, 165
- viewWillAppear:**, 248
- and XIB files, xvii

UIWebView, 456-458

UIWindow

- described, 123-124
- and responder chain, 364
- rootViewController**, 149

unarchiveObjectWithFile:, 285

universal applications

- accommodating device differences, 263, 464, 473, 478, 478
- setting device family, 477
- user interfaces for, 181-182, 262-263
- using iPad-only classes, 467
- XIB files for, 262

unknown key exception, 229

unrecognized selector, 56, 311

unsafe unretained references, 71

URLForUbiquityContainerIdentifier:, 561

URLs, 441, 441

- (see also **NSURL**)

user interface

- drill-down, 219, 463
- hiding status bar, 141
- keyboard, 108, 256
- making universal, 181-182, 262-263
- scrolling, 137-139
- zooming (views), 139-141

userInterfaceIdiom (UIDevice), 263

utilities area, 7, 202

UUIDs, 251

V

variables, 62

- (see also instance variables, local variables, pointers)
- block, 486-489
- static, 193

variables view, 92, 93

video recording, 257-259

view controllers, 103-104

- (see also controller objects, **UIViewController**)
- adding to navigation controller, 230-231
- adding to popover controller, 264
- adding to split view controller, 464-466
- creating, 150

- creating delegate protocol for, 469-474
- creating in a storyboard, 423-435
- detail, 464
- families of, 274, 276
- initializing, 161-162
- lifecycle of, 161-168
- loading views, 152, 156, 162-167
- master, 464, 474
- modal, 246
- and number of views, 150
- passing data between, 231-232, 468
- presenting, 156
- relationships between, 274-278
- reloading subviews, 248
- role in application, 145
- templates for, 169
- unloading views, 164
- using XIB to create views, 150
- and view hierarchy, 145, 148
- and views, 145
- view hierarchy, 124-126, 145, 148
- viewController, 274
- viewController (UINavigationController)**, 220
- viewDidAppear:**, 167
- viewDidDisappear:**, 167
- viewDidLoad**, 162, 229, 263
- viewDidUnload**, 165
- viewForZoomingInScrollView:**, 140
- views, 1
 - (see also **UIView**)
 - adding to window, 124, 145, 148
 - autoresize masks for, 182-184
 - autoresizing, 179-180
 - autorotating, 176-185
 - creating, 124-128
 - defined, 1, 121-123
 - and drawing contexts, 128-134
 - and layers, 399-400
 - life cycle of, 162-168
 - loading, 150, 152, 156, 162-167
 - modal presentation of, 246
 - in Model-View-Controller, 9, 103-104
 - redrawing, 134
 - resizing, 179-180, 240-241
 - and run loop, 134
 - scrolling, 137-139
 - setting attributes of, 108

- size and position of, 126, 129, 131
- and subviews, 124-126, 145
- unloading, 164
- and view controllers, 145
- zooming, 139-141
- viewWillAppear:**, 167-168, 232-233, 232, 248
- viewWillDisappear:**, 167, 232

W

- weak, 73
- weak references
 - and instance variables, 71
 - declaring, 73
 - defined, 67
 - and outlets, 166
 - vs. unsafe unretained, 71
- web services
 - caching data, 529-545
 - credentials, 460-461
 - for data storage, 338
 - described, 438-439
 - and HTTP protocol, 459-460
 - parsing retrieved XML, 444-455
 - requesting data from, 440-444
 - security, 460-461
- Whereami application
 - adding a user preference, 351-356
 - changing map type, 351-352
 - configuring user interface, 105-108
 - finding and annotating locations, 108
 - object diagram, 103-104
- willAnimateRotationToInterface...**, 185
- workspaces (Xcode), 3
- writeToFile:atomically:**, 289
- writeToFile:atomically:encoding:error:**, 294

X

- Xcode, 3
 - (see also debugging tools, inspectors, Instruments, libraries, navigators, projects, simulator)
 - assistant editor, 224-229, 242-244
 - build settings, 395-397
 - building applications, 45
 - building interfaces, 5-17
 - canvas, 5

- canvas area, 6
- code-completion, 18, 202-204, 511
- console, 21
- containers, 392
- creating a class in, 39-41
- debug area, 21
- debugger, 91-97
- editor area, 5
- keyboard shortcuts, 229
- navigators, 3
- Organizer window, 23
- placeholders in, 18, 19, 204
- products, 82
- profiling applications in, 381-383
- projects, 82
- regular expression search, 482
- schemes, 23, 392-395
- size inspector, 179-180
- static analyzer, 379-381
- tabs, 229
- targets, 82
- templates, 2, 125, 169
- utilities area, 7, 202
- workspaces, 3

XIB files

- and archiving, 281
- bad connections in, 228
- connecting objects in, 13-17
- connecting with source files, 224-229, 242-244, 302, 302, 302
- creating, 150, 151-156
- creating properties from, 302, 302
- defined, 5
- editing in Xcode, 5-9
- File's Owner, 151-156
- and
 - initWithNibName:bundle:**, 161
- for iPad, 262
- loading manually, 210, 303
- localizing, 343-346
- making connections in, 224-229, 242-244
- naming, 162
- vs. NIB files, 5
- objects in, 151
- placeholders in, 152
- and properties, 302
- setting pointers in, 14-15
- vs. storyboards, 423
- top-level objects in, 166
- and **UINib**, 304
- in universal applications, 262
- and view hierarchy, 126
- when to use, 150, 156

XML

- collecting from web service, 442-444
- constructing tree, 446-454
- parsing, 444-455
- property lists, 295

Z

- zooming (maps), 109-114
- zooming (views), 139-141
- zPosition, 404-405



ABOUT US



Big
nerd
ranch

THE BIG NERD STORY

Big Nerd Ranch exists to broaden the minds of our students and the businesses of our clients. Whether we are training talented individuals or developing a company's mobile strategy, our core philosophy is integral to everything we do.

The brainchild of CEO Aaron Hillegass, Big Nerd Ranch has hosted more than 2,000 students at the Ranch since its inception in 2001. Over the past ten years, we have had the opportunity to work with some of the biggest companies in the world such as Apple, Samsung, Nokia, Google, AOL, Los Alamos National Laboratory and Adobe, helping them realize their programming goals. Our team of software engineers are among the brightest in the business and it shows in our work. We have developed dozens of innovative and flexible solutions for our clients.

The Story Behind the Hat

Back in 2001, Big Nerd Ranch founder, Aaron Hillegass, showed up at WWDC (World Wide Developers Conference) to promote the Big Nerd Ranch brand. Without the money to buy an expensive booth, Aaron donned a ten-gallon cowboy hat to draw attention while passing out Big Nerd literature to prospective students and clients. A week later, we landed our first big client and the cowboy hat has been synonymous with the Big Nerd brand ever since. Already easily recognizable at 6'5, Aaron can be spotted wearing his cowboy hat at speaking engagements and conferences all over the world.

The New Ranch – Opening 2012

In the continuing effort to perfect the student experience, Big Nerd Ranch is building its own facility. Located just 20 minutes from the Atlanta airport, the new Ranch will be a monastic learning center that encompasses Aaron Hillegass' vision for technical education featuring a state-of-the-art classroom, fine dining and exercise facilities.

www.bignerdranch.com