

Erica Sadun

Companion to *The Advanced
iOS 6 Developer's Cookbook*



Fourth Edition

iOS 6,
new **Objective-C**
features, and
more!

The Core iOS 6 Developer's Cookbook

Developer's Library



FREE SAMPLE CHAPTER

SHARE WITH OTHERS



The Core iOS 6 Developer's Cookbook

This page intentionally left blank

The Core iOS 6 Developer's Cookbook

Erica Sadun

◆ Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

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.

The author and publisher have taken care in the preparation of 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.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

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

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

International Sales
international@pearsoned.com

AirPlay, AirPort, AirPrint, AirTunes, App Store, Apple, the Apple logo, Apple TV, Aqua, Bonjour, the Bonjour logo, Cocoa, Cocoa Touch, Cover Flow, Dashcode, Finder, FireWire, iMac, Instruments, Interface Builder, iOS, iPad, iPhone, iPod, iPod touch, iTunes, the iTunes logo, Leopard, Mac, Mac logo, Macintosh, Multi-Touch, Objective-C, Quartz, QuickTime, QuickTime logo, Safari, Snow Leopard, Spotlight, and Xcode are trademarks of Apple, Inc., registered in the U.S. and other countries. OpenGL, or OpenGL Logo: OpenGL is a registered trademark of Silicon Graphics, Inc. The YouTube logo is a trademark of Google, Inc. Intel, Intel Core, and Xeon are trademarks of Intel Corp. in the United States and other countries.

Visit us on the Web: informit.com/aw

Copyright © 2013 Pearson Education, 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, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671-3447

ISBN-13: 978-0-321-88421-3

ISBN-10: 0-321-88421-3

Second Printing: May 2013

Editor-in-Chief

Mark Taub

Senior Acquisitions Editor

Trina MacDonald

Senior Development Editor

Chris Zahn

Managing Editor

Kristy Hart

Project Editor

Jovana San Nicolas-Shirley

Copy Editor

Keith Cline

Indexer

WordWise Publishing
Services

Proofreader

Debbie Williams

Technical Editors

Duncan Champney
Oliver Drobnik
Rich Wardwell

Editorial Assistant

Olivia Basegio

Cover Designer

Chuti Prasertsith

Compositor

Nonie Ratcliff



I dedicate this book with love to my husband, Alberto, who has put up with too many gadgets and too many SDKs over the years while remaining both kind and patient at the end of the day.



Contents at a Glance

Preface	xxii
1 Gestures and Touches	1
2 Building and Using Controls	49
3 Alerting the User	101
4 Assembling Views and Animations	129
5 View Constraints	169
6 Text Entry	209
7 Working with View Controllers	249
8 Common Controllers	299
9 Accessibility	341
10 Creating and Managing Table Views	353
11 Collection Views	403
12 A Taste of Core Data	441
13 Networking Basics	471
Appendix Objective-C Literals	495
Index	501

Table of Contents

Preface	xxii
1 Gestures and Touches	1
Touches	1
Phases	2
Touches and Responder Methods	3
Touching Views	4
Multi-Touch	4
Gesture Recognizers	5
Recipe: Adding a Simple Direct Manipulation Interface	5
Recipe: Adding Pan Gesture Recognizers	7
Recipe: Using Multiple Gesture Recognizers Simultaneously	9
Resolving Gesture Conflicts	12
Recipe: Constraining Movement	13
Recipe: Testing Touches	15
Recipe: Testing Against a Bitmap	17
Recipe: Drawing Touches Onscreen	19
Recipe: Smoothing Drawings	21
Recipe: Using Multi-Touch Interaction	24
Recipe: Detecting Circles	28
Recipe: Creating a Custom Gesture Recognizer	33
Recipe: Dragging from a Scroll View	35
Recipe: Live Touch Feedback	39
Enabling Touch Feedback	39
Intercepting and Forwarding Touch Events	40
Implementing the TOUCHkit Overlay View	41
Recipe: Adding Menus to Views	44
Summary	46
2 Building and Using Controls	49
The UIControl Class	49
Target-Action	50
Kinds of Controls	50
Control Events	51
Buttons	53

- Buttons in Interface Builder 54
 - Connecting Buttons to Actions 55
- Recipe: Building Buttons 55
 - Multiline Button Text 58
 - Adding Animated Elements to Buttons 58
 - Adding Extra State to Buttons 59
- Recipe: Animating Button Responses 59
- Recipe: Adding a Slider with a Custom Thumb 61
 - Customizing `UISlider` 62
 - Adding Efficiency 63
- Appearance Proxies 66
- Recipe: Creating a Twice-Tappable Segmented Control 69
 - Second-Tap Feedback 70
 - Controls and Attributes 70
- Working with Switches and Steppers 72
- Recipe: Subclassing `UIControl` 73
 - Creating Controls 74
 - Tracking Touches 74
 - Dispatching Events 74
- Recipe: Building a Star Slider 77
- Recipe: Building a Touch Wheel 80
- Recipe: Creating a Pull Control 83
 - Discoverability 83
 - Testing Touches 85
- Recipe: Building a Custom Lock Control 87
- Adding a Page Indicator Control 90
- Recipe: Creating a Customizable Paged Scroller 93
- Building Toolbars 98
- Summary 100

3 Alerting the User 101

- Talking Directly to Your User Through Alerts 101
 - Building Simple Alerts 102
 - Alert Delegates 103
 - Displaying the Alert 104
 - Kinds of Alerts 104

“Please Wait”: Showing Progress to Your User	105
Using <code>UIActivityIndicatorView</code>	106
Using <code>UIProgressView</code>	107
Recipe: No-Button Alerts	107
Building a Floating Progress Monitor	109
Recipe: Creating Modal Alerts with Run Loops	110
Recipe: Using Variadic Arguments with Alert Views	113
Presenting Lists of Options	114
Scrolling Menus	117
Displaying Text in Action Sheets	117
Recipe: Building Custom Overlays	117
Tappable Overlays	119
Recipe: Basic Popovers	119
Recipe: Local Notifications	121
Best Practices	122
Alert Indicators	123
Badging Applications	124
Recipe: Simple Audio Alerts	124
System Sounds	124
Vibration	125
Alerts	126
Delays	126
Disposing of System Sounds	126
Summary	128

4 Assembling Views and Animations 129

View Hierarchies	129
Recipe: Recovering a View Hierarchy Tree	131
Exploring XIB and Storyboard Views	132
Recipe: Querying Subviews	133
Managing Subviews	134
Adding Subviews	134
Reordering and Removing Subviews	135
View Callbacks	135
Tagging and Retrieving Views	136
Using Tags to Find Views	136

Recipe: Naming Views by Object Association	137
Naming Views in Interface Builder	137
View Geometry	139
Frames	140
Rectangle Utility Functions	140
Points and Sizes	141
Transforms	142
Coordinate Systems	142
Recipe: Working with View Frames	143
Adjusting Sizes	144
CGRects and Centers	146
Other Geometric Elements	147
Recipe: Retrieving Transform Information	151
Retrieving Transform Properties	151
Testing for View Intersection	152
Display and Interaction Traits	157
UIView Animations	158
Building Animations with Blocks	159
Recipe: Fading a View In and Out	159
Recipe: Swapping Views	161
Recipe: Flipping Views	162
Recipe: Using Core Animation Transitions	163
Recipe: Bouncing Views as They Appear	165
Recipe: Image View Animations	166
Summary	167

5 View Constraints 169

What Are Constraints?	169
Alignment Rectangles	170
Declaring Alignment Rectangles	171
Constraint Attributes	171
Constraint Math	172
The Laws of Constraints	173
Creating Constraints	175
Basic Constraint Declarations	175
Visual Format Constraints	176
Variable Bindings	177

Format Strings	177
Orientation	177
View Names	179
Connections	179
Predicates	183
Metrics	183
View-to-View Predicates	184
Priorities	184
Format String Summary	184
Storing and Updating Constraints	186
Recipe: Comparing Constraints	187
Recipe: Describing Constraints	189
Recipe: Creating Fixed-Size Constrained Views	192
Disabling Autosizing Constraints	192
Starting within View Bounds	193
Constraining Size	193
Recipe: Centering Views	196
Recipe: Setting Aspect Ratio	197
Aligning Views and Flexible Sizing	199
Why You Cannot Distribute Views	199
Recipe: Responding to Orientation Changes	200
Constraint Macros	202
Consistent Constraints	202
Sufficient Constraints	203
Macros	203
Debugging Your Constraints	205
Summary	207
6 Text Entry	209
Recipe: Dismissing a UITextField Keyboard	209
Preventing Keyboard Dismissal	210
Text Trait Properties	211
Other Text Field Properties	212
Recipe: Dismissing Text Views with Custom Accessory Views	213
Recipe: Adjusting Views Around Keyboards	216

- Recipe: Adjusting Views Around Accessory Views 220
 - Testing for Hardware Keyboards 221
- Recipe: Creating a Custom Input View 223
- Recipe: Making Text-Input-Aware Views 227
- Recipe: Adding Custom Input Views to Nontext Views 230
 - Adding Input Clicks 231
- Recipe: Building a Better Text Editor (Part I) 233
- Recipe: Building a Better Text Editor (Part II) 236
 - Enabling Attributed Text 236
 - Controlling Attributes 236
 - Other Responder Functionality 237
- Recipe: Text-Entry Filtering 239
- Recipe: Detecting Text Patterns 242
 - Rolling Your Own Expressions 242
 - Enumerating Regular Expressions 243
 - Data Detectors 243
 - Using Built-In Type Detectors 244
 - Useful Websites 244
- Recipe: Detecting Misspelling in a UITextView 246
- Searching for Text Strings 247
- Summary 248

7 Working with View Controllers 249

- View Controllers 249
 - The UIViewController Class 250
 - Navigation Controllers 250
 - Tab Bar Controllers 251
 - Split View Controllers 251
 - Page View Controller 251
 - Popover Controllers 251
- Developing with Navigation Controllers and Split Views 252
 - Using Navigation Controllers and Stacks 253
 - Pushing and Popping View Controllers 254
 - Bar Buttons 254

Recipe: The Navigation Item Class	255
Titles and Back Buttons	255
Macros	256
Recipe: Modal Presentation	257
Presenting a Custom Modal Information View	258
Recipe: Building Split View Controllers	261
Recipe: Creating Universal Split View/Navigation Apps	266
Recipe: Tab Bars	268
Remembering Tab State	272
Recipe: Page View Controllers	275
Book Properties	276
Wrapping the Implementation	277
Exploring the Recipe	279
Building a Presentation Index	279
Recipe: Scrubbing Pages in a Page View Controller	285
Recipe: Custom Containers	286
Adding and Removing a Child View Controller	287
Transitioning Between View Controllers	288
Recipe: Segues	292
Segues, Interface Builder, and iOS 6	297
Summary	298

8 Common Controllers 299

Image Picker Controller	299
Image Sources	299
Presenting the Picker on iPhone and iPad	300
Recipe: Selecting Images	300
How To: Adding Photos to the Simulator	301
The Assets Library Framework	301
Presenting a Picker	302
Handling Delegate Callbacks	303
Recipe: Snapping Photos	306
Setting Up the Picker	307
Displaying Images	308
Saving Images to the Photo Album	309

- Recipe: Recording Video 310
 - Creating the Video Recording Picker 311
 - Saving the Video 311
- Recipe: Playing Video with Media Player 313
- Recipe: Editing Video 316
 - AV Foundation and Core Media 316
- Recipe: Picking and Editing Video 318
- Recipe: E-Mailing Pictures 321
 - Creating Message Contents 321
- Recipe: Sending a Text Message 323
- Recipe: Posting Social Updates 325
- Recipe: Activity View Controller 328
 - Creating and Presenting the Controller 328
 - Adding Services 331
 - Items and Services 335
- Recipe: The Quick Look Preview Controller 336
 - Implementing Quick Look 337
- Summary 340

9 Accessibility 341

- Accessibility 101 341
 - Accessibility in Interface Builder 342
- Enabling Accessibility 343
- Traits 344
- Labels 345
- Hints 346
- Testing with the Simulator 347
- Broadcasting Updates 348
- Testing Accessibility on the iPhone 349
- Summary 351

10 Creating and Managing Table Views 353

- iOS Tables 353
- Delegation 354
- Creating Tables 355
 - Table Styles 355
 - Laying Out the View 355

Assigning a Data Source	356
Serving Cells	356
Registering Cell Classes	356
Dequeueing Cells	357
Assigning a Delegate	357
Recipe: Implementing a Basic Table	358
Data Source Methods	359
Responding to User Touches	359
Table View Cells	361
Selection Color	362
Adding in Custom Selection Traits	363
Recipe: Creating Checked Table Cells	363
Working with Disclosure Accessories	365
Recipe: Table Edits	367
Adding Undo Support	368
Supporting Undo	369
Displaying Remove Controls	369
Handling Delete Requests	369
Swiping Cells	370
Reordering Cells	370
Adding Cells	370
Recipe: Working with Sections	375
Building Sections	375
Counting Sections and Rows	376
Returning Cells	377
Creating Header Titles	378
Customizing Headers and Footers	379
Creating a Section Index	379
Handling Section Mismatches	380
Delegation with Sections	380
Recipe: Searching Through a Table	382
Creating a Search Display Controller	383
Registering Cells for the Search Display Controller	384
Building the Searchable Data Source Methods	384
Delegate Methods	385
Using a Search-Aware Index	386

- Recipe: Adding Pull-to-Refresh to Your Table 388
- Recipe: Adding Action Rows 391
- Coding a Custom Group Table 394
 - Creating Grouped Preferences Tables 395
- Recipe: Building a Multiwheel Table 396
 - Creating the UIPickerView 397
 - Data Source and Delegate Methods 397
 - Using Views with Pickers 397
- Using the UIDatePicker 399
 - Creating the Date Picker 400
- Summary 401

11 Collection Views 403

- Collection Views Versus Tables 403
 - Practical Implementation Differences 405
- Establishing Collection Views 405
 - Controllers 405
 - Views 406
 - Data Sources and Delegates 406
- Flow Layouts 407
 - Scroll Direction 407
 - Item Size and Line Spacing 407
 - Header and Footer Sizing 409
 - Insets 410
- Recipe: Basic Collection View Flows 411
- Recipe: Custom Cells 415
- Recipe: Scrolling Horizontal Lists 416
- Recipe: Introducing Interactive Layout Effects 420
- Recipe: Scroll Snapping 422
- Recipe: Creating a Circle Layout 423
 - Creation and Deletion Animation 424
 - Powering the Circle Layout 424
 - The Layout 425
- Recipe: Adding Gestures to Layout 429
- Recipe: Creating a True Grid Layout 431
- Recipe: Custom Item Menus 437
- Summary 439

12 A Taste of Core Data 441

- Introducing Core Data 441
- Entities and Models 442
 - Building a Model File 442
 - Attributes and Relationships 443
 - Building Object Classes 444
- Creating Contexts 444
- Adding Data 445
 - Examining the Data File 446
- Querying the Database 448
 - Setting Up the Fetch Request 449
 - Performing the Fetch 449
- Removing Objects 450
- Recipe: Using Core Data for a Table Data Source 451
 - Index Path Access 451
 - Section Key Path 451
 - Section Groups 452
 - Index Titles 452
 - Table Readiness 452
- Recipe: Search Tables and Core Data 455
- Recipe: Adding Edits to Core Data Table Views 457
 - Adding Undo/Redo Support 458
 - Creating Undo Transactions 459
 - Rethinking Edits 459
- Recipe: A Core Data-Powered Collection View 464
- Summary 469

13 Networking Basics 471

- Recipe: Checking Your Network Status 471
- Scanning for Connectivity Changes 474
- Recipe: Synchronous Downloads 476
- Recipe: Asynchronous Downloads 480
- One-Call No-Feedback Asynchronous Downloads 486
- Recipe: Using JSON Serialization 487
- Recipe: Converting XML into Trees 489
 - Trees 489
 - Building a Parse Tree 490
- Summary 492

Appendix Objective-C Literals 495

Numbers 495

Boxing 496

 Enums 496

Container Literals 497

Subscripting 498

Feature Tests 499

Index 501

Acknowledgments

This book would not exist without the efforts of Chuck Toporek, who was my editor and whipcracker for many years and multiple publishers. He is now at Apple and deeply missed. There'd be no Cookbook were it not for him. He balances two great skill sets: inspiring authors to do what they think they cannot, and wielding the large "reality trout" of whacking¹ to keep subject matter focused and in the real world. There's nothing like being smacked repeatedly by a large virtual fish to bring a book in on deadline and with compelling content.

Thanks go as well to Trina MacDonald (my terrific new editor), Chris Zahn (the awesomely talented development editor), and Olivia Basegio (the faithful and rocking editorial assistant who kept things rolling behind the scenes). Also, a big thank you to the entire Addison-Wesley/Pearson production team, specifically Kristy Hart, Jovana San Nicolas-Shirley, Keith Cline, Larry Sweazy, Debbie Williams, Nonie Ratcliff, and Chuti Prasertsith. Thanks also to the crew at Safari for getting my book up in Rough Cuts and for quickly fixing things when technical glitches occurred.

Thanks go as well to Neil Salkind, my agent of many years, to the tech reviewers Oliver Drobnik, Rich Wardwell, and Duncan Champney, who helped keep this book in the realm of sanity rather than wishful thinking, and to all my colleagues, both present and former, at TUAW, Ars Technica, and the Digital Media/Inside iPhone blog.

I am deeply indebted to the wide community of iOS developers, including Jon Bauer, Tim Burks, Matt Martel, Tim Isted, Joachim Bean, Aaron Basil, Roberto Gamboni, John Muchow, Scott Mikolaitis, Alex Schaefer, Nick Penree, James Cuff, Jay Freeman, Mark Montecalvo, August Joki, Max Weisel, Optimo, Kevin Brosius, Planetbeing, Pytey, Michael Brennan, Daniel Gard, Michael Jones, Roxfan, MuscleNerd, np101137, UnterPerro, Jonathan Watmough, Youssef Francis, Bryan Henry, William DeMuro, Jeremy Sinclair, Arshad Tayyeb, Jonathan Thompson, Dustin Voss, Daniel Peebles, ChronicProductions, Greg Hartstein, Emanuele Vulcano, Sean Heber, Josh Bleacher Snyder, Eric Chamberlain, Steven Troughton-Smith, Dustin Howett, Dick Applebaum, Kevin Ballard, Hamish Allan, Oliver Drobnik, Rod Strougo, Kevin McAllister, Jay Abbott, Tim Grant Davies, Maurice Sharp, Chris Samuels, Chris Greening, Jonathan Willing, Landon Fuller, Jeremy Tregunna, Wil Macaulay, Stefan Hafeneger, Scott Yelich, chralllinder, John Varghese, Andrea Fanfani, J. Roman, jtbandes, Artissimo, Aaron Alexander, Christopher Campbell Jensen, Nico Ameghino, Jon Moody, Julián Romero, Scott Lawrence, Evan K. Stone, Kenny Chan Ching-King, Matthias Ringwald, Jeff Tentschert, Marco Fanciulli, Neil Taylor, Sjoerd van Geffen, Absentia, Nownot, Emerson Malca, Matt Brown, Chris Foresman, Aron Trimble, Paul Griffin, Paul Robichaux, Nicolas Haunold, Anatol Ulrich (hypnocode GmbH), Kristian Glass, Remy "psy" Demarest, Yanik Magnan, ashikase, Shane Zatezalo, Tito Ciuro, Mahipal Raythaththa, Jonah Williams of Carbon Five, Joshua Weinberg, biappi, Eric Mock, and everyone at the iPhone developer channels at irc.saurik.com and irc.freenode.net, among many others too numerous to name individually. Their techniques, suggestions, and feedback helped make this book possible. If I have overlooked anyone who helped contribute, please accept my apologies for the oversight.

Special thanks go out to my family and friends, who supported me through month after month of new beta releases and who patiently put up with my unexplained absences and frequent howls of despair. I appreciate you all hanging in there with me. And thanks to my children for their steadfastness, even as they learned that a hunched back and the sound of clicking keys is a pale substitute for a proper mother. My kids provided invaluable assistance over the past few months by testing applications, offering suggestions, and just being awesome people. I try to remind myself on a daily basis how lucky I am that these kids are part of my life.

About the Author

Erica Sadun is the bestselling author, coauthor, and contributor to several dozen books on programming, digital video and photography, and web design, including the widely popular *The iOS 5 Developer's Cookbook*. She currently blogs at TUAUW.com, and has blogged in the past at O'Reilly's Mac Devcenter, Lifehacker, and Ars Technica. In addition to being the author of dozens of iOS-native applications, Erica holds a Ph.D. in Computer Science from Georgia Tech's Graphics, Visualization and Usability Center. A geek, a programmer, and an author, she's never met a gadget she didn't love. When not writing, she and her geek husband parent three geeks-in-training, who regard their parents with restrained bemusement, when they're not busy rewiring the house or plotting global dominance.

Preface

Welcome to a new iOS Cookbook!

With iOS 6, Apple’s mobile device family has reached new levels of excitement and possibility. This Cookbook is here to help you get started developing. This revision introduces all new features announced at the latest WWDC, showing you how to incorporate them into your applications.

For this edition, my publishing team has sensibly split the Cookbook material into manageable print volumes. This book, *The Core iOS 6 Developer’s Cookbook*, provides solutions for the heart of day-to-day development. It covers all the classes you need for creating iOS applications using standard APIs and interface elements. It offers the recipes you need for working with graphics, touches, and views to create mobile applications.

A second volume, *The Advanced iOS 6 Developer’s Cookbook*, centers on common frameworks such as Store Kit, Game Kit, and Core Location. It helps you build applications that leverage these special-purpose libraries and move beyond the basics. This volume is for those who have a strong grasp on iOS development and are looking for practical how-to for specialized areas.

Finally, there’s *Learning iOS 6: A Hands-on Guide to the Fundamentals of iOS Programming*, which will cover much of the tutorial material that used to comprise the first several chapters of the Cookbook. There you’ll find all the fundamental how-to you need to learn iOS 6 development from the ground up. From Objective-C to Xcode, debugging to deployment, *Learning iOS 6* teaches you how to get started with Apple’s development tool suite.

As in the past, you can find sample code at GitHub. You’ll find the repository for this Cookbook at <https://github.com/erica/iOS-6-Cookbook>, all of it refreshed for iOS 6 after WWDC 2012.

If you have suggestions, bug fixes, corrections, or any thing else you’d like to contribute to a future edition, please contact me at erica@ericasadun.com. Let me thank you all in advance. I appreciate all feedback that helps make this a better, stronger book.

—Erica Sadun, September 2012

What You’ll Need

It goes without saying that, if you’re planning to build iOS applications, you’re going to need at least one iOS device to test out your application, preferably a new model iPhone or tablet. The following list covers the basics of what you need to begin:

- **Apple’s iOS SDK**—You can download the latest version of the iOS SDK from Apple’s iOS Dev Center (<http://developer.apple.com/ios>). If you plan to sell apps through the App Store, become a paid iOS developer. This costs \$99/year for individuals and \$299/year for enterprise (that is, corporate) developers. Registered developers receive certificates that allow them to “sign” and download their applications to their iPhone/iPod touch for testing and debugging and to gain early access to prerelease versions of iOS. Free-program

developers can test their software on the Mac-based simulator but cannot deploy to device or submit to the App Store.

University Student Program

Apple also offers a University Program for students and educators. If you are a computer science student taking classes at the university level, check with your professor to see whether your school is part of the University Program. For more information about the iPhone Developer University Program, see <http://developer.apple.com/support/iphone/university>.

- **A modern Mac running Mac OS X Lion (v 10.7) or, preferably, Mac OS X Mountain Lion (v 10.8)**—You need plenty of disk space for development, and your Mac should have as much RAM as you can afford to put into it.
- **An iOS device**—Although the iOS SDK includes a simulator for you to test your applications in, you really do need to own iOS hardware to develop for the platform. You can tether your unit to the computer and install the software you’ve built. For real-life App Store deployment, it helps to have several units on hand, representing the various hardware and firmware generations, so that you can test on the same platforms your target audience will use.
- **An Internet connection**—This connection enables you to test your programs with a live Wi-Fi connection as well as with an EDGE or 3G service.
- **Familiarity with Objective-C**—To program for the iPhone, you need to know Objective-C 2.0. The language is based on ANSI C with object-oriented extensions, which means you also need to know a bit of C too. If you have programmed with Java or C++ and are familiar with C, you should be able to make the move to Objective-C.

Your Roadmap to Mac/iOS Development

One book can’t be everything to everyone. Try as I might, if we were to pack everything you need to know into this book, you wouldn’t be able to pick it up. (As it stands, this book offers an excellent tool for upper-body development. Please don’t sue if you strain yourself lifting it.) There is, indeed, a lot you need to know to develop for the Mac and iOS platforms. If you are just starting out and don’t have any programming experience, your first course of action should be to take a college-level course in the C programming language. Although the alphabet might start with the letter A, the root of most programming languages, and certainly your path as a developer, is C.

Once you know C and how to work with a compiler (something you’ll learn in that basic C course), the rest should be easy. From there, you’ll hop right on to Objective-C and learn how to program with that alongside the Cocoa frameworks. The flowchart shown in Figure P-1 shows you key titles offered by Pearson Education that can help provide the training you need to become a skilled iOS developer.

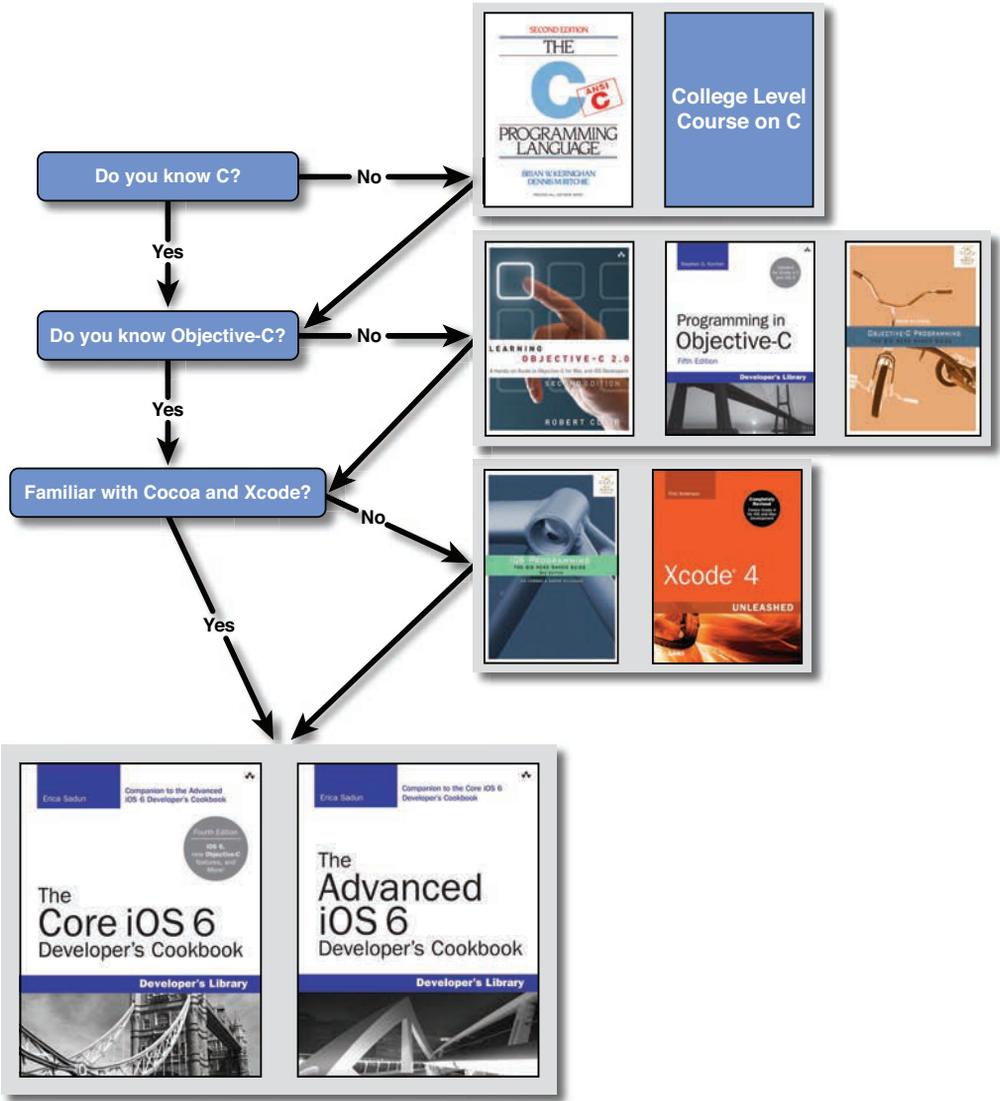


Figure P-1 A roadmap to becoming an iOS developer.

Once you know C, you've got a few options for learning how to program with Objective-C. If you want an in-depth view of the language, you can either read Apple's own documentation or pick up one of these books on Objective-C:

- *Objective-C Programming: The Big Nerd Ranch Guide*, by Aaron Hillegass (Big Nerd Ranch, 2012)

- *Learning Objective-C 2.0: A Hands-on Guide to Objective-C for Mac and iOS Developers*, by Robert Clair (Addison-Wesley, 2012)
- *Programming in Objective-C, Fifth Edition*, by Stephen Kochan (Addison-Wesley, 2012)

With the language behind you, next up is tackling Cocoa and the developer tools, otherwise known as Xcode. For that, you have a few different options. Again, you can refer to Apple's own documentation on Cocoa and Xcode,² or if you prefer books, you can learn from the best. Aaron Hillegass, founder of the Big Nerd Ranch in Atlanta,³ is the coauthor of *iOS Programming: The Big Nerd Ranch Guide, Second Edition*, and author of *Cocoa Programming for Mac OS X*, soon to be in its fourth edition. Aaron's book is highly regarded in Mac developer circles and is the most-recommended book you'll see on the cocoa-dev mailing list. To learn more about Xcode, look no further than Fritz Anderson's *Xcode 4 Unleashed* from Sams Publishing.

Note

There are plenty of other books from other publishers on the market, including the bestselling *Beginning iPhone 4 Development*, by Dave Mark, Jack Nutting, and Jeff LaMarche (Apress, 2011). Another book that's worth picking up if you're a total newbie to programming is *Beginning Mac Programming*, by Tim Isted (Pragmatic Programmers, 2011). Don't just limit yourself to one book or publisher. Just as you can learn a lot by talking with different developers, you will learn lots of tricks and tips from other books on the market.

To truly master Mac development, you need to look at a variety of sources: books, blogs, mailing lists, Apple's own documentation, and, best of all, conferences. If you get the chance to attend WWDC, you'll know what I'm talking about. The time you spend at those conferences talking with other developers, and in the case of WWDC, talking with Apple's engineers, is well worth the expense if you are a serious developer.

How This Book Is Organized

This book offers single-task recipes for the most common issues new iOS developers face: laying out interface elements, responding to users, accessing local data sources, and connecting to the Internet. Each chapter groups together related tasks, allowing you to jump directly to the solution you're looking for without having to decide which class or framework best matches that problem.

The Core iOS 6 Developer's Cookbook offers you "cut-and-paste convenience," which means you can freely reuse the source code from recipes in this book for your own applications and then tweak the code to suit your app's needs.

Here's a rundown of what you find in this book's chapters:

- **Chapter 1, "Gestures and Touches"**—On iOS, the touch provides the most important way that users communicate their intent to an application. Touches are not limited to button presses and keyboard interaction. This chapter introduces direct manipulation

interfaces, Multi-Touch, and more. You see how to create views that users can drag around the screen and read about distinguishing and interpreting gestures, as well as how to create custom gesture recognizers.

- **Chapter 2, “Building and Using Controls”**—Take your controls to the next level. This chapter introduces everything you need to know about how controls work. You discover how to build and customize controls in a variety of ways. From the prosaic to the obscure, this chapter introduces a range of control recipes you can reuse in your programs.
- **Chapter 3, “Alerting the User”**—iOS offers many ways to provide users with a heads-up, from pop-up dialogs and progress bars to local notifications, popovers, and audio pings. Chapter 3 shows how to build these indications into your applications and expand your user-alert vocabulary. It introduces standard ways of working with these classes and offers solutions that allow you to craft linear programs without explicit callbacks.
- **Chapter 4, “Assembling Views and Animations”**—The `UIView` class and its subclasses populate the iOS device screens. This chapter introduces views from the ground up. This chapter dives into view recipes, exploring ways to retrieve, animate, and manipulate view objects. You learn how to build, inspect, and break down view hierarchies and understand how views work together. You discover the role geometry plays in creating and placing views into your interface, and you read about animating views so they move and transform onscreen.
- **Chapter 5, “View Constraints”**—The iOS 6 software development kit (SDK) revolutionized view layout. Apple’s layout features are about to make your life easier and your interfaces more consistent. This is especially important when working across members of the same device family with different screen sizes—like the iPhone 4S and the iPhone 5. This chapter introduces code-level constraint development. You’ll discover how to create relations between onscreen objects and specify the way iOS automatically arranges your views. The outcome is a set of robust rules that adapt to screen geometry.
- **Chapter 6, “Text Entry”**—Chapter 6 introduces text recipes that support a wide range of solutions. You’ll read about controlling keyboards, making onscreen elements “text aware,” scanning text, formatting text, and so forth. From text fields and text views to iOS’s inline spelling checkers, this chapter introduces everything you need to know to work with iOS text in your apps.
- **Chapter 7, “Working with View Controllers”**—Discover the various view controller classes that enable you to enlarge and order the virtual spaces your users interact with, learning from how-to recipes that cover page view controllers, split view controllers, navigation controllers, and more.
- **Chapter 8, “Common Controllers”**—The iOS SDK provides a wealth of system-supplied controllers that you can use in your day-to-day development tasks. This chapter introduces some of the most popular ones. You read about selecting images from your photo library, snapping photos, and recording and editing videos. You discover how to allow users to compose e-mails and text messages, and how to post updates to social services such as Twitter and Facebook.

- **Chapter 9, “Accessibility”**—This chapter offers a brief overview of VoiceOver accessibility to extend your audience to the widest possible range of users. You read about adding accessibility labels and hints to your applications and testing those features in the simulator and on the iOS device.
- **Chapter 10, “Creating and Managing Table Views”**—Tables provide a scrolling interaction class that works particularly well both on smaller devices and as a key player on larger tablets. Many iOS apps center on tables due to their simple natural organization features. Chapter 10 introduces tables. It explains how tables work, what kinds of tables are available to you as a developer, and how you can leverage table features in your applications.
- **Chapter 11, “Collection Views”**—Collection views use many of the same concepts as tables but provide more power and more flexibility. This chapter walks you through all the basics you need to get started. Prepare to read about creating side-scrolling lists, grids, one-of-a-kind layouts like circles, and more. You’ll learn about integrating visual effects through layout specifications and snapping items into place after scrolling, and you’ll discover how to take advantage of built-in animation support to create the most effective interactions possible.
- **Chapter 12, “A Taste of Core Data”**—Core Data offers managed data stores that can be queried and updated from your application. It provides a Cocoa Touch–based object interface that brings relational data management out from SQL queries and into the Objective-C world of iOS development. Chapter 12 introduces Core Data. It provides just enough recipes to give you a taste of the technology, offering a jumping-off point for further Core Data learning. You learn how to design managed database stores, add and delete data, and query that data from your code and integrate it into your UIKit table views and collection views.
- **Chapter 13, “Networking Basics”**—As an Internet-connected device, iOS is particularly suited to subscribing to Web-based services. Apple has lavished the platform with a solid grounding in all kinds of network computing services and their supporting technologies. Chapter 13 surveys common techniques for network computing and offers recipes that simplify day-to-day tasks. You read about network reachability, synchronous and asynchronous downloads, using operation queues, working with the iPhone’s secure keychain to meet authentication challenges, XML parsing, JSON serialization, and more.
- **Appendix, “Objective-C Literals”**—This appendix introduces new Objective-C constructs for specifying numbers, arrays, and dictionaries.

About the Sample Code

For the sake of pedagogy, this book’s sample code uses a single main.m file. This is not how people normally develop iPhone or Cocoa applications, or, honestly, how they should be developing them, but it provides a great way of presenting a single big idea. It’s hard to tell a story when readers must look through five or seven or nine individual files at once. Offering a single file concentrates that story, allowing access to that idea in a single chunk.

These examples are not intended as stand-alone applications. They are there to demonstrate a single recipe and a single idea. One `main.m` file with a central presentation reveals the implementation story in one place. Readers can study these concentrated ideas and transfer them into normal application structures, using the standard file structure and layout. The presentation in this book does not produce code in a standard day-to-day best-practices approach. Instead, it reflects a pedagogy that offers concise solutions that you can incorporate back into your work as needed.

Contrast that to Apple's standard sample code, where you must comb through many files to build up a mental model of the concepts that are being demonstrated. Those examples are built as full applications, often doing tasks that are related to but not essential to what you need to solve. Finding just those relevant portions is a lot of work. The effort may outweigh any gains.

In this book, you'll find exceptions to this one-file-with-the-story rule: the Cookbook provides standard class and header files when a class implementation is the recipe. Instead of highlighting a technique, some recipes offer these classes and categories (that is, extensions to a preexisting class rather than a new class). For those recipes, look for separate `.m` and `.h` files in addition to the skeletal `main.m` that encapsulates the rest of the story.

For the most part, the examples for this book use a single application identifier: `com.sadun.helloworld`. This book uses one identifier to avoid clogging up your iOS devices with dozens of examples at once. Each example replaces the previous one, ensuring that your home screen remains relatively uncluttered. If you want to install several examples simultaneously, simply edit the identifier, adding a unique suffix, such as `com.sadun.helloworld.table-edits`. You can also edit the custom display name to make the apps visually distinct. Your Team Provisioning Profile matches every application identifier, including `com.sadun.helloworld`. This allows you to install compiled code to devices without having to change the identifier; just make sure to update your signing identity in each project's build settings.

Getting the Sample Code

You'll find the source code for this book at github.com/erica/iOS-6-Cookbook on the open-source GitHub hosting site. There, you find a chapter-by-chapter collection of source code that provides working examples of the material covered in this book. Recipes are numbered as they are in the book. Recipe 6 in Chapter 5, for example, appears in the `C05` folder in the `06` subfolder.

Any project numbered `00` or that has a suffix (like `05b` or `02c`) refers to material used to create in-text coverage and figures. For example, Chapter 10's `00 – Cell Types` project helped build Figure 10-2, showing system-supplied table view cells styles. Normally I delete these extra projects. Early readers of this manuscript requested that I include them in this edition. You'll find a half dozen or so of these extra samples scattered around the repository.

Contribute!

Sample code is never a fixed target. It continues to evolve as Apple updates its SDK and the Cocoa Touch libraries. Get involved. You can pitch in by suggesting bug fixes and corrections as well as by expanding the code that's on offer. GitHub allows you to fork repositories and grow them with your own tweaks and features, and share those back to the main repository. If you come up with a new idea or approach, let me know. My team and I are happy to include great suggestions both at the repository and in the next edition of this Cookbook.

Getting Git

You can download this Cookbook's source code using the git version control system. An OS X implementation of git is available at <http://code.google.com/p/git-osx-installer>. OS X git implementations include both command-line and GUI solutions, so hunt around for the version that best suits your development needs.

Getting GitHub

GitHub (<http://github.com>) is the largest git-hosting site, with more than 150,000 public repositories. It provides both free hosting for public projects and paid options for private projects. With a custom Web interface that includes wiki hosting, issue tracking, and an emphasis on social networking of project developers, it's a great place to find new code or collaborate on existing libraries. You can sign up for a free account at their website, allowing you to copy and modify the Cookbook repository or create your own open-source iOS projects to share with others.

Contacting the Author

If you have any comments or questions about this book, please drop me an e-mail message at erica@ericasadun.com, or stop by the github repository and contact me there.

Endnotes

¹ No trouts, real or imaginary, were hurt in the development and production of this book. The same cannot be said for countless cans of Diet Coke, who selflessly surrendered their contents in the service of this manuscript.

² See the *Cocoa Fundamentals Guide* (<http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaFundamentals.pdf>) for a head start on Cocoa, and for Xcode, see *A Tour of Xcode* (http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/A_Tour_of_Xcode/A_Tour_of_Xcode.pdf).

³ Big Nerd Ranch: www.bignerdranch.com.

Editor's Note: We Want to Hear from You!

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

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

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

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

E-mail: trina.macdonald@pearson.com

Mail: Trina MacDonald
Senior Acquisitions Editor
Addison-Wesley/Pearson Education, Inc.
75 Arlington St., Ste. 300
Boston, MA 02116

Gestures and Touches

The touch represents the heart of iOS interaction; it provides the core way that users communicate their intent to an application. Touches are not limited to button presses and keyboard interaction. You can design and build applications that work directly with users' gestures in meaningful ways. This chapter introduces direct manipulation interfaces that go far beyond prebuilt controls. You see how to create views that users can drag around the screen. You also discover how to distinguish and interpret gestures, which are a high-level touch abstraction, and gesture recognizer classes, which automatically detect common interaction styles like taps, swipes, and drags. By the time you finish reading this chapter, you'll have read about many different ways you can implement gesture control in your own applications.

Touches

Cocoa Touch implements direct manipulation in the simplest way possible. It sends touch events to the view you're working with. As an iOS developer, you tell the view how to respond. Before jumping into gestures and gesture recognizers, you should gain a solid foundation in this underlying touch technology. It provides the essential components of all touch-based interaction.

Each touch conveys information: where the touch took place (both the current and previous location), what phase of the touch was used (essentially mouse down, mouse moved, mouse up in the desktop application world, corresponding to finger or touch down, moved, and up in the direct manipulation world), a tap count (for example, single-tap/double-tap), and when the touch took place (through a time stamp).

iOS uses what is called a *responder chain* to decide which objects should process touches. As their name suggests, responders are objects that respond to events and they act as a chain of possible managers for those events. When the user touches the screen, the application looks for an object to handle this interaction. The touch is passed along, from view to view, until some object takes charge and responds to that event.

At the most basic level, touches and their information are stored in `UITouch` objects, which are passed as groups in `UIEvent` objects. Each object represents a single touch event, containing single or multiple touches. This depends both on how you've set up your application to respond (that is, if you've enabled Multi-Touch interaction), and how the user touches the screen (that is, the physical number of touch points).

Your application receives touches in view or view controller classes; both implement touch handlers via inheritance from the `UIResponder` class. You decide where you process and respond to touches. Trying to implement low-level gesture control in nonresponder classes has tripped up many new iOS developers.

Handling touches in views may seem counterintuitive. You probably expect to separate the way an interface looks (its view) from the way it responds to touches (its controller). Further, using views for direct touch interaction may seem to contradict Model-View-Controller design orthogonality, but it can be necessary and help promote encapsulation.

Consider the case of working with multiple touch-responsive subviews such as game pieces on a board. Building interaction behavior directly into view classes allows you to send meaningful semantically rich feedback to your main application while hiding implementation minutia. For example, you can inform your model that a pawn has moved to Queen's Bishop 5 at the end of an interaction sequence rather than transmit a meaningless series of vector changes. By hiding the way the game pieces move in response to touches, your model code can focus on game semantics instead of view position updates.

Drawing presents another reason to work in the `UIView` class. When your application handles any kind of drawing operation in response to user touches, you need to implement touch handlers in views. Unlike views, view controllers don't implement the all-important `drawRect:` method needed for providing custom presentations.

Working at the view controller level also has its perks. Instead of pulling out primary handling behavior into a secondary class implementation, adding touch management directly to the view controller allows you to interpret standard gestures, such as tap-and-hold or swipes, where those gestures have meaning. This better centralizes your code and helps tie controller interactions directly to your application model.

In the following sections and recipes, you discover how touches work, how you can incorporate them into your apps, and how you connect what a user sees with how that user interacts with the screen.

Phases

Touches have life cycles. Each touch can pass through any of five phases that represent the progress of the touch within an interface. These phases are as follows:

- **`UITouchPhaseBegan`**—Starts when the user touches the screen.
- **`UITouchPhaseMoved`**—Means a touch has moved on the screen.

- **UITouchPhaseStationary**—Indicates that a touch remains on the screen surface but that there has not been any movement since the previous event.
- **UITouchPhaseEnded**—Gets triggered when the touch is pulled away from the screen.
- **UITouchPhaseCancelled**—Occurs when the iOS system stops tracking a particular touch. This usually occurs due to a system interruption, such as when the application is no longer active or the view is removed from the window.

Taken as a whole, these five phases form the interaction language for a touch event. They describe all the possible ways that a touch can progress or fail to progress within an interface and provide the basis for control for that interface. It's up to you as the developer to interpret those phases and provide reactions to them. You do that by implementing a series of responder methods.

Touches and Responder Methods

All subclasses of the `UIResponder` class, including `UIView` and `UIViewController`, respond to touches. Each class decides whether and how to respond. When choosing to do so, they implement customized behavior when a user touches one or more fingers down in a view or window.

Predefined callback methods handle the start, movement, and release of touches from the screen. Corresponding to the phases you've already seen, the methods involved are as follows. Notice that `UITouchPhaseStationary` does not generate a callback.

- **touchesBegan:withEvent:**—Gets called at the starting phase of the event, as the user starts touching the screen.
- **touchesMoved:withEvent:**—Handles the movement of the fingers over time.
- **touchesEnded:withEvent:**—Concludes the touch process, where the finger or fingers are released. It provides an opportune time to clean up any work that was handled during the movement sequence.
- **touchesCancelled:WithEvent:**—Called when Cocoa Touch must respond to a system interruption of the ongoing touch event.

Each of these is a `UIResponder` method, often implemented in a `UIView` or `UIViewController` subclass. All views inherit basic nonfunctional versions of the methods. When you want to add touch behavior to your application, you override these methods and add a custom version that provides the responses your application needs.

Your classes can implement all or just some of these methods. For real-world deployment, you will always want to add a `touchesCancelled` event to handle the case of a user dragging his or her finger offscreen or the case of an incoming phone call, both of which cancel an ongoing touch sequence. As a rule, you can generally redirect a canceled touch to your `touchesEnded:withEvent:` method. This allows your code to complete the touch sequence, even if the user's finger has not left the screen. Apple recommends overriding all four methods as a best practice when working with touches.

Note

Views have a mode called *exclusive touch* that prevents touches from being delivered to other views in the same window. When enabled, this property blocks other views from receiving touch events. The primary view handles all touch events exclusively.

Touching Views

When dealing with many onscreen views, iOS automatically decides which view the user touched and passes any touch events to the proper view for you. This helps you write concrete direct manipulation interfaces where users touch, drag, and interact with onscreen objects.

Just because a touch is physically on top of a view doesn't mean that a view has to respond. Each view can use a "hit test" to choose whether to handle a touch or to let that touch fall through to views beneath it. As you see in the recipes that follow, you can use clever response strategies to decide when your view should respond, particularly when you're using irregular art with partial transparency.

With touch events, the first view that passes the hit test opts to handle or deny the touch. If it passes, the touch continues to the view's superview and then works its way up the responder chain until it is handled or until it reaches the window that owns the views. If the window does not process it, the touch moves to the application instance, where it is either processed or discarded.

Note

Touches are limited to `UIView`s and their subclasses. This includes windows. When developing apps targeted at both iPhone 4S-and-earlier and iPhone 5 platforms, make sure your window extends across the entire screen. Problems are most commonly seen when the iPhone 5 is in portrait mode and apps fail to respond to touches at the bottom of the screen. This occurs when the application's key window is sized to a 3.5" screen but used on the larger 4" model. Without a backing window view, touches will not be recognized. Make sure your apps support both 3.5" and 4" screens by extending their key `UIWindow` to the screen's full vertical extent.

Multi-Touch

iOS supports both single- and Multi-Touch interfaces. Single-touch GUIs handle just one touch at any time. This relieves you of any responsibility to determine which touch you were tracking. The one touch you receive is the only one you need to work with. You look at its data, respond to it, and wait for the next event.

When working with Multi-Touch—that is, when you respond to multiple onscreen touches at once—you receive an entire set of touches. It is up to you to order and respond to that set. You can, however, track each touch separately and see how it changes over time, providing a richer set of possible user interaction. Recipes for both single-touch and Multi-Touch interaction follow in this chapter.

Gesture Recognizers

With gesture recognizers, Apple added a powerful way to detect specific gestures in your interface. Gesture recognizers simplify touch design. They encapsulate touch methods, so you don't have to implement these yourself, and provide a target-action feedback mechanism that hides implementation details. They also standardize how certain movements are categorized, as drags or swipes, and so forth.

With gesture recognizer classes, you can trigger callbacks when iOS perceives that the user has tapped, pinched, rotated, swiped, panned, or used a long press. Although their software development kit (SDK) implementations remain imperfect, these detection capabilities simplify development of touch-based interfaces. You can code your own for improved reliability, but a majority of developers will find that the recognizers, as-shipped, are robust enough for many application needs. You'll find several recognizer-based recipes in this chapter. Because recognizers all basically work in the same fashion, you can easily extend these recipes to your specific gesture recognition requirements.

Here is a rundown of the kinds of gestures built in to recent versions of the iOS SDK:

- **Taps**—Taps correspond to single or multiple finger taps onscreen. Users can tap with one or more fingers; you specify how many fingers you require as a gesture recognizer property and how many taps you want to detect. You can create a tap recognizer that works with single finger taps, or more nuanced recognizers that look for, for example, two-fingered triple-taps.
- **Swipes**—Swipes are short, single- or Multi-Touch gestures that move in a single cardinal direction: up, down, left, or right. They cannot move too far off course from that primary direction. You set the direction you want your recognizer to work with. The recognizer returns the detected direction as a property.
- **Pinches**—To pinch or unpinch, a user must move two fingers together or apart in a single movement. The recognizer returns a scale factor indicating the degree of pinching.
- **Rotations**—To rotate, a user moves two fingers at once either in a clockwise or counterclockwise direction, producing an angular rotation as the main returned property.
- **Pan**—Pans occur when users drag their fingers across the screen. The recognizer determines the change in translation produced by that drag.
- **Long press**—To create a long press, the user touches the screen and holds his or her finger (or fingers) there for a specified period of time. You can specify how many fingers must be used before the recognizer triggers.

Recipe: Adding a Simple Direct Manipulation Interface

Your design focus moves from the `UIViewController` to the `UIView` when you work with direct manipulation. The view, or more precisely the `UIResponder`, forms the heart of direct

manipulation development. You create touch-based interfaces by customizing methods that derive from the `UIResponder` class.

Recipe 1-1 centers on touches in action. This example creates a child of `UIImageView` called `DragView` and adds touch responsiveness to the class. Being an image view, it's important to enable user interaction (that is, set `setUserInteractionEnabled` to `YES`). This property affects all the view's children as well as the view itself. User interaction is generally enabled for most views, but `UIImageView` is the one exception that stumps most beginners; Apple apparently didn't think people would generally manipulate them.

The recipe works by updating a view's center to match the movement of an onscreen touch. When a user first touches any `DragView`, the object stores the start location as an offset from the view's origin. As the user drags, the view moves along with the finger—always maintaining the same origin offset so that the movement feels natural. Movement occurs by updating the object's center. Recipe 1-1 calculates x and y offsets and adjusts the view center by those offsets after each touch movement.

Upon being touched, the view pops to the front. That's due to a call in the `touchesBegan:withEvent:` method. The code tells the superview that owns the `DragView` to bring that view to the front. This allows the active element to always appear foremost in the interface.

This recipe does not implement `touches-ended` or `touches-cancelled` methods. Its interests lie only in the movement of onscreen objects. When the user stops interacting with the screen, the class has no further work to do.

Recipe 1-1 Creating a Draggable View

```
@interface DragView : UIImageView
{
    CGPoint startLocation;
}
@end

@implementation DragView
- (id) initWithImage: (UIImage *) anImage
{
    if (self = [super initWithImage:anImage])
        self.userInteractionEnabled = YES;
    return self;
}

- (void) touchesBegan:(NSSet*)touches withEvent:(UIEvent*)event
{
    // Calculate and store offset, and pop view into front if needed
    startLocation = [[touches anyObject] locationInView:self];
    [self.superview bringSubviewToFront:self];
}
}
```

```
- (void) touchesMoved:(NSSet*)touches withEvent:(UIEvent*)event
{
    // Calculate offset
    CGPoint pt = [[touches anyObject] locationInView:self];
    float dx = pt.x - startLocation.x;
    float dy = pt.y - startLocation.y;
    CGPoint newcenter = CGPointMake(
        self.center.x + dx,
        self.center.y + dy);

    // Set new location
    self.center = newcenter;
}
@end
```

Get This Recipe's Code

To find this recipe's full sample project, point your browser to <https://github.com/erica/iOS-6-Cookbook> and go to the folder for Chapter 1.

Recipe: Adding Pan Gesture Recognizers

With gesture recognizers, you can achieve the same kind of interaction shown in Recipe 1-1 without working quite so directly with touch handlers. Pan gesture recognizers detect dragging gestures. They allow you to assign a callback that triggers whenever iOS senses panning.

Recipe 1-2 mimics Recipe 1-1's behavior by adding a recognizer to the view when it is first initialized. As iOS detects the user dragging on a `DragView` instance, the `handlePan:` callback updates the view's center to match the distance dragged.

This code uses what might seem like an odd way of calculating distance. It stores the original view location in an instance variable (`previousLocation`) and then calculates the offset from that point each time the view updates with a pan detection callback. This allows you to use affine transforms or apply the `setTranslation:inView:` method; you normally do not move view centers, as done here. This recipe creates a dx/dy offset pair and applies that offset to the view's center, changing the view's actual frame.

Unlike simple offsets, affine transforms allow you to meaningfully work with rotation, scaling, and translation all at once. To support transforms, gesture recognizers provide their coordinate changes in absolute terms rather than relative ones. Instead of issuing iterative offset vectors, the `UIPanGestureRecognizer` returns a single vector representing a translation in terms of some view's coordinate system, typically the coordinate system of the manipulated view's superview. This vector translation lends itself to simple affine transform calculations and can be mathematically combined with other changes to produce a unified transform representing all changes applied simultaneously.

Here's what the `handlePan:` method looks like using straight transforms and no stored state:

```
- (void) handlePan: (UIPanGestureRecognizer *) uigr
{
    if (uigr.state == UIGestureRecognizerStateEnded)
    {
        CGPoint newCenter = CGPointMake(
            self.center.x + self.transform.tx,
            self.center.y + self.transform.ty);
        self.center = newCenter;

        CGAffineTransform theTransform = self.transform;
        theTransform.tx = 0.0f;
        theTransform.ty = 0.0f;
        self.transform = theTransform;

        return;
    }

    CGPoint translation = [uigr translationInView:self.superview];
    CGAffineTransform theTransform = self.transform;
    theTransform.tx = translation.x;
    theTransform.ty = translation.y;
    self.transform = theTransform;
}
```

Notice how the recognizer checks for the end of interaction and then updates the view's position and resets the transform's translation. This adaptation requires no local storage and would eliminate the need for a `touchesBegan:withEvent:` method. Without these modifications, Recipe 1-2 has to store previous state.

Recipe 1-2 Using a Pan Gesture Recognizer to Drag Views

```
@interface DragView : UIImageView
{
    CGPoint previousLocation;
}
@end

@implementation DragView
- (id) initWithImage: (UIImage *) anImage
{
    if (self = [super initWithImage:anImage])
    {
        self.userInteractionEnabled = YES;
        UIPanGestureRecognizer *panRecognizer =
            [[UIPanGestureRecognizer alloc]
```

```
        initWithTarget:self action:@selector(handlePan:));
        self.gestureRecognizers = @[panRecognizer];
    }
    return self;
}

- (void) touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    // Promote the touched view
    [self.superview bringSubviewToFront:self];

    // Remember original location
    previousLocation = self.center;
}

- (void) handlePan: (UIPanGestureRecognizer *) uigr
{
    CGPoint translation = [uigr translationInView:self.superview];
    self.center = CGPointMake(previousLocation.x + translation.x,
                              previousLocation.y + translation.y);
}
@end
```

Get This Recipe's Code

To find this recipe's full sample project, point your browser to <https://github.com/erica/iOS-6-Cookbook> and go to the folder for Chapter 1.

Recipe: Using Multiple Gesture Recognizers Simultaneously

Recipe 1-3 builds off the ideas presented in Recipe 1-2, but with several differences. First, it introduces multiple recognizers that work in parallel. To achieve this, the code uses three separate recognizers—rotation, pinch, and pan—and adds them all to the `DragView`'s `gestureRecognizers` property. It assigns the `DragView` as the delegate for each recognizer. This allows the `DragView` to implement the `gestureRecognizer:shouldRecognizeSimultaneouslyWithGestureRecognizer:delegate` method, enabling these recognizers to work simultaneously. Until this method is added to return `YES` as its value, only one recognizer will take charge at a time. Using parallel recognizers allows you to, for example, both zoom and rotate in response to a user's pinch gesture.

Note

UITouch objects store an array of gesture recognizers. The items in this array represent each recognizer that receives the touch object in question. When a view is created without gesture recognizers, its responder methods will be passed touches with empty recognizer arrays.

Recipe 1-3 extends the view's state to include scale and rotation instance variables. These items keep track of previous transformation values and permit the code to build compound affine transforms. These compound transforms, which are established in Recipe 1-3's `updateTransformWithOffset:` method, combine translation, rotation, and scaling into a single result. Unlike the previous recipe, this recipe uses transforms uniformly to apply changes to its objects, which is the standard practice for recognizers.

Finally, this recipe introduces a hybrid approach to gesture recognition. Instead of adding a `UITapGestureRecognizer` to the view's recognizer array, Recipe 1-3 demonstrates how you can add the kind of basic touch method used in Recipe 1-1 to catch a triple-tap. In this example, a triple-tap resets the view back to the identity transform. This undoes any manipulation previously applied to the view and reverts it to its original position, orientation, and size. As you can see, the touches began, moved, ended, and cancelled methods work seamlessly alongside the gesture recognizer callbacks, which is the point of including this extra detail in this recipe. Adding a tap recognizer would have worked just as well.

This recipe demonstrates the conciseness of using gesture recognizers to interact with touches.

Recipe 1-3 Recognizing Gestures in Parallel

```
@interface DragView : UIImageView <UIGestureRecognizerDelegate>
{
    CGFloat tx; // x translation
    CGFloat ty; // y translation
    CGFloat scale; // zoom scale
    CGFloat theta; // rotation angle
}
@end

@implementation DragView
- (void) touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    // Promote the touched view
    [self.superview bringSubviewToFront:self];

    // initialize translation offsets
    tx = self.transform.tx;
    ty = self.transform.ty;
}
}
```

```

- (void) touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    if (touch.tapCount == 3)
    {
        // Reset geometry upon triple-tap
        self.transform = CGAffineTransformIdentity;
        tx = 0.0f; ty = 0.0f; scale = 1.0f; theta = 0.0f;
    }
}

- (void) touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event
{
    [self touchesEnded:touches withEvent:event];
}

- (void) updateTransformWithOffset:(CGPoint) translation
{
    // Create a blended transform representing translation,
    // rotation, and scaling
    self.transform = CGAffineTransformMakeTranslation(
        translation.x + tx, translation.y + ty);
    self.transform = CGAffineTransformRotate(self.transform, theta);
    self.transform = CGAffineTransformScale(self.transform, scale, scale);
}

- (void) handlePan: (UIPanGestureRecognizer *) uigr
{
    CGPoint translation = [uigr translationInView:self.superview];
    [self updateTransformWithOffset:translation];
}

- (void) handleRotation: (UIRotationGestureRecognizer *) uigr
{
    theta = uigr.rotation;
    [self updateTransformWithOffset:CGPointZero];
}

- (void) handlePinch: (UIPinchGestureRecognizer *) uigr
{
    scale = uigr.scale;
    [self updateTransformWithOffset:CGPointZero];
}

- (BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer
    shouldRecognizeSimultaneouslyWithGestureRecognizer:
        (UIGestureRecognizer *)otherGestureRecognizer

```

```

{
    return YES;
}

- (id) initWithImage:(UIImage *)image
{
    // Initialize and set as touchable
    if (!(self = [super initWithImage:image])) return nil;

    self.userInteractionEnabled = YES;

    // Reset geometry to identities
    self.transform = CGAffineTransformIdentity;
    tx = 0.0f; ty = 0.0f; scale = 1.0f; theta = 0.0f;

    // Add gesture recognizer suite
    UIRotationGestureRecognizer *rot = [[UIRotationGestureRecognizer alloc]
        initWithTarget:self action:@selector(handleRotation:)];
    UIPinchGestureRecognizer *pinch = [[UIPinchGestureRecognizer alloc]
        initWithTarget:self action:@selector(handlePinch:)];
    UIPanGestureRecognizer *pan = [[UIPanGestureRecognizer alloc]
        initWithTarget:self action:@selector(handlePan:)];
    self.gestureRecognizers = @[rot, pinch, pan];
    for (UIGestureRecognizer *recognizer in self.gestureRecognizers)
        recognizer.delegate = self;

    return self;
}
@end

```

Resolving Gesture Conflicts

Gesture conflicts may arise when you need to recognize several types of gestures at the same time. For example, what happens when you need to recognize both single- and double-taps? Should the single-tap recognizer fire at the first tap, even when the user intends to enter a double-tap? Or should you wait and respond only after it's clear that the user isn't about to add a second tap? The iOS SDK allows you to take these conflicts into account in your code.

Your classes can specify that one gesture must fail in order for another to succeed. Accomplish this by calling `requireGestureRecognizerToFail:`. This is a gesture method that takes one argument, another gesture recognizer. This call creates a dependency between the object receiving this message and another gesture object. What it means is this: For the first gesture to trigger, the second gesture must fail. If the second gesture is recognized, the first gesture will not be.

In real life, this typically means that the recognizer adds a delay until it can be sure that the dependent recognizer has failed. It waits until the second gesture is no longer possible. Only then does the first recognizer complete. If you recognize both single- and double-taps, the application waits a little longer after the first tap. If no second tap happens, the single-tap fires. Otherwise, the double-tap fires, but not both.

Your GUI responses will slow down to accommodate this change. Your single-tap responses become slightly laggy. That's because there's no way to tell if a second tap is coming until time elapses. You should never use both kinds of recognizers where instant responsiveness is critical to your user experience. Try, instead, to design around situations where that tap means "do something *now*" and avoid requiring both gestures for those modes.

Don't forget that you can add, remove, and disable gesture recognizers on-the-fly. A single-tap may take your interface to a place where it then makes sense to further distinguish between single- and double-taps. When leaving that mode, you could disable or remove the double-tap recognizer to regain better single-tap recognition. Tweaks like this limit interface slowdowns to where they're absolutely needed.

Get This Recipe's Code

To find this recipe's full sample project, point your browser to <https://github.com/erica/iOS-6-Cookbook> and go to the folder for Chapter 1.

Recipe: Constraining Movement

One problem with the simple approach of the earlier recipes in this chapter is that it's entirely possible to drag a view offscreen to the point where the user cannot see or easily recover it. Those recipes use unconstrained movement. There is no check to test whether the object remains in view and is touchable. Recipe 1-4 fixes this problem by constraining a view's movement to within its parent.

It achieves this by limiting movement in each direction, splitting its checks into separate x and y constraints. This two-check approach allows the view to continue to move even when one direction has passed its maximum. If the view has hit the rightmost edge of its parent, for example, it can still move up and down.

Figure 1-1 shows a sample interface. The subviews (flowers) are constrained into the black rectangle in the center of the interface and cannot be dragged off-view. Recipe 1-4's code is general and can adapt to parent bounds and child views of any size.

Recipe 1-4 Bounded Movement

```
- (void) handlePan: (UIPanGestureRecognizer *) uigr
{
    CGPoint translation = [uigr translationInView:self.superview];
    CGPoint newcenter = CGPointMake(
```

```
        previousLocation.x + translation.x,  
        previousLocation.y + translation.y);  
  
    // Restrict movement into parent bounds  
    float halfx = CGRectGetMidX(self.bounds);  
    newcenter.x = MAX(halfx, newcenter.x);  
    newcenter.x = MIN(self.superview.bounds.size.width - halfx,  
        newcenter.x);  
  
    float halfy = CGRectGetMidY(self.bounds);  
    newcenter.y = MAX(halfy, newcenter.y);  
    newcenter.y = MIN(self.superview.bounds.size.height - halfy,  
        newcenter.y);  
  
    // Set new location  
    self.center = newcenter;  
}
```

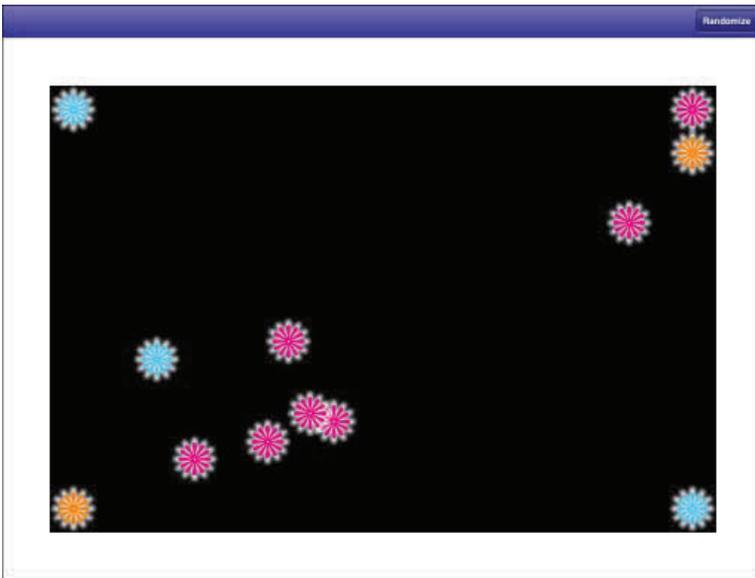


Figure 1-1 The movement of these flowers is bounded into the black rectangle.

Get This Recipe's Code

To find this recipe's full sample project, point your browser to <https://github.com/erica/iOS-6-Cookbook> and go to the folder for Chapter 1.

Recipe: Testing Touches

Most onscreen view elements for direct manipulation interfaces are not rectangular. This complicates touch detection because parts of the actual view rectangle may not correspond to actual touch points. Figure 1-2 shows the problem in action. The screenshot on the right shows the interface with its touch-based subviews. The shot on the left shows the actual view bounds for each subview. The light gray areas around each onscreen circle fall within the bounds, but touches to those areas should not “hit” the view in question.

iOS senses user taps throughout the entire view frame. This includes the undrawn area, such as the corners of the frame outside the actual circles of Figure 1-2, just as much as the primary presentation. That means that unless you add some sort of hit test, users may attempt to tap through to a view that’s “obscured” by the clear portion of the `UIView` frame.

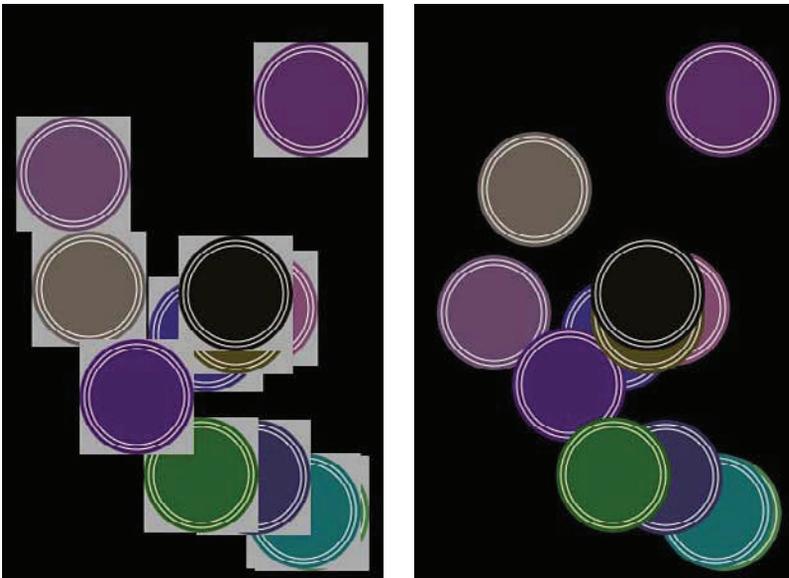


Figure 1-2 The application should ignore touches to the gray areas that surround each circle (left). The actual interface (right) uses a clear background (zero alpha values) to hide the parts of the view that are not used.

Visualize your actual view bounds by setting its background color, for example:

```
dragger.backgroundColor = [UIColor lightGrayColor];
```

This adds the backslashes shown in Figure 1-2 (left) without affecting the actual onscreen art. In this case, the art consists of a centered circle with a transparent background. Unless you add some sort of test, all taps to any portion of this frame are captured by the view in question. Enabling background colors offers a convenient debugging aid to visualize the true extent of

each view; don't forget to comment out the background color assignment in production code. Alternatively, you can set a view layer's border width or style.

Recipe 1-5 adds a simple hit test to the views, determining whether touches fall within the circle. This test overrides the standard `UIView`'s `pointInside:withEvent:` method. This method returns either `YES` (the point falls inside the view) or `NO` (it does not). The test here uses basic geometry, checking whether the touch lies within the circle's radius. You can provide any test that works with your onscreen views. As you see in Recipe 1-6, which follows in the next section, that test can be expanded for much finer control.

Be aware that the math for touch detection on Retina display devices remains the same as that for older units. The extra onboard pixels do not affect your gesture-handling math. Your view's coordinate system remains floating point with subpixel accuracy. The number of pixels the device uses to draw to the screen does not affect `UIView` bounds and `UITouch` coordinates. It simply provides a way to provide higher detail graphics within that coordinate system.

Note

Do not confuse the point inside test, which checks whether a point falls inside a view, with the similar-sounding `hitTest:withEvent:`. The hit test returns the topmost view (closest to the user/screen) in a view hierarchy that contains a specific point. It works by calling `pointInside:withEvent:` on each view. If the point inside method returns `YES`, the search continues down that hierarchy.

Recipe 1-5 Providing a Circular Hit Test

```
- (BOOL) pointInside:(CGPoint)point withEvent:(UIEvent *)event
{
    CGPoint pt;
    float HALFSIDE = SIDELENGTH / 2.0f;

    // normalize with centered origin
    pt.x = (point.x - HALFSIDE) / HALFSIDE;
    pt.y = (point.y - HALFSIDE) / HALFSIDE;

    // x^2 + y^2 = radius^2
    float xsquared = pt.x * pt.x;
    float ysquared = pt.y * pt.y;

    // If the radius <= 1, the point is within the clipped circle
    if ((xsquared + ysquared) <= 1.0) return YES;
    return NO;
}
```

Get This Recipe's Code

To find this recipe's full sample project, point your browser to <https://github.com/erica/iOS-6-Cookbook> and go to the folder for Chapter 1.

Recipe: Testing Against a Bitmap

Unfortunately, most views don't fall into the simple geometries that make the hit test from Recipe 1-5 so straightforward. The flowers shown in Figure 1-1, for example, offer irregular boundaries and varied transparencies. For complicated art, it helps to test touches against a bitmap. Bitmaps provide byte-by-byte information about the contents of an image-based view, allowing you to test whether a touch hits a solid portion of the image or should pass through to any views below.

Recipe 1-6 extracts an image bitmap from a `UIImageView`. It assumes that the image used provides a pixel-by-pixel representation of the view in question. When you distort that view (normally by resizing a frame or applying a transform), update the math accordingly. `CGPoint`s can be transformed via `CGPointApplyAffineTransform()` to handle scaling and rotation changes. Keeping the art at a 1:1 proportion to the actual view pixels simplifies lookup and avoids any messy math. You can recover the pixel in question, test its alpha level, and determine whether the touch has hit a solid portion of the view.

This example uses a cutoff of 85. That corresponds to a minimum alpha level of 33% (that is, $85 / 255$). This custom `pointInside:` method considers any pixel with an alpha level below 33% to be transparent. This is arbitrary. Use any level (or other test for that matter) that works with the demands of your actual GUI.

Note

Unless you need pixel-perfect touch detection, you can probably scale down the bitmap so that it uses less memory and adjust the detection math accordingly.

Recipe 1-6 Testing Touches Against Bitmap Alpha Levels

```
// Return the offset for the alpha pixel at (x,y) for RGBA
// 4-bytes-per-pixel bitmap data
static NSUInteger alphaOffset(NSUInteger x, NSUInteger y, NSUInteger w)
    {return y * w * 4 + x * 4;}

// Return the bitmap from a provided image
NSData *getBitmapFromImage(UIImage *image)
{
    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    if (colorSpace == NULL)
    {
```

```

        fprintf(stderr, "Error allocating color space\n");
        return NULL;
    }

    CGSize size = image.size;
    unsigned char *bitmapData = calloc(size.width * size.height * 4, 1);
    if (bitmapData == NULL)
    {
        fprintf (stderr, "Error: Memory not allocated!");
        CGColorSpaceRelease(colorSpace);
        return NULL;
    }

    CGContextRef context = CGContextCreate (bitmapData,
        size.width, size.height, 8, size.width * 4, colorSpace,
        kCGImageAlphaPremultipliedFirst);
    CGColorSpaceRelease(colorSpace );
    if (context == NULL)
    {
        fprintf (stderr, "Error: Context not created!");
        free (bitmapData);
        return NULL;
    }

    CGRect rect = CGRectMake(0.0f, 0.0f, size.width, size.height);
    CGContextDrawImage(context, rect, image.CGImage);
    unsigned char *data = CGContextGetData(context);
    CGContextRelease(context);

    NSData *bytes = [NSData dataWithBytes:data length:size.width * size.height * 4];
    free(bitmapData);

    return bytes;
}

// Store the bitmap data into an NSData instance variable
- (id) initWithImage: (UIImage *) anImage
{
    if (self = [super initWithImage:anImage])
    {
        self.userInteractionEnabled = YES;
        data = getBitmapFromImage(anImage);
    }
    return self;
}

```

```
// Does the point hit the view?
- (BOOL) pointInside:(CGPoint)point withEvent:(UIEvent *)event
{
    if (!CGRectContainsPoint(self.bounds, point)) return NO;
    Byte *bytes = (Byte *)data.bytes;
    uint offset = alphaOffset(point.x, point.y, self.image.size.width);
    return (bytes[offset] > 85);
}
```

Get This Recipe's Code

To find this recipe's full sample project, point your browser to <https://github.com/erica/iOS-6-Cookbook> and go to the folder for Chapter 1.

Recipe: Drawing Touches Onscreen

UIView hosts the realm of direct onscreen drawing. Its `drawRect:` method offers a low-level way to draw content directly, letting you create and display arbitrary elements using Quartz 2D calls. Touch plus drawing join together to build concrete, manipulatable interfaces.

Recipe 1-7 combines gestures with `drawRect` to introduce touch-based painting. As a user touches the screen, the `TouchTrackerView` class builds a Bezier path that follows the user's finger. To paint the progress as the touch proceeds, the `touchesMoved:withEvent:` method calls `setNeedsDisplay`. This, in turn, triggers a call to `drawRect:`, where the view strokes the accumulated Bezier path. Figure 1-3 shows the interface with a path created in this way.



Figure 1-3 A simple painting tool for iOS requires little more than collecting touches along a path and painting that path with UIKit/Quartz 2D calls.

Although you could adapt this recipe to use gesture recognizers, there's really no point to it. The touches are essentially meaningless, only provided to create a pleasing tracing. The basic responder methods (namely touches began, moved, and so on) are perfectly capable of handling path creation and management tasks.

This example is meant for creating continuous traces. It does not respond to any touch event without a move. If you want to expand this recipe to add a simple dot or mark, you'll have to add that behavior yourself.

Recipe 1-7 Touch-Based Painting in a UIView

```
@interface TouchTrackerView : UIView
{
    UIBezierPath *path;
}
@end

@implementation TouchTrackerView
- (void) touchesBegan:(NSSet *) touches withEvent:(UIEvent *) event
{
    // Initialize a new path for the user gesture
    self.path = [UIBezierPath bezierPath];
    path.lineWidth = 4.0f;

    UITouch *touch = [touches anyObject];
    [path moveToPoint:[touch locationInView:self]];
}

- (void) touchesMoved:(NSSet *) touches withEvent:(UIEvent *) event
{
    // Add new points to the path
    UITouch *touch = [touches anyObject];
    [self.path addLineToPoint:[touch locationInView:self]];
    [self setNeedsDisplay];
}

- (void) touchesEnded:(NSSet *) touches withEvent:(UIEvent *) event
{
    UITouch *touch = [touches anyObject];
    [path addLineToPoint:[touch locationInView:self]];
    [self setNeedsDisplay];
}

- (void) touchesCancelled:(NSSet *) touches withEvent:(UIEvent *) event
{
    [self touchesEnded:touches withEvent:event];
}
}
```

```

- (void) drawRect:(CGRect)rect
{
    // Draw the path
    [path stroke];
}

- (id) initWithFrame:(CGRect)frame
{
    if (self = [super initWithFrame:frame])
        self.multipleTouchEnabled = NO;
    return self;
}
@end

```

Get This Recipe's Code

To find this recipe's full sample project, point your browser to <https://github.com/erica/iOS-6-Cookbook> and go to the folder for Chapter 1.

Recipe: Smoothing Drawings

Depending on the device in use and the amount of simultaneous processing involved, capturing user gestures may produce results that are rougher than desired. Touch events are often limited by CPU demands as well as by shaking hands. A smoothing algorithm can offset those limitations by interpolating between points. Figure 1-4 demonstrates the kind of angularity that derives from granular input and the smoothing that can be applied instead.

Catmull-Rom splines create continuous curves between key points. This algorithm ensures that each initial point you provide remains part of the final curve. The resulting path retains the original path's shape. You choose the number of interpolation points between each pair of reference points. The trade-off lies between processing power and greater smoothing. The more points you add, the more CPU resources you'll consume. As you can see when using the sample code that accompanies this chapter, a little smoothing goes a long way, even on newer devices. The latest iPad is so responsive that it's hard to draw a particularly jaggy line in the first place.

Recipe 1-8 demonstrates how to extract points from an existing Bezier path and then apply splining to create a smoothed result. Catmull-Rom uses four points at a time to calculate intermediate values between the second and third points, using a granularity you specify between those points.

Recipe 1-8 provides an example of just one kind of real-time geometric processing you might add to your applications. Many other algorithms out there in the world of computational geometry can be applied in a similar manner.

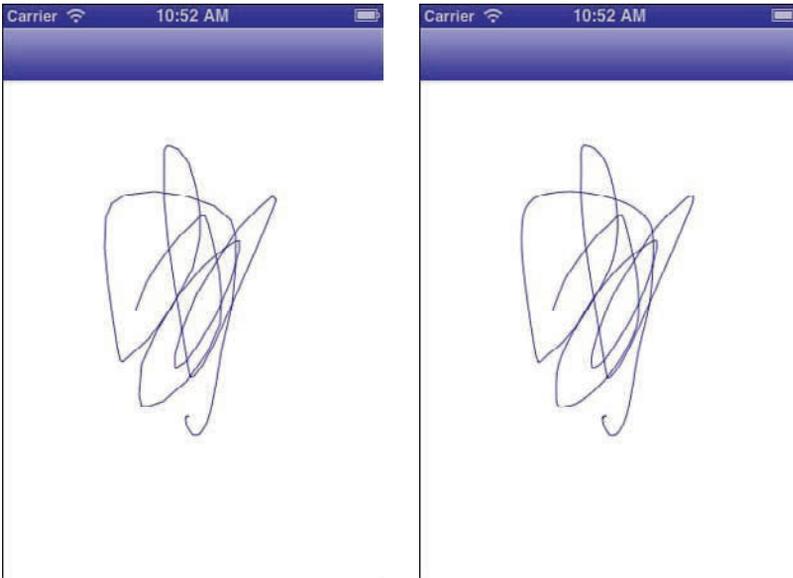


Figure 1-4 Catmull-Rom smoothing can be applied in real time to improve arcs between touch events. The images shown here are based on an identical gesture input, with and without smoothing applied.

Recipe 1-8 Creating Smoothed Bezier Paths Using Catmull-Rom Splining

```
#define VALUE(_INDEX_) [NSValue valueWithCGPoint:points[_INDEX_]]

@implementation UIBezierPath (Points)
void getPointsFromBezier(void *info, const CGPathElement *element)
{
    NSMutableArray *bezierPoints = (__bridge NSMutableArray *)info;

    // Retrieve the path element type and its points
    CGPathElementType type = element->type;
    CGPoint *points = element->points;

    // Add the points if they're available (per type)
    if (type != kCGPathElementCloseSubpath)
    {
        [bezierPoints addObject:VALUE(0)];
        if ((type != kCGPathElementAddLineToPoint) &&
            (type != kCGPathElementMoveToPoint))
            [bezierPoints addObject:VALUE(1)];
    }
}
```

```

    if (type == kCGPathElementAddCurveToPoint)
        [bezierPoints addObject:VALUE(2)];
}

- (NSArray *)points
{
    NSMutableArray *points = [NSMutableArray array];
    CGPathApply(self.CGPath, (__bridge void *)points, getPointsFromBezier);
    return points;
}
@end

#define POINT(_INDEX_) \
    [(NSValue *)[points objectAtIndex:_INDEX_] CGPointValue]

@implementation UIBezierPath (Smoothing)
- (UIBezierPath *) smoothedPath: (int) granularity
{
    NSMutableArray *points = [self.points mutableCopy];
    if (points.count < 4) return [self copy];

    // Add control points to make the math make sense
    // Via Josh Weinberg
    [points insertObject:[points objectAtIndex:0] atIndex:0];
    [points addObject:[points lastObject]];

    UIBezierPath *smoothedPath = [UIBezierPath bezierPath];

    // Copy traits
    smoothedPath.lineWidth = self.lineWidth;

    // Draw out the first 3 points (0..2)
    [smoothedPath moveToPoint:POINT(0)];

    for (int index = 1; index < 3; index++)
        [smoothedPath addLineToPoint:POINT(index)];

    for (int index = 4; index < points.count; index++)
    {
        CGPoint p0 = POINT(index - 3);
        CGPoint p1 = POINT(index - 2);
        CGPoint p2 = POINT(index - 1);
        CGPoint p3 = POINT(index);

        // now add n points starting at p1 + dx/dy up
        // until p2 using Catmull-Rom splines
        for (int i = 1; i < granularity; i++)

```

```

    {
        float t = (float) i * (1.0f / (float) granularity);
        float tt = t * t;
        float ttt = tt * t;

        CGPoint pi; // intermediate point
        pi.x = 0.5 * (2*p1.x+(p2.x-p0.x)*t +
            (2*p0.x-5*p1.x+4*p2.x-p3.x)*tt + (3*p1.x-p0.x-3*p2.x+p3.x)*ttt);
        pi.y = 0.5 * (2*p1.y+(p2.y-p0.y)*t +
            (2*p0.y-5*p1.y+4*p2.y-p3.y)*tt + (3*p1.y-p0.y-3*p2.y+p3.y)*ttt);
        [smoothedPath addLineToPoint:pi];
    }

    // Now add p2
    [smoothedPath addLineToPoint:p2];
}

// finish by adding the last point
[smoothedPath addLineToPoint:POINT(points.count - 1)];

return smoothedPath;
}

@end

// Example usage:
// Replace the path with a smoothed version after drawing completes
- (void) touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    UITouch *touch = [touches anyObject];
    [path addLineToPoint:[touch locationInView:self]];
    path = [path smoothedPath:4];
    [self setNeedsDisplay];
}

```

Get This Recipe's Code

To find this recipe's full sample project, point your browser to <https://github.com/erica/iOS-6-Cookbook> and go to the folder for Chapter 1.

Recipe: Using Multi-Touch Interaction

Enabling Multi-Touch interaction in `UIView` instances lets iOS recover and respond to more than one finger touch at a time. Set the `UIView` property `multipleTouchEnabled` to `YES` or override `isMultipleTouchEnabled` for your view. When enabled, each touch callback returns

an entire set of touches. When that set's count exceeds 1, you know you're dealing with Multi-Touch.

In theory, iOS supports an arbitrary number of touches. You can explore that limit by running the following recipe on an iPad, using as many fingers as possible at once. The practical upper limit has changed over time; this recipe modestly demurs from offering a specific number.

When Multi-Touch was first explored on the iPhone, developers did not dream of the freedom and flexibility that Multi-Touch combined with multiple users offered. Adding Multi-Touch to your games and other applications opens up not just expanded gestures but also new ways of creating profoundly exciting multiuser experiences, especially on larger screens like the iPad. I encourage you to include Multi-Touch support in your applications wherever it is practical and meaningful.

Multi-Touch touches are not grouped. If you touch the screen with two fingers from each hand, for example, there's no way to determine which touches belong to which hand. The touch order is also arbitrary. Although grouped touches retain the same finger order (or, more specifically, the same memory address) for the lifetime of a single touch event, from touch down through movement to release, the correspondence between touches and fingers may and likely will change the next time your user touches the screen. When you need to distinguish touches from each other, build a touch dictionary indexed by the touch objects, as shown in this recipe.

Perhaps it's a comfort to know that if you need it, the extra finger support has been built in. Unfortunately, when you are using three or more touches at a time, the screen has a pronounced tendency to lose track of one or more of those fingers. It's hard to programmatically track smooth gestures when you go beyond two finger touches. So instead of focusing on gesture interpretation, think of the Multi-Touch experience more as a series of time-limited independent interactions. You can treat each touch as a distinct item and process it independently of its fellows.

Recipe 1-9 adds Multi-Touch to a `UIView` by setting its `multipleTouchEnabled` property and tracing the lines that each finger draws. It does this by keeping track of each touch's physical address in memory but without pointing to or retaining the touch per Apple's recommendations.

This is, obviously, an oddball approach, but it has worked reliably throughout the history of the SDK. That's because each `UITouch` object persists at a single address throughout the touch-move-release life cycle. Apple recommends against retaining `UITouch` instances, which is why the integer values of these objects are used as keys in this recipe. By using the physical address as a key, you can distinguish each touch, even as new touches are added or old touches are removed from the screen.

Be aware that new touches can start their life cycle via `touchesBegan:withEvent:` independently of others as they move, end, or cancel. Your code should reflect that reality.

This recipe expands from Recipe 1-7. Each touch grows a separate Bezier path, which is painted in the view's `drawRect` method. Recipe 1-7 essentially started a new drawing at the end of each

touch cycle. That worked well for application bookkeeping but failed when it came to creating a standard drawing application, where you expect to iteratively add elements to a picture.

Recipe 1-9 continues adding traces into a composite picture without erasing old items. Touches collect into an ever-growing mutable array, which can be cleared on user demand. This recipe draws in-progress tracing in a slightly lighter color, to distinguish it from paths that have already been stored to the drawing's stroke array.

Recipe 1-9 Accumulating User Tracings for a Composite Drawing

```
@interface TouchTrackerView : UIView
{
    NSMutableArray *strokes;
    NSMutableDictionary *touchPaths;
}
- (void) clear;
@end

@implementation TouchTrackerView

// Establish new views with storage initialized for drawing
- (id) initWithFrame:(CGRect)frame
{
    if (self = [super initWithFrame:frame])
    {
        self.multipleTouchEnabled = YES;
        strokes = [NSMutableArray array];
        touchPaths = [NSMutableDictionary dictionary];
    }

    return self;
}

// On clear remove all existing strokes, but not in-progress drawing
- (void) clear
{
    [strokes removeAllObjects];
    [self setNeedsDisplay];
}

// Start touches by adding new paths to the touchPath dictionary
- (void) touchesBegan:(NSSet *) touches withEvent:(UIEvent *) event
{
    for (UITouch *touch in touches)
    {
        NSString *key = [NSString stringWithFormat:@"%d", (int) touch];
        CGPoint pt = [touch locationInView:self];
    }
}
```

```

    UIBezierPath *path = [UIBezierPath bezierPath];
    path.lineWidth = IS_IPAD? 8: 4;
    path.lineCapStyle = kCGLineCapRound;
    [path moveToPoint:pt];

    [touchPaths setObject:path forKey:key];
}
}
// Trace touch movement by growing and stroking the path
- (void) touchesMoved:(NSSet *) touches withEvent:(UIEvent *) event
{
    for (UITouch *touch in touches)
    {
        NSString *key =
            [NSString stringWithFormat:@"%d", (int) touch];
        UIBezierPath *path = [touchPaths objectForKey:key];
        if (!path) break;

        CGPoint pt = [touch locationInView:self];
        [path addLineToPoint:pt];
    }

    [self setNeedsDisplay];
}

// On ending a touch, move the path to the strokes array
- (void) touchesEnded:(NSSet *) touches withEvent:(UIEvent *) event
{
    for (UITouch *touch in touches)
    {
        NSString *key = [NSString stringWithFormat:@"%d", (int) touch];
        UIBezierPath *path = [touchPaths objectForKey:key];
        if (path) [strokes addObject:path];
        [touchPaths removeObjectForKey:key];
    }

    [self setNeedsDisplay];
}

- (void) touchesCancelled:(NSSet *) touches withEvent:(UIEvent *) event
{
    [self touchesEnded:touches withEvent:event];
}

// Draw existing strokes in dark purple, in-progress ones in light
- (void) drawRect:(CGRect)rect
{

```

```

[COOKBOOK_PURPLE_COLOR set];
for (UIBezierPath *path in strokes)
    [path stroke];

[[COOKBOOK_PURPLE_COLOR colorWithAlphaComponent:0.5f] set];
for (UIBezierPath *path in [touchPaths allValues])
    [path stroke];
}
@end

```

Get This Recipe's Code

To find this recipe's full sample project, point your browser to <https://github.com/erica/iOS-6-Cookbook> and go to the folder for Chapter 1.

Note

Apple provides many Core Graphics/Quartz 2D resources on its developer website. Although many of these forums, mailing lists, and source code examples are not iOS specific, they offer an invaluable resource for expanding your iOS Core Graphics knowledge.

Recipe: Detecting Circles

In a direct manipulation interface like iOS, you'd imagine that most people could get by just pointing to items onscreen. And yet, circle detection remains one of the most requested gestures. Developers like having people circle items onscreen with their fingers. In the spirit of providing solutions that readers have requested, Recipe 1-10 offers a relatively simple circle detector, which is shown in Figure 1-5.

In this implementation, detection uses a multistep test. A time test checks that the stroke was not lingering. A circle gesture should be quickly drawn. There's an inflection test checking that the touch did not change directions too often. A proper circle includes four direction changes. This test allows for five. There's a convergence test. The circle must start and end close enough together that the points are somehow related. A fair amount of leeway is needed because when you don't provide direct visual feedback, users tend to undershoot or overshoot where they began. The pixel distance used here is generous, approximately a third of the view size.

The final test looks at movement around a central point. It adds up the arcs traveled, which should equal 360 degrees in a perfect circle. This example allows any movement that falls within 45 degrees for not-quite-finished circles and 180 degrees for circles that continue on a bit wider, allowing the finger to travel more naturally.

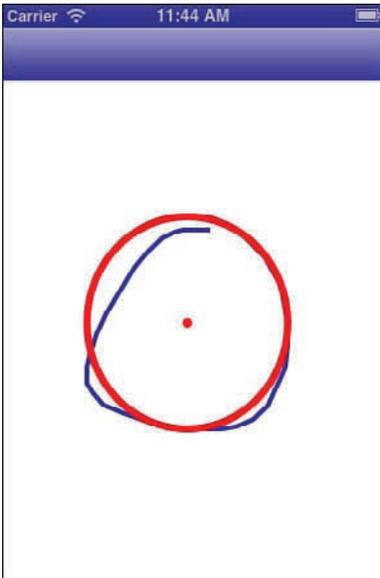


Figure 1-5 The dot and the outer ellipse show the key features of the detected circle.

Upon these tests being passed, the algorithm produces a least bounding rectangle and centers that rectangle on the geometric mean of the points from the original gesture. This result is assigned to the circle instance variable. It's not a perfect detection system (you can try to fool it when testing the sample code), but it's robust enough to provide reasonably good circle checks for many iOS applications.

Recipe 1-10 Detecting Circles

```
// Retrieve center of rectangle
CGPoint GEORectGetCenter(CGRect rect)
{
    return CGPointMake(CGRectGetMidX(rect), CGRectGetMidY(rect));
}

// Build rectangle around a given center
CGRect GEORectAroundCenter(CGPoint center, float dx, float dy)
{
    return CGRectMake(center.x - dx, center.y - dy, dx * 2, dy * 2);
}

// Center one rect inside another
CGRect GEORectCenteredInRect(CGRect rect, CGRect mainRect)
```

```

{
    CGFloat dx = CGRectGetMidX(mainRect)-CGRectGetMidX(rect);
    CGFloat dy = CGRectGetMidY(mainRect)-CGRectGetMidY(rect);
    return CGRectOffset(rect, dx, dy);
}

// Return dot product of two vectors normalized
CGFloat dotproduct (CGPoint v1, CGPoint v2)
{
    CGFloat dot = (v1.x * v2.x) + (v1.y * v2.y);
    CGFloat a = ABS(sqrt(v1.x * v1.x + v1.y * v1.y));
    CGFloat b = ABS(sqrt(v2.x * v2.x + v2.y * v2.y));
    dot /= (a * b);

    return dot;
}

// Return distance between two points
CGFloat distance (CGPoint p1, CGPoint p2)
{
    CGFloat dx = p2.x - p1.x;
    CGFloat dy = p2.y - p1.y;

    return sqrt(dx*dx + dy*dy);
}

// Offset in X
CGFloat dx(CGPoint p1, CGPoint p2)
{
    return p2.x - p1.x;
}

// Offset in Y
CGFloat dy(CGPoint p1, CGPoint p2)
{
    return p2.y - p1.y;
}

// Sign of a number
NSInteger sign(CGFloat x)
{
    return (x < 0.0f) ? (-1) : 1;
}

// Return a point with respect to a given origin
CGPoint pointWithOrigin(CGPoint pt, CGPoint origin)
{

```

```

    return CGPointMake(pt.x - origin.x, pt.y - origin.y);
}

// Calculate and return least bounding rectangle
#define POINT(_INDEX_) [(NSValue *)[points \
    objectAtIndex:_INDEX_] CGPointValue]

CGRect boundingRect(NSArray *points)
{
    CGRect rect = CGRectZero;
    CGRect ptRect;

    for (int i = 0; i < points.count; i++)
    {
        CGPoint pt = POINT(i);
        ptRect = CGRectMake(pt.x, pt.y, 0.0f, 0.0f);
        rect = (CGRectEqualToRect(rect, CGRectZero) ?
            ptRect : CGRectUnion(rect, ptRect);
    }

    return rect;
}

CGRect testForCircle(NSArray *points, NSDate *firstTouchDate)
{
    if (points.count < 2)
    {
        NSLog(@"Too few points (2) for circle");
        return CGRectZero;
    }

    // Test 1: duration tolerance
    float duration = [[NSDate date]
        timeIntervalSinceDate:firstTouchDate];
    NSLog(@"Transit duration: %0.2f", duration);

    float maxDuration = 2.0f;
    if (duration > maxDuration)
    {
        NSLog(@"Excessive duration");
        return CGRectZero;
    }

    // Test 2: Direction changes should be limited to near 4
    int inflections = 0;
    for (int i = 2; i < (points.count - 1); i++)
    {

```

```

float deltax = dx(POINT(i), POINT(i-1));
float delty = dy(POINT(i), POINT(i-1));
float px = dx(POINT(i-1), POINT(i-2));
float py = dy(POINT(i-1), POINT(i-2));

if ((sign(deltax) != sign(px)) ||
    (sign(delty) != sign(py)))
    inflections++;
}

if (inflections > 5)
{
    NSLog(@"Excessive inflections");
    return CGRectZero;
}

// Test 3: Start and end points near each other
float tolerance = [[[UIApplication sharedApplication]
    keyWindow] bounds].size.width / 3.0f;
if (distance(POINT(0), POINT(points.count - 1)) > tolerance)
{
    NSLog(@"Start too far from end");
    return CGRectZero;
}

// Test 4: Count the distance traveled in degrees.
CGRect circle = boundingRect(points);
CGPoint center = GEORectGetCenter(circle);
float distance = ABS(acos(dotproduct(
    pointWithOrigin(POINT(0), center),
    pointWithOrigin(POINT(1), center))));
for (int i = 1; i < (points.count - 1); i++)
    distance += ABS(acos(dotproduct(
        pointWithOrigin(POINT(i), center),
        pointWithOrigin(POINT(i+1), center))));

float transitTolerance = distance - 2 * M_PI;

if (transitTolerance < 0.0f) // fell short of 2 PI
{
    if (transitTolerance < - (M_PI / 4.0f)) // under 45
    {
        NSLog(@"Transit too short");
        return CGRectZero;
    }
}
}

```

```
    if (transitTolerance > M_PI) // additional 180 degrees
    {
        NSLog(@"Transit too long ");
        return CGRectZero;
    }

    return circle;
}
@end
```

Get This Recipe's Code

To find this recipe's full sample project, point your browser to <https://github.com/erica/iOS-6-Cookbook> and go to the folder for Chapter 1.

Recipe: Creating a Custom Gesture Recognizer

It takes little work to transform the code shown in Recipe 1-10 into a custom recognizer, as introduced in Recipe 1-11. Subclassing `UIGestureRecognizer` enables you to build your own circle recognizer that you can add to views in your applications.

Start by importing `UIGestureRecognizerSubclass.h` into your new class. The file declares everything you need your recognizer subclass to override or customize. For each method you override, make sure to call the original version of the method by calling the superclass method before invoking your new code.

Gestures fall into two types: continuous and discrete. The circle recognizer is discrete. It either recognizes a circle or fails. Continuous gestures include pinches and pans, where recognizers send updates throughout their life cycle. Your recognizer generates updates by setting its `state` property.

Recognizers are basically state machines for fingertips. All recognizers start in the possible state (`UIGestureRecognizerStatePossible`), and then for continuous gestures pass through a series of changed states (`UIGestureRecognizerStateChanged`). Discrete recognizers either succeed in recognizing a gesture (`UIGestureRecognizerStateRecognized`) or fail (`UIGestureRecognizerStateFailed`), as demonstrated in Recipe 1-11. The recognizer sends actions to its target each time you update state *except* when the state is set to possible or failed.

The rather long comments you see in Recipe 1-11 belong to Apple, courtesy of the subclass header file. I've included them here because they help explain the roles of the key methods that override their superclass. The `reset` method returns the recognizer back to its quiescent state, allowing it to prepare itself for its next recognition challenge.

The touches began (and so on) methods are called at similar points as their `UIResponder` analogs, enabling you to perform your tests at the same touch life cycle points. This example

waits to check for success or failure until the touches ended callback, and uses the same `testForCircle` method defined in Recipe 1-10.

Note

As an overriding philosophy, gesture recognizers should fail as soon as possible. When they succeed, you should store information about the gesture in local properties. The circle gesture should save any detected circle so users know where the gesture occurred.

Recipe 1-11 Creating a Gesture Recognizer Subclass

```
#import <UIKit/UITapGestureRecognizerSubclass.h>
@implementation CircleRecognizer

// called automatically by the runtime after the gesture state has
// been set to UITapGestureRecognizerStateEnded any internal state
// should be reset to prepare for a new attempt to recognize the gesture
// after this is received all remaining active touches will be ignored
// (no further updates will be received for touches that had already
// begun but haven't ended)
- (void)reset
{
    [super reset];

    points = nil;
    firstTouchDate = nil;
    self.state = UITapGestureRecognizerStatePossible;
}

// mirror of the touch-delivery methods on UIResponder
// UITapGestureRecognizer aren't in the responder chain, but observe
// touches hit-tested to their view and their view's subviews
// UITapGestureRecognizer receive touches before the view to which
// the touch was hit-tested
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    [super touchesBegan:touches withEvent:event];

    if (touches.count > 1)
    {
        self.state = UITapGestureRecognizerStateFailed;
        return;
    }

    points = [NSMutableArray array];
    firstTouchDate = [NSDate date];
}
```

```

UITouch *touch = [touches anyObject];
[points addObject: [NSValue valueWithCGPoint:
    [touch locationInView:self.view]]];
}

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
    [super touchesMoved:touches withEvent:event];
    UITouch *touch = [touches anyObject];
    [points addObject: [NSValue valueWithCGPoint:
        [touch locationInView:self.view]]];
}

- (void) touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    [super touchesEnded:touches withEvent: event];
    BOOL detectionSuccess = !CGRectEqualToRect(CGRectZero,
        testForCircle(points, firstTouchDate));
    if (detectionSuccess)
        self.state = UIGestureRecognizerStateRecognized;
    else
        self.state = UIGestureRecognizerStateFailed;
}
@end

```

Get This Recipe's Code

To find this recipe's full sample project, point your browser to <https://github.com/erica/iOS-6-Cookbook> and go to the folder for Chapter 1.

Recipe: Dragging from a Scroll View

iOS's rich set of gesture recognizers doesn't always accomplish exactly what you're looking for. Here's an example. Imagine a horizontal scrolling view filled with image views, one next to another, so you can scroll left and right to see the entire collection. Now, imagine that you want to be able to drag items out of that view and add them to a space directly below the scrolling area. To do this, you need to recognize downward touches on those child views (that is, orthogonal to the scrolling direction).

This was the puzzle I encountered while trying to help developer Alex Hosgrove, who was trying to build an application roughly equivalent to a set of refrigerator magnet letters. Users could drag those letters down into a workspace and then play with and arrange the items they'd chosen. There were two challenges with this scenario. First, who owned each touch? Second, what happened after the downward touch was recognized?

Both the scroll view and its children own an interest in each touch. A downward gesture should generate new objects; a sideways gesture should pan the scroll view. Touches have to be shared to allow both the scroll view and its children to respond to user interactions. This problem can be solved using gesture delegates.

Gesture delegates allow you to add simultaneous recognition, so that two recognizers can operate at the same time. You add this behavior by declaring a protocol (`UIGestureRecognizerDelegate`) and adding a simple delegate method:

```
- (BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer
    shouldRecognizeSimultaneouslyWithGestureRecognizer:
        (UIGestureRecognizer *)otherGestureRecognizer
{
    return YES;
}
```

You cannot reassign gesture delegates for scroll views, so you must add this delegate override to the implementation for the scroll view's children.

The second question, converting a swipe into a drag, is addressed by thinking about the entire touch lifetime. Each touch that creates a new object starts as a directional drag but ends up as a pan once the new view is created. A pan recognizer works better here than a swipe recognizer, whose lifetime ends at the point of recognition.

To make this happen, Recipe 1-12 manually adds that directional-movement detection, outside of the built-in gesture detection. In the end, that working-outside-the-box approach provides a major coding win. That's because once the swipe has been detected, the underlying pan gesture recognizer continues to operate. This allows the user to keep moving the swiped object without having to raise his or her finger and retouch the object in question.

This implementation detects swipes that move down at least 16 vertical pixels without straying more than 8 pixels to either side. When this code detects a downward swipe, it adds a new `DragView` (the same class used earlier in this chapter) to the screen and allows it to follow the touch for the remainder of the pan gesture interaction.

At the point of recognition, the class marks itself as having handled the swipe (`gestureWasHandled`) and disables the scroll view for the duration of the panning event. This allows the child complete control over the ongoing pan gesture without the scroll view reacting to further touch movement.

Recipe 1-12 Dragging Items Out of Scroll Views

```
@implementation DragView
```

```
#define DX(p1, p2)    (p2.x - p1.x)
```

```
#define DY(p1, p2)    (p2.y - p1.y)
```

```
#define SWIPE_DRAG_MIN 16
```

```
#define DRAGLIMIT_MAX 8
```

```

// Categorize swipe types
typedef enum {
    TouchUnknown,
    TouchSwipeLeft,
    TouchSwipeRight,
    TouchSwipeUp,
    TouchSwipeDown,
} SwipeTypes;

@implementation PullView
// Create a new view with an embedded pan gesture recognizer
- (id) initWithImage: (UIImage *) anImage
{
    if (self = [super initWithImage:anImage])
    {
        self.userInteractionEnabled = YES;
        UIPanGestureRecognizer *pan =
            [[[UIPanGestureRecognizer alloc] initWithTarget:self
              action:@selector(handlePan:)] autorelease];
        pan.delegate = self;
        self.gestureRecognizers = @[pan];
    }

// Allow simultaneous recognition
- (BOOL)gestureRecognizer:(UIGestureRecognizer *)gestureRecognizer
    shouldRecognizeSimultaneouslyWithGestureRecognizer:
        (UIGestureRecognizer *)otherGestureRecognizer
{
    return YES;
}

// Handle pans by detecting swipes
- (void) handlePan: (UISwipeGestureRecognizer *) uigr
{
    // Only deal with scroll view superviews
    if (![self.superview isKindOfClass:[UIScrollView class]]) return;

    // Extract superviews
    UIView *supersuper = self.superview.superview;
    UIScrollView *scrollView = (UIScrollView *) self.superview;

    // Calculate location of touch
    CGPoint touchLocation = [uigr locationInView:supersuper];

    // Handle touch based on recognizer state

```

```

if(uigr.state == UIGestureRecognizerStateBegan)
{
    // Initialize recognizer
    gestureWasHandled = NO;
    pointCount = 1;
    startPoint = touchLocation;
}

if(uigr.state == UIGestureRecognizerStateChanged)
{
    pointCount++;

    // Calculate whether a swipe has occurred
    float dx = DX(touchLocation, startPoint);
    float dy = DY(touchLocation, startPoint);

    BOOL finished = YES;
    if ((dx > SWIPE_DRAG_MIN) && (ABS(dy) < DRAGLIMIT_MAX))
        touchtype = TouchSwipeLeft;
    else if ((-dx > SWIPE_DRAG_MIN) && (ABS(dy) < DRAGLIMIT_MAX))
        touchtype = TouchSwipeRight;
    else if ((dy > SWIPE_DRAG_MIN) && (ABS(dx) < DRAGLIMIT_MAX))
        touchtype = TouchSwipeUp;
    else if ((-dy > SWIPE_DRAG_MIN) && (ABS(dx) < DRAGLIMIT_MAX))
        touchtype = TouchSwipeDown;
    else
        finished = NO;

    // If unhandled and a downward swipe, produce a new draggable view
    if (!gestureWasHandled && finished &&
        (touchtype == TouchSwipeDown))
    {
        dv = [[DragView alloc] initWithImage:self.image];
        dv.center = touchLocation;
        [supersuper addSubview:dv];
        scrollView.scrollEnabled = NO;
        gestureWasHandled = YES;
    }
    else if (gestureWasHandled)
    {
        // allow continued dragging after detection
        dv.center = touchLocation;
    }
}

if(uigr.state == UIGestureRecognizerStateEnded)
{

```

```

        // ensure that the scroll view returns to scrollable
        if (gestureWasHandled)
            scrollView.scrollEnabled = YES;
    }
}
@end

```

Get This Recipe's Code

To find this recipe's full sample project, point your browser to <https://github.com/erica/iOS-6-Cookbook> and go to the folder for Chapter 1.

Recipe: Live Touch Feedback

Have you ever needed to record a demo for an iOS app? There's always compromise involved. Either you use an overhead camera and struggle with reflections and the user's hand blocking the screen or you use a tool like Reflection (<http://reflectionapp.com>) but you only get to see what's directly on the iOS device screen. These app recordings lack any indication of the user's touch and visual focus.

Recipe 1-13 offers a simple set of classes (called *TOUCHkit*) that provide a live touch feedback layer for demonstration use. With it, you can see both the screen that you're recording as well as the touches that create the interactions you're trying to present. It provides a way to compile your app for both normal and demonstration deployment. You don't change your core application to use it. It's designed to work as a single toggle, providing builds for each use.

To demonstrate this, the code shown in Recipe 1-13 is bundled in the sample code repository with a standard Apple demo. This shows how you can roll the kit into nearly any standard application.

Enabling Touch Feedback

You add touch feedback by switching on the TOUCHkit feature, without otherwise affecting your normal code. To enable TOUCHkit, you set a single flag, compile and use that build for demonstration, complete with touch overlay. For App Store deployment, you disable the flag. The application reverts to its normal behavior, and there are no App Store unsafe calls to worry about:

```
#define USES_TOUCHkit    1
```

This recipe assumes that you're using a standard application with a single primary window. When compiled in, the kit replaces that window with a custom class that captures and duplicates all touches, allowing your application to show the user's touch bubble feedback.

There is one key code-level change you must make, but it's a very small one. In your application delegate class, you define a `WINDOW_CLASS` to use when building your iOS screen:

```
#if USES_TOUCHkit
#import "TOUCHkitView.h"
#import "TOUCHOverlayWindow.h"
#define WINDOW_CLASS TOUCHOverlayWindow
#else
#define WINDOW_CLASS UIWindow
#endif
```

Then instead of declaring a `UIWindow`, you use whichever class has been set by the toggle:

```
WINDOW_CLASS *window;
window = [[WINDOW_CLASS alloc]
    initWithFrame:[UIScreen mainScreen] bounds];
```

From here, you can set the window's `rootViewController` as normal.

Intercepting and Forwarding Touch Events

The key to this overlay lies in intercepting touch events, creating a floating presentation above your normal interface, and then forwarding those events on to your application. A `TOUCHkit` view lies on top of your interface. The custom window class grabs user touch events and presents them as circles in the `TOUCHkit` view. It then forwards them as if the user were interacting with a normal `UIWindow`. To accomplish this, this recipe uses event forwarding.

Event forwarding is achieved by calling a secondary event handler. The `TOUCHOverlayWindow` class overrides `UIWindow`'s `sendEvent:` method to force touch drawing and then invokes its superclass implementation to return control to the normal responder chain.

The following implementation is drawn from Apple's Event Handling Guide for iOS. It collects all the touches associated with the current event, allowing Multi-Touch as well as single touch interactions, dispatches them to `TOUCHkit` view layer, and then redirects them to the window via the normal `UIWindow` `sendEvent:` implementation:

```
@implementation TOUCHOverlayWindow
- (void) sendEvent:(UIEvent *)event
{
    // Collect touches
    NSSet *touches = [event allTouches];
    NSMutableSet *began = nil;
    NSMutableSet *moved = nil;
    NSMutableSet *ended = nil;
    NSMutableSet *cancelled = nil;

    // Sort the touches by phase for event dispatch
    for(UITouch *touch in touches) {
        switch ([touch phase]) {
```

```

        case UITouchPhaseBegan:
            if (!began) began = [NSMutableSet set];
            [began addObject:touch];
            break;
        case UITouchPhaseMoved:
            if (!moved) moved = [NSMutableSet set];
            [moved addObject:touch];
            break;
        case UITouchPhaseEnded:
            if (!ended) ended = [NSMutableSet set];
            [ended addObject:touch];
            break;
        case UITouchPhaseCancelled:
            if (!cancelled) cancelled = [NSMutableSet set];
            [cancelled addObject:touch];
            break;
        default:
            break;
    }
}

// Create pseudo-event dispatch
if (began)
    [[TOUCHkitView sharedInstance]
     touchesBegan:began withEvent:event];
if (moved)
    [[TOUCHkitView sharedInstance]
     touchesMoved:moved withEvent:event];
if (ended)
    [[TOUCHkitView sharedInstance]
     touchesEnded:ended withEvent:event];
if (cancelled)
    [[TOUCHkitView sharedInstance]
     touchesCancelled:cancelled withEvent:event];

// Call normal handler for default responder chain
[super sendEvent: event];
}
@end

```

Implementing the TOUCHkit Overlay View

The TOUCHkit overlay is a single clear `UIView` singleton. It's created the first time the application requests its shared instance, and the call adds it to the application's key window. The overlay's user interaction flag is disabled, allowing touches to continue on through the responder

chain, even after processing those touches through the standard began/moved/ended/cancelled event callbacks.

The touch processing events draw a circle at each touch point, creating a strong pointer to the touches until that drawing is complete. Recipe 1-13 details the callback and drawing methods that handle that functionality.

Recipe 1-13 Creating a Touch Feedback Overlay View

```
+ (id) sharedInstance
{
    // Create shared instance if it does not yet exist
    if(!sharedInstance)
    {
        sharedInstance = [[self alloc] initWithFrame:CGRectZero];
    }

    // Parent it to the key window
    if (!sharedInstance.superview)
    {
        UIWindow *keyWindow= [UIApplication sharedApplication].keyWindow;
        sharedInstance.frame = keyWindow.bounds;
        [keyWindow addSubview:sharedInstance];
    }

    return sharedInstance;
}

// You can override the default touchColor if you want
- (id) initWithFrame:(CGRect)frame
{
    if (self = [super initWithFrame:frame])
    {
        self.backgroundColor = [UIColor clearColor];
        self.userInteractionEnabled = NO;
        self.multipleTouchEnabled = YES;
        touchColor =
            [[UIColor whiteColor] colorWithAlphaComponent:0.5f];
        touches = nil;
    }

    return self;
}

// Basic Touches processing
- (void) touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
```

```

    touches = theTouches;
    [self setNeedsDisplay];
}

- (void) touchesMoved:(NSSet *)theTouches withEvent:(UIEvent *)event
{
    touches = theTouches;
    [self setNeedsDisplay];
}

- (void) touchesEnded:(NSSet *)theTouches withEvent:(UIEvent *)event
{
    touches = nil;
    [self setNeedsDisplay];
}

// Draw touches interactively
- (void) drawRect:(CGRect) rect
{
    // Clear
    CGContextRef context = UIGraphicsGetCurrentContext();
    CGContextClearRect(context, self.bounds);

    // Fill see-through
    [[UIColor clearColor] set];
    CGContextFillRect(context, self.bounds);

    float size = 25.0f; // based on 44.0f standard touch point

    for (UITouch *touch in touches)
    {
        // Create a backing frame
        [[[UIColor darkGrayColor] colorWithAlphaComponent:0.5f] set];
        CGPoint aPoint = [touch locationInView:self];
        CGContextAddEllipseInRect(context,
            CGRectMake(aPoint.x - size, aPoint.y - size, 2 * size, 2 * size));
        CGContextFillPath(context);

        // Draw the foreground touch
        float dsize = 1.0f;
        [touchColor set];
        aPoint = [touch locationInView:self];
        CGContextAddEllipseInRect(context,
            CGRectMake(aPoint.x - size - dsize, aPoint.y - size - dsize,
                2 * (size - dsize), 2 * (size - dsize)));
        CGContextFillPath(context);
    }
}

```

```
// Reset touches after use  
touches = nil;  
}
```

Get This Recipe's Code

To find this recipe's full sample project, point your browser to <https://github.com/erica/iOS-6-Cookbook> and go to the folder for Chapter 1.

Recipe: Adding Menus to Views

The `UIMenuController` class allows you to add pop-up menus to any item that acts as a first responder. Normally menus are used with text views and text fields, enabling users to select, copy, and paste. Menus also provide a way to add actions to interactive elements like the small drag views used throughout this chapter. Figure 1-6 shows a customized menu. In Recipe 1-14, this menu is presented after long-tapping a flower. The actions will zoom, rotate, or hide the associated drag view.

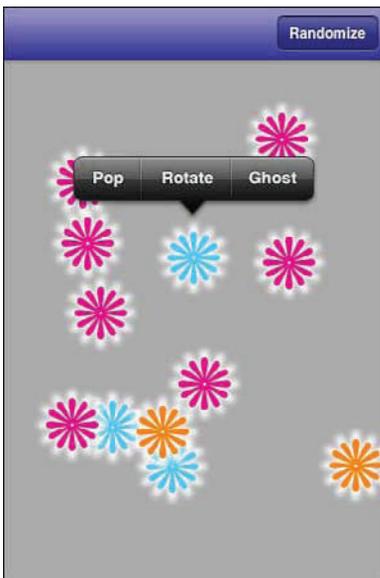


Figure 1-6 Contextual pop-up menus allow you to add interactive actions to first responder views.

This recipe demonstrates how to retrieve the shared menu controller and assign items to it. Set the menu's target rectangle (typically the bounds of the view that presents it), adjust the menu's arrow direction, and update the menu with your changes. The menu can now be set visible.

Menu items work with standard target-action callbacks, but you do not assign the target directly. Their target is always the first responder view. This recipe omits a `canPerformAction:withSender:` responder check, but you'll want to add that if some views support certain actions and other views do not. With menus, that support is often tied to state. For example, you don't want to offer a copy command if the view has no content to copy.

Recipe 1-14 Adding Menus to Interactive Views

```
- (BOOL) canBecomeFirstResponder
{
    // Menu only work with first responders
    return YES;
}

- (void) pressed: (UILongPressGestureRecognizer *) recognizer
{
    if (![self becomeFirstResponder])
    {
        NSLog(@"Could not become first responder");
        return;
    }

    UIMenuController *menu = [UIMenuController sharedMenuController];
    UIMenuItem *pop = [[UIMenuItem alloc]
        initWithTitle:@"Pop" action:@selector(popSelf)];
    UIMenuItem *rotate = [[UIMenuItem alloc]
        initWithTitle:@"Rotate" action:@selector(rotateSelf)];
    UIMenuItem *ghost = [[UIMenuItem alloc]
        initWithTitle:@"Ghost" action:@selector(ghostSelf)];
    [menu setMenuItems:@[pop, rotate, ghost]];

    [menu setTargetRect:self.bounds inView:self];
    menu.arrowDirection = UIMenuControllerArrowDown;
    [menu update];

    [menu setMenuVisible:YES];
}
```

```

- (id) initWithImage: (UIImage *) anImage
{
    if (!(self = [super initWithImage:anImage])) return nil;

    self.userInteractionEnabled = YES;
    UILongPressGestureRecognizer *pressRecognizer =
        [[UILongPressGestureRecognizer alloc] initWithTarget:self
            action:@selector(pressed:)];
    [self addGestureRecognizer:pressRecognizer];

    return self;
}

```

Get This Recipe's Code

To find this recipe's full sample project, point your browser to <https://github.com/erica/iOS-6-Cookbook> and go to the folder for Chapter 1.

Summary

`UIView`s and their underlying layers provide the onscreen components your users see. Touch input lets users interact directly with views via the `UITouch` class and gesture recognizers. As this chapter has shown, even in their most basic form, touch-based interfaces offer easy-to-implement flexibility and power. You discovered how to move views around the screen and how to bound that movement. You read about testing touches to see whether views should or should not respond to them. You saw how to “paint” on a view and how to attach recognizers to views to interpret and respond to gestures. Here's a collection of thoughts about the recipes in this chapter that you might want to ponder before moving on:

- Be concrete. iOS devices have perfectly good touch screens. Why not let your users drag items around the screen or trace lines with their fingers? It adds to the reality and the platform's interactive nature.
- Users typically have five fingers per hand. iPads, in particular, offer a lot of screen real estate. Don't limit yourself to a one-finger interface when it makes sense to expand your interaction into Multi-Touch territory for one or more users, screen space allowing.
- A solid grounding in Quartz graphics and Core Animation will be your friend. Using `drawRect:`, you can build any kind of custom `UIView` presentation you want, including text, Bezier curves, scribbles, and so forth.
- If Cocoa Touch doesn't provide the kind of specialized gesture recognizer you're looking for, write your own. It's not that hard, although it helps to be as thorough as possible when considering the states your custom touch might pass through.

- Use Multi-Touch whenever possible, especially when you can expand your application to invite more than one user to touch the screen at a time. Don't limit yourself to one-person, one-touch interactions when a little extra programming will open doors of opportunity for multiuser use.
- Explore! This chapter only touched lightly on the ways you can use direct manipulation in your applications. Use this material as a jumping-off point to explore the full vocabulary of the `UITouch` class.

This page intentionally left blank

Index

SYMBOLS

- , (commas), 183
- (hyphens), 180
- @ (at sign), 184
- | (pipe) character, 179
- ~ (tilde), 382
- ¨ (umlaut), 382

A

- above method, 135
- accessibility, 341
 - broadcasting, 348
 - enabling, 343
 - hints, 346
 - IB, 342
 - index paths, 451
 - labels, 345-346
 - overview of, 341-342
 - simulators, testing, 347
 - testing, 349-350
 - traits, 344-345
 - updating, 347
- Accessibility Inspector, 347
- accessibilityFrame property, 341-342
- accessibilityHint property, 346
- accessibilityLabel property, 342, 345
- accessibilityTraits property, 342, 344
- accessibilityValue property, 342

accessories, disclosure, 365-367

accessory views, 364. See also views

customizing, dismissing text views,
213-216

text, modifying around, 220-222

accumulating user tracings, 26

actions

action sheets, 101

alerts, 114-117

displaying text in, 117

objects, 103

buttons, connecting, 55

interaction, adding, 44

ongoing, displaying, 52

rows, adding, 391-394

UIProgressView class, 107

activities

customizing, adding, 333

data types, 335-336

services, items, 335-336

ActivityAlert class, 107

activityImage method, 333

activityTitle method, 332

activityType method, 332

activityViewController method, 333

activity view controllers, 328-336

Add Attribute button, 443

addConstraints: method, 186

adding

accessories, 216

action rows, 391-394

animation buttons, 58-59

attachments, 321

Auto-Hide sliders, 285

backsplashes, 16

buttons, 102, 104, 215, 254-255

cells, 370

compound predicates, 183

constraints, 186, 205

custom activities, 333

custom input views to nontext views,
230-233

data, 445-448

delay, 13

direct manipulation interfaces, 5-7

editing features to tables, 367

edits to Core Data table views, 457-463

efficiency, 63

entities, 443

flexible spaces between views, 181

frameworks, 301-302

gestures

layouts, 429-430

recognizers, 13

images, 301, 415

input clicks, 231

interactive actions, 44

menus, views, 44-46

modal controllers, 259

Multi-Touch interaction, 25

nametag properties, 137

pages, 94

pan gesture recognizers, 7-9

persistence, 234

pull-to-refresh option, 388-391

services, 331-332

state to buttons, 59

subviews, 134-135

tags, 136

traces, 26

transforms, scaling, 151

undo support, 458

universal support for split views, 267

views

- controllers, 287
- types to labels, 345

AFC (Apple File Connection), 301

affine matrices, 151

Ahmad, Bilal Sayed, 391

AIFF files, 124

AirPlay, enabling, 314

albums, images, 299. See also images

- Saved Photos, 299
- saving to, 309

alerts, 101

- action sheets, 114-117
- audio, 124-128
- building, 102
- customizing, 107
- delegates, 103-104
- displaying, 104
- indicators, 123-124
- interaction, restricting, 118
- local notifications, 121-123
- modal, creating with run loops, 110-113
- modifying, 107
- no-button, 107-110
- overlays, customizing, 117-119
- overview of, 101-105
- Please Wait, displaying progress, 105-107
- popovers, 119-121
- sounds, 126
- types, 104-105
- UIActivityIndicatorView class, 106
- views, applying variadic arguments, 113-114

alertVisualStyle property, 104

algorithms, smoothing, 22

aligning

- alignment property, 431
- alignment rectangles, 170-171
- indicators, 106
- views, constraints, 199-200

alignmentRectForFrame: method, 171

allApplicationSubviews() function, 133

allowsEditing property, 302, 307, 316

alpha levels, testing bitmap touches against, 17

Alpha property, 160

alphanumeric characters, 239

AND conditions, 67

animationImage property, 167

animations, 129

- blocks, combining multiple view changes, 161
- buttons, responses, 59-61
- creation/deletion, 424
- elements, adding to buttons, 58-59
- star sliders, 77
- transitions, 164
- UIView class, 158-159
- views
 - blocks, 159
 - bouncing, 165-133
 - Core Animation transitions, 163-165
 - fading in and out, 159-160
 - flipping, 162-163
 - image animations, 166-167
 - swapping, 161

APIs (application programming interfaces), 253

- Bluetooth connectivity, 471

appearance proxies, 66-68

Apple

Accessibility Programming Guide for iPhone, 343

gestures, detecting, 5

Apple File Connection. See AFC

application programming interfaces.

See APIs

Application Support folder, 301

application:didFinishLaunchingWithOptions: method, 122

applicationIconBadgeNumber property, 124

applications, 5

activities, 333

badging, 124

constraints, executing, 174

context, creating, 445

gestures, troubleshooting, 347

local notifications, 121-123

networks, checking status, 471

Reflection, 314

universal split view/navigation, 266-268

views, 129. *See also* views

VoiceOver gestures, 350. *See also* VoiceOver

windows, 133

applicationSupportsShakeToEdit property, 369

applying

built-in detectors, 244

disclosure accessories, 365-367

fetch requests, 449-450

JSON serialization, 487-489

multiple gesture recognizers simultaneously, 9-13

UIActivityIndicatorView class, 106

UIDatePicker class, 399-400

UIProgressView class, 107

variadic arguments with alert views, 113-114

views, pickers, 397-398

ARC (automatic reference counting), 98

arrays497

sections, 375-376

windows, returning, 133

arrows, formatting, 120

aspect ratios, formatting, 197-199

assets, libraries, 301-302

assigning

colors to backgrounds, 157

data sources to tables, 356

delegates, 357

images to buttons, 56

ranges to controls, 73

titles, 255

User Defined Runtime Attributes, 137

associating

center properties, 196

objects, naming views, 137-139

asynchronous downloads

networks, 480-486

one-call no-feedback, 486-487

asynchronous feedback, 105

at sign (@), 184

attachments, 321. See also e-mailing images

attributed text, enabling, 236

attributes, 443

baseline, 172

bottom, 171

centerX, 172

centerY, 172

constraints, 171-173

controls, 70-72

customizing, 443

height, 172
 leading, 172
 left, 171
 right, 171
 text, managing, 236
 top, 171
 trailing, 172
 User Defined Runtime Attributes,
 assigning, 137
 width, 172

audio

alerts, 124-128
 delays, 126
 disposing of, 126-127
 interfaces, 341. *See also* accessibility;
 VoiceOver
 system sounds, building, 124-125
 vibration, 125-126

Audio Queue, 124

AudioServicesAddSystemSoundCompletion()
 method, 125

AudioServicesPlaySystemSound class, 124

Auto-Hide sliders, adding, 285

autocapitalizationType property, 211

autocorrectionType property, 211

autolayout, 139

 constraints, 173

automatic reference counting. *See* ARC

autorepeats, steppers, 73

autoresizesSubviews property, 144

autoresizingMask property, 144

autosizing constraints, 192-193

availability, Wi-Fi, 471

AVAudioPlayer class, 124, 126

AVFoundation, 316-318

axes, 176

 constraints, 184

B

Back button, 250, 254

 titles, 256

backgrounds

 buttons, 56

 colors, 15, 157

backsplashes, adding, 16

badging applications, 124

bar buttons, 254-255

 macros, 256

baseline attribute, 172

basic flows, collection views, 411-414

beginTrackingWithTouch:withEvent:
 method, 74

beginUpdates method, 392

below method, 135

best practices, local notifications, 122-123

Bezier paths, 19

 smoothing, 22

bindings, variables, 177

bitmap images, testing against, 17-19.
See also images

blocks

 animations, combining multiple view
 changes, 161

 code, 105

 interaction, 109

 views, animations, 159

blue disclosures, 367

Bluetooth connections, 471

book properties, 276-277

Boolean values, 310

borders, styles, 212

bottom attribute, 171

bouncing views, 165-133

boundaries, defining, 140

bounds

- constraints, 174, 193
- movement, 14
- properties, 140, 144
- rectangles, 29
- views, 15

boxing Objective-C literals, 496-497**BrightnessController class, 269****broadcasting, accessibility updates, 348****building**

- alerts, 102
 - indicators, 123-124
 - overlays, 117-119
- buttons, 55-59
- controls, subclassing, 73-77
- dynamic slider thumbs, 63
- entities, 442
- floating progress monitors, 109-110
- model files, 442-443
- multiwheel tables, 396-399
- objects, classes, 444
- page indicator controls, 90-92
- paged scrollers, customizing, 93
- presentation indexes, 279
- pull controls, 83-87
- pull-to-refresh into tables, 389
- sections, 375-376, 453
- sounds, 124-125
- split view controllers, 261-266
- star slider, 77-80
- tables, 359
- text editors, 233-239
- toolbars, 99
- touch wheels, 80-83
- transforms, 33

tree parsers, 490

views, 134-136

XIB, 132

built-in cameras, iPhones, 300**built-in detectors, applying, 244****buttons, 49, 53-54. See also controls**

- actions, connecting, 55
- Add Attribute, 443
- adding, 102, 104, 254-255
- animation, adding to, 58-59
- appearance proxies, 66-68
- Back, 250, 254, 256
- bar, 254-255, 256
- building, 55-59
- Camera, 314
- Cancel, 102
 - alerts, 110
 - avoiding use on iPads, 116
 - video, 319
- clear, 212
- Contact Add, 53
- custom lock controls, 87-90
- deleting, 106, 108
- Detail Disclosure, 53
- Done, 120, 213
- Info Dark, 53
- Info Light, 53
- Interface Builder, 54-55
- keyboards, adding, 215
- no-button alerts, 107-110
- page indicator controls, 90-92
- paged scrollers, customizing, 93
- Play, 314
- pull controls, 83-87
- Redo, 233

- removing, 255
- responses, animating, 59-61
- Rounded Rectangle, 53, 54
- rounding, 56
- sliders, adding with custom thumbs, 61-66
- star sliders, 77-80
- state, adding to, 59
- switches/steppers, 72-73
- text, multiline, 58
- toolbars, 99
- touch wheels, 80-83
- twice-tappable segmented controls, 69-72
- UIControl class, subclassing, 73-77
- Undo, 233

C

- caches, executing fetch requests, 449**
- CAF files, 124**
- calculating distance, 7. See also mathematics**
- Calendar, 365**
- callbacks, 3**
 - alerts, 103
 - image picker controllers, 303-304
 - video, 319
 - views, 135-136
- calling view controllers, 250**
- Camera button, 314**
- cameraCaptureMode property, 308**
- cameraDevice property, 307**
- cameraFlashMode property, 308**
- Camera Roll, 299, 302**
 - activity view controllers, 330

- cameras**
 - images, snapping, 306-310
 - iPhones, 300
 - video, recording, 310-313
- Cancel button, 102**
 - alerts, 110
 - iPads, avoiding, 116
 - video, 319
- cancelButtonIndex property, 103, 115**
- cancelTrackingWithEvent: method, 74**
- canEditVideoAtPath: method, 318**
- canPerformAction: withSender: method, 45, 239**
- canPerformWithActivityItems: method, 333**
- capitalizing labels, 345**
- capturing video, 300**
- Catmull-Rom smoothing, 22**
- cellForRowAtIndex: method, 364**
- cells**
 - adding, 370
 - classes, registering, 356-357
 - colors, selecting, 362
 - customizing, 415-416
 - dequeueing, 357, 384
 - reordering, 370
 - returning, 377-378
 - search display controllers, registering, 384
 - swipes, 370
 - tables
 - creating checked, 363-365
 - views, 343, 361-363
- center property, 140**

centering, 145

- CGRect structures, 146

- indicators, 106

- views, 6, 196-197

centerX attribute, 172**centerY attribute, 172****CFRunLoopRun() method, 112****CGAffineTransformMakeRotation()**
function, 151**CGAffineTransformScale() function, 151****CGPointApplyAffineTransform() method, 17****CGRect structures, 146****CGRectContainsRect() function, 146****CGRectDivide() function, 141****CGRectEqualToRect() function, 141****CGRectFromString() function, 140****CGRectGetMidX() function, 141****CGRectInset() function, 141****CGRectIntersectsRect() function, 141, 152****CGRectMake() function, 140****CGRectOffset() function, 141****CGRectZero function, 141****Chagalls, Marc, 395****chains, responders, 1****characters**

- alphanumeric, 239

- commas (,), 183

- hyphens (-), 180

- pipe (|), 179

- at sign (@), 184

- spelling out, 350

- tilde (~), 382

- umlaut (¨), 382

checking network status, 471-473**Choi, Charles, 85****circles**

- detecting, 28-33

- layouts, 423-429

circular hit tests, 16. See also hit tests**classes**

- ActivityAlert, 107

- AudioServicesPlaySystemSound, 124

- AVAudioPlayer, 124, 126

- BrightnessController, 269

- cells, registering, 356-357

- customizing, subscripting 498

- DataManager, 389

- DetailViewController, 263

- DownloadHelper, 480

- InsetCollectionView, 417

- ModalAlertDelegate, 112

- NSCompoundPredicate, 449

- NSEntityDescription, 445

- NSIndexPath, 404

- NSJSONSerialization, 487

- NSLayoutConstraint, 175

- NSManagedObject, 444

- NSManagedObjectContext, 444

- NSPredicate, 239

- NSRegularExpression, 243

- NSTextCheckingResult, 244

- NSURLConnection, 476, 486

- NSXMLParser, 489

- objects, building, 444

- support, appearance proxies, 67

- TouchTrackView, 19

- UIActionSheet, 101, 114, 117

- UIActivityIndicatorView, 105-106

- UIActivityIndicatorViewStyleGray, 106

- UIActivityIndicatorViewStyleWhite, 106

collections

constraints, 186. *See also* constraints

views, 403

adding gestures to layouts, 429-430

basic flows, 411-414

Core Data, 464-468

creating circle layouts, 423-429

custom cells, 415-416

customizing item menus, 437-439

establishing, 405-407

flow layouts, 407-410

formatting true grid layouts, 431-437

interactive layout effects, 420-422

overview of, 403-405

scrolling horizontal lists, 416-420

scroll snapping, 422-423

colors

backgrounds, 15, 157

cells, selecting, 362

controls, customizing, 75

comma-separated value. *See* CSV

commands, copy, 45

commas (,), 183

common controllers, 299. *See also*

controllers

communication, alerts, 101-105.

See also alerts

comparing

collection views/tables, 403-405

constraints, 187-189

view-to-view, 184

components, format strings, 184-185

composite drawings, 26

compound predicates, 183

compound transforms, 33

conditions, AND, 67

configuring

fetch requests, 449

image pickers, 307-308

objects, delegates, 355

System Configuration framework, 472

conflicts, gestures, 12-13

connecting. *See also* networks

AFC, 301

buttons to actions, 55

networks

scanning for connectivity changes,
474-475

testing, 472

store coordinators, 445

views, 179-182, 183

consistency, constraints, 202

constants

properties, 187

UITableViewIndexSearch, 386

constraint:matches: method, 188

constraintMatchingConstraint: method, 188

constraintPosition: method, 194

constraints, 139

adding, 186

aspect ratios, formatting, 197-199

attributes, 171-173

autosizing, 192-193

comparing, 187-189

debugging, 205-207

declaring, 175

defining, 194

describing, 189-191

dumps, 207

format strings, 177-182, 184-185

formatting, 175-177

hierarchies, 174

- laws, 173-174
- macros, 202-205
- managing, 187
- mathematics, 172-173
- movement, 13-14
- orientation, 177-179, 200-202
- overview of, 169-170
- predicates, 183-184
- priorities, 173
- rectangles, aligning, 170-171
- sizes, 193-196
- storing, 186-187
- updating, 186-187
- views, 169
 - aligning, 199-200
 - centering, 196-197
 - connecting, 179-182
 - creating fixed-size, 192-196
 - updating, 201
- visual format, 176
- constraintsAffectingLayoutForAxis:**
 - method, 179**
- constraintSize: method, 184-185**
- constraintWithinSuperviewBounds**
 - method, 193**
- Contact Add button, 53**
- Contacts Add button style, 93**
- Contacts list, 365**
- containers**
 - accessibility, 343
 - customizing, 286-292
 - Objective-C literals, 497-498
- content**
 - messages, creating, 321-322
 - modes, viewing images, 308
 - sizing, 120
 - tables, 354
 - target offsets, customizing, 422
- contentSizeForViewInPopover property, 120**
- contentViewController property, 252**
- contexts, creating, 444**
- continueTrackingWithTouch:withEvent:**
 - method, 74**
- controllers, 299**
 - collection views, 405-406
 - images
 - e-mailing images, 321-323
 - pickers, 299-300
 - selecting images, 300-306
 - snapping, 306-310
 - menus, 45
 - message compose view, 323-325
 - modal, customizing, 258-259
 - navigation, 250, 252-255
 - page view, 251
 - popovers, 251-252
 - Quick Look preview, 336-339
 - search display, 383
 - social updates, posting, 325-328
 - split view, 251-255, 365
 - tab bar, 251
 - video
 - editing, 316-318
 - playing, 313-316
 - recording, 310-313
 - selecting, 318-320
 - views, 249
 - activity, 328-336
 - building split view controllers, 261-266
 - custom containers, 286-292

- developing navigation controllers/
split views, 252-255
 - message view, 323-325
 - modal presentation, 257-261
 - overview of, 249-252
 - page view controllers, 275-284
 - pushing/popping, 254
 - scrubbing pages in page view
controllers, 285-286
 - sequoias, 292-297
 - tab bars, 268-272
 - tab state memory, 272-275
 - transitions, 288-289
 - UINavigationController class, 255-257
 - universal split view/navigation
apps, 266-268
- controls, 49**
- attributes, 70-72
 - buttons
 - adding sliders with custom
thumbs, 61-66
 - animating responses, 59-61
 - appearance proxies, 66-68
 - building, 55-59
 - custom lock controls, 87-90
 - customizing paged scrollers, 93
 - Interface Builder, 54-55
 - overview of, 53-54
 - page indicator controls, 90-92
 - pull controls, 83-87
 - star sliders, 77-80
 - switches/steppers, 72-73
 - toolbars, 99
 - touch wheels, 80-83
 - twice-tappable segmented
controls, 69-72
 - events, 51-53
 - ranges, assigning, 73
 - remove, displaying, 369
 - ribbon, dragging, 85
 - types of, 50
 - UIControl class, 49-53, 73-77
- converting**
- swipes to drags, 36
 - XML into trees, 489-492
- coordinate systems, 142-143**
- coordinators, connecting store, 445**
- copy command, 45**
- copy: method, 237**
- Core Animation transitions, 163-165**
- Core Data, 441**
- collection views, 464-468
 - contexts, creating, 444
 - databases, querying, 448-450
 - entities, 442-444
 - models, 442-444
 - objects, removing, 450-451
 - overview of, 441
 - tables
 - data sources, 451-454
 - editing, 457-463
 - searching, 455-457
- Core Graphics**
- Quartz calls, 62
 - transforms, 142
- Core Media, 316-318**
- countDownDuration property, 401**
- counting rows/sections, 376**
- createLabel: method, 197**
- creation/deletion animations, 424**
- cross-references, 451**
- CSV (comma-separated value), 337**
- Curl style, 258**
- currentMode property, 145**

currentResponder method, 225
customizableViewControllers property, 251

customizing

- accessory views, dismissing text views, 213-216
- action sheets, 114-117
- activities, adding, 333
- alerts, 107, 117-119
- attributes, 443
- buttons, adding to keyboards, 215
- cells, 415-416
- classes, subscripting, 498
- containers, 286-292
- controls, building, 73-77
- default appearance of members, 66
- gestures, recognizers, 33-35
- group tables, coding, 394-395
- input views, 223-227, 230-233
- items, menus, 437-439
- lock controls, 87-90
- modal controllers, 258-259
- navigation controllers, 254
- objects, 446
- paged scrollers, 93
- relationships, 443
- segues, 292-297
- star sliders, 77
- tab bars, 269
- target content offsets, 422
- thumbs, adding sliders, 61-66
- traits, 363
- UISlider class, 62-63

cut: method, 237

cycles, constraints, 174

D

data

- adding, 445-448
- viewing, 446-448

Data Core, 445-448

data detectors, 243

data sources

- collection views, 406
- headers/footers, 411
- methods, 359, 384-385
- multiwheel tables, 356
- tables, 354
 - assigning, 356
 - Core Data as, 451-454

data types, activities, 335-336

databases, 443

- objects, removing, 450-451
- querying, 448-450
- SQLite, 447

DataManager class, 389

datePickerMode property, 400

dates

- pickers, creating, 399-400
- properties, 401

dealloc method, 126

debugging constraints, 189, 205-207

decimal points, entering numbers with, 239

declaring

- alignment rectangles, 171
- constraints, 175

decoration views, 404

defaults

- appearance of members, customizing, 66
- tab state, storing, 273

defining

- boundaries, 140
- constraints, 139, 194

delays

- adding, 13
- sounds, 126

delegate property, 210, 302

- video, editing, 318

delegates

- alerts, 103-104
- assigning, 357
- collection views, 406
- gestures, 36
- methods, 385
- multiwheel tables, 356
- objects, configuring, 355
- sections, 380
- tables, 354
- views, 354-355

delete: method, 237**deleteBackwards method, 228****deleting**

- badges, 124
- buttons, 106, 108, 255
- creation/deletion animations, 424
- gestures, recognizers, 13
- objects, 450-451
- pages, 94
- requests, 369-370
- text patterns, 242-246
- view controllers, 287

deploying iPhones, view controllers, 266**dequeueing cells, 357, 384****describing constraints, 189-191****design**

- constraints, 170. *See also* constraints
- target-action patterns, 50, 74

Detail Disclosure button, 53**DetailViewController class, 263****detecting**

- circles, 28-33
- gestures, 5
- nonreciprocal relationships, 451
- second-tap feedback, 70
- swipes, 36
- touches, mathematics, 16

detectors

- built-in, applying, 244
- data, 243

developing constraints, 189**devices, 63****diacritics, 382, 449****dictionaries, 497**

- info, 316
- metrics, 176

did-save method, 319**didAddSubview: method, 135****didMoveToSuperview: method, 135****didMoveToWindow: method, 135****direct manipulation interfaces**

- adding, 5-7
- touches, testing, 15

directions, scrolling, 407, 431**disabling**

- constraints, autosizing, 192-193
- gestures, recognizers, 13
- pattern matching, 244
- properties, transitions, 158
- VoiceOver, 349

disclosure accessories, 365-367**discoverability, pull controls, 83-85****discrete valued star sliders, building, 78**

dismissing

- modal controllers, 259

- no-button alerts, 108

- text

- keyboards, 209-213

- views, 213-216

dismissViewControllerAnimated:completion:
method, 257

dispatching events, 74-75

displaying

- alerts, 104

- HUD (heads-up display), 107

- images, 308

- ongoing actions, 52

- progress, 105-107

- remove controls, 369

- search display controllers, 383

- text in action sheets, 117

- views, 129, 157-158, 257-261. *See also*
views

disposing of system sounds, 126-127

distance, calculating, 7

distribution, views, 199-200

DNS (Domain Name System), 472

Done button, 120, 213

DownloadHelper class, 480

downloads

- asynchronous

- networks, 480-486

- one-call no-feedback, 486-487

- helpers, 482

- synchronous, 476-480

downward gestures, 36

dragging

- ribbon controls, 85

- scroll views, 35-39

- swipes, converting, 36

- views

- applying pan gesture recognizers, 8

- formatting, 6

DragView, 6

drawing, 2

- composite, 26

- smooth, 21-24

- touches, 19-21

drawRect: method, 2, 19, 26

Drobnik, Oliver, 110

dumps, constraints, 207

dynamic slider thumbs, building, 63

E

e-mailing images, 321-323

edges, insets, 411

editing

- images, 302

- tables, 371

- Core Data, 457-463

- views, 367-375

- text views, 246

- video, 316-320

- XIB, 132

editors, building text, 233-239

effects, interactive layouts, 420-422

efficiency, adding, 63

ejecting keyboards, 220

elements

- animation, adding to buttons, 58-59

- geometry, 147

embedding

- collection views, 418

- views, 406

Enabled check box, 343

enableInputClickWhenVisible method, 231

enablesReturnKeyAutomatically property, 211

enabling

- accessibility, 343
- AirPlay, 314
- attributed text, 236
- background colors, 16
- gestures, recognizers, 13
- touch feedback, 39-40

endRefreshing method, 389

endTrackingWithTouch:withEvent: method, 74

endUpdates method, 392

entering

- numbers, 239
- text, 209. *See also* text

entities, Core Data, 442-444

enumerating regular expressions, 243

enums, boxing, 496-497

equality, 172

equations, constraints, 172-173. *See also* mathematics

erasing old items, 26. *See also* deleting

errors

- networks, 471
- runtime, constraints, 192

Eschers, M. C., 395

establishing collection views, 405-407

events

- controls, 51-53
- dispatching, 74-75

exceptions, constraints, 170

executing

- applications, constraints, 174
- fetch requests, 449

existing popovers, 119. *See also* popovers

exporting video, 317

expressions

- boxing, 497
- regular, 242-243
- enumerating, 243
- websites, 244

extra state to buttons, adding, 59

extracting view hierarchy trees, 131

F

F (float), 496

Facebook, 299

- activity view controllers, 328-336
- social updates, posting, 325-328

Fade style, 258

fading views, 159-160

failures, runtime, 174

feature tests, Objective-C literals, 499

feedback, 62

- asynchronous, 105
- nonblocking, 109
- one-call no-feedback asynchronous downloads, 486-487
- second-tap, 70
- touches, 39-44

fetch requests, 448

- applying, 449-450
- configuring, 449
- predicates, 455

fields

- location, 248
- text, 49, 212-213. *See also* controls

files

- AFC, 301
- CSV, 337
- data, viewing, 446-448
- headers, UIKit, 67

- models, building, 442-443
- SQLite, 447
- store coordinators, connecting, 445
- types, 124, 337
- xcdatamodel, 442
- XIB, 132
- fileURLWithPath: method, 314**
- filtering text-entry, 239-241**
- finding views, 136-137**
- fingers, support, 25**
- firstAttribute property, 187**
- firstItem property, 187**
- firstResponder method, 225**
- fitting, 145**
- fixed size views, creating constraints, 192-196**
- flags**
 - autosizing, 192
 - clipsToBounds, 145
 - kSCNetworkReachability
 - FlagsConnectionOnTraffic, 472
 - kSCNetworkReachabilityFlagsIsDirect, 472
 - kSCNetworkReachabilityFlagsIsWWAN, 472
 - SCNetworkReachabilityGetFlags, 472
 - traits, 344
- flexible sizes, 199-200**
- flipping views, 162-163**
- Flip style, 258**
- float (F), 496**
- floating-point numbers, 242**
- floating progress monitors, building, 109-110**
- flow layouts, collection views, 407-414**
- folders**
 - Application Support, 301
 - Media, 301
- footerReferenceSize property, 409**
- footers**
 - data sources, 411
 - headers, formatting, 378
 - sizing, 409-410
- Form Sheet style, 258**
- format strings, constraints, 177-182, 184-185**
- formatting**
 - arrows, 120
 - aspect ratios, 197-199
 - Autolayout, 139
 - checked table cells, 363-365
 - constraints, 175-177
 - contexts, 444
 - controls, 74
 - headers/footers, 378
 - messages, content, 321-322
 - progress views, 107
 - repeating, 398
 - sections, indexes, 379-380
 - table views, 355-358
 - text, 237
 - text-input-aware views, 227-230
 - true grid layouts, 431-437
 - undo transactions, 459
 - video recording pickers, 311
 - views, dragging, 6
- forwarding touch events, 40-41**
- frameForAlignmentRect: method, 171**
- frames, 140**
 - alignment rectangles, 171
 - taps, sensing, 15
 - views, 140, 143-150

frameworks

- adding, 301-302
- AVFoundation, 316-318
- Core Data, 441. *See also* Core Data
- Media Player, playing video, 313-316
- Message UI, 321
- Quick Look, 336
- social updates, posting, 325-328
- System Configuration, 472

Full Screen style, 258**functionality, responders, 237****functions**

- allApplicationSubviews(), 133
- CGAffineTransformMakeRotation(), 151
- CGAffineTransformScale(), 151
- CGRectContainsRect(), 146
- CGRectDivide(), 141
- CGRectEqualToRect(), 141
- CGRectFromString(), 140
- CGRectGetMidX(), 141
- CGRectInset(), 141
- CGRectIntersectsRect(), 141, 152
- CGRectMake(), 140
- CGRectOffset(), 141
- CGRectZero, 141
- CMTimeMakeWithSeconds(), 317
- NSDictionaryOfVariableBindings(), 177
- NSStringFromCGRect(), 140
- subview utility, 133
- UIImageWriteToSavedPhotosAlbum(), 309
- utilities, rectangles, 140-141

G**gaps, view-to-view, 181****generating layouts, 409****geometry**

- constraints, 171
- orientation, modifying, 200-202
- rules, 181
- testing, 17
- views, 139-143, 147

gestureRecognizers property, 9**gestures, 1**

- conflicts, resolving, 12-13
- delegates, 36
- downward, 36
- layouts, adding, 429-430
- multiple recognizers, applying simultaneously, 9-13
- pan recognizers, adding, 7-9
- reassigning, 36
- recognizers, 5, 33-35
- VoiceOver, 347, 350

gestureWasHandled, 36**goesForward property, 294****graphical user interfaces. *See* GUIs****gray disclosures, 367****greater-than inequality, 172****groups**

- sections, 452
- tables, coding custom, 394-395
- undo, 459

guidelines

- buttons, 53
- HIGs, 54

GUIs (graphical user interfaces), 4
 navigation controllers, 252
 subview hierarchies, 129

H

H.264 Baseline Profile Level 3.0 video, 313

handlePan: method, 7-8

handling touches in views, 2

hardware, keyboards, 220. *See also*
 keyboards

hasText method, 228

headerReferenceSize property, 409

headers

data sources, 411

files, UIKit, 67

footers, formatting, 378

sizing, 409-410

heads-up display. *See* HUD

height attribute, 172

helpers

classes, XMLParser, 490

downloads, 482

hidden property, 157

hiding

badges, 124

custom alert overlays, 118

hierarchies

constraints, 174

Core Data, 442

views, 129-131

controllers, 259

recovering, 131-132

highlightedTextColor property, 363

HIGs (Human Interface Guidelines), 54

buttons, 53

hints, accessibility, 346

hit tests, 4, 16. *See also* tests

Holleman, Matthijs, 84

horizontal axis, 176

horizontal flow, 407

horizontal lists, scrolling, 416-420

Hosgrove, Alex, 35

HUD (heads-up display), 107

Human Interface Guidelines. *See* HIGs

hyphens (-), 180

I

IB (Interface Builder). *See also* XIB

accessibility, 342

constraints, 170

layout rules, 181

modal controllers, 259

views, naming, 137-139

iCloud, image picker controllers, 299

Identity Inspector, opening, 137

IMA audio, 124

image:didFinishSavingWithError:contextInfo:
 method, 309

images

adding, 415

albums, saving to, 309

animations, 166-167

bitmaps, testing against, 17-19

buttons, 56

displaying, 308

editing, 302

pickers

configuring, 307-308

controllers, 299-300

e-mailing images, 321-323

iPads/iPhones, 300

selecting images, 300-306

snapping images, 306-310

- retaking, 307
- Saved Photos album, 299
- selecting, 300
- shadows, 171
- simulators, adding, 301
- snapping, 309
- sources, 299-300
- synchronization, 299
- imageView property, 415**
- implementing**
 - collection views, 405
 - Quick Look, 337-339
 - tables, 358-361
 - TOUCHkit overlays, 41
 - wrapping, 277-278
- indexes****498**
 - paths, access, 451
 - presentations, 279
 - searching, 386
 - sections, 379-380, 455
 - titles, 452
- indexPathForObject: method, 451**
- indicators**
 - alerts, 123-124
 - centering, 106
 - progress, 117
- Info Dark button, 53**
- info dictionary, 316**
- Info Light button, 53**
- initWithFormat: arguments: method, 113**
- input**
 - accessories, dismissing text views, 214
 - clicks, adding, 231
 - matching, 242
 - text-input-aware views, formatting, 227-230
- inputAccessoryView property, 223**
- inputView property, 223**
- inserting text, modifying, 350**
- insertText method, 228**
- InsetCollectionView class, 417**
- insets, collection views, 410**
- instances**
 - appearance, customizing, 68
 - self.view, 186
 - UIViewController class, 250
- instantiating camera versions, image pickers, 307**
- integrating object-index paths, 451**
- interaction**
 - actions, adding, 44
 - alerts, 103, 118. *See also* alerts
 - blocking, 109
 - labels, 346
 - layout effects, 420-422
 - Multi-Touch, 2, 4, 24-28
 - views, 157-158
- intercepting touch events, 40-41**
- Interface Builder. See IB**
- interfaces**
 - activity view controllers, 328-336
 - APIs, 253
 - audio, 341. *See also* accessibility; VoiceOver
 - AVFoundation, 316
 - Bluetooth connectivity, 471
 - Core Data, 441. *See also* Core Data
 - direct manipulation
 - adding, 5-7
 - testing touches, 15
 - GUIs, subview hierarchies, 129
 - HIGs, 53-54

- modal controllers, 258
- multiple-button, 69. *See also* buttons
- Multi-Touch interaction, 4
- XIB, 132

intersections, testing views, 152

intersectView: method, 152

inverse relationships, 451

iOS

- BOCHS emulators, 228
- tables, 353-354

iPads

- action sheets, 114-117
- buttons, customizing sliders, 63
- image pickers, 300
- keyboards, dismissing, 209
- modal controllers, 258
- popovers, 120
- split view controllers, 253, 261, 365
- video, recording, 310
- view controllers, 266

iPhones

- accessibility, 341, 349-350. *See also* accessibility
- Accessibility Inspector, 347
- action sheets, 114-117
- Apple Accessibility Programming Guide for iPhone, 343
- Bluetooth connections, 471
- buttons, customizing sliders, 63
- cameras, 300
- coordinate systems, 143
- images, 299-300
- keyboards, dismissing, 209
- modal controllers, 258
- navigation controllers, 253

- popovers, 120
- tab bar controllers, 251
- unlocking, 350
- video, recording, 310
- view controllers, 266

iPods

- action sheets, 114-117
- buttons, customizing sliders, 63
- coordinate systems, 143
- keyboards, dismissing, 209
- modal controllers, 258
- touch wheels, building, 80-83
- video, recording, 310
- view controllers, 266

isAccessibilityElement property, 343

isCameraDeviceAvailable: method, 308

isDescendantOfView: method, 133

isFlashAvailableForCameraDevice: method, 308

items

- menus, customizing, 437-439
- old, erasing, 26
- selecting, 350
- services, activities, 335-336
- sizing, 407-408
- zooming, 420

itemSize property, 407

J

JavaScript

- Object Notation. *See* JSON
- regular expressions, testing, 244

JSON (JavaScript Object Notation) serialization, 487-489

K

keyboardAppearance property, 211

keyboards

- buttons, adding, 215
- input, adding to nontext views, 229
- replacing, 223
- tests, 221-222
- text
 - dismissing, 209-213
 - modifying around, 216-220

keyboardType property, 211

keys, paths, 451

keywords, 498

kSCNetworkReachabilityFlags

ConnectionOnTraffic flag, 472

kSCNetworkReachabilityFlagsIs

Direct flag, 472

kSCNetworkReachabilityFlagsIs

WWAN flag, 472

L

L (long), 496

labels

- accessibility, 345-346
- retrieving, 137

landscape layouts, 200

laws, constraints, 173-174

layers, views, 129

layouts

- Autolayout, 139
- circles, 423-429
- constraints, 169. *See also* constraints
- effects, interactive, 420-422
- flow, collection views, 407-414
- generating, 409
- gestures, adding, 429-430

nil, 405

presentations, 429

rules, 181

table views, 355

true grid, formatting, 431-437

leading attribute, 172

left attribute, 171

less-than inequality, 172

libraries

- assets, frameworks, 301-302
- images, 299
- Interface Builder, buttons, 54-55
- Regular Expression Library, 244
- save-to-library feature, 319

lifetimes

- alerts, 102
- UIViewController class, 250

limiting

- movement, 13
- touches, 25

lines, spacing, 407-408

lists, scrolling horizontal, 416-420

literals, Objective-C, 495

- boxing, 496-497
- containers, 497-498
- feature tests, 499
- numbers, 495-496
- subscripting, 498

live feedback, touches, 39-44

LL (longlong)496

LLVM Clang compilers, 495, 497

loadView method, 392, 405, 455

localization, accessibility, 346

local notifications, alerts, 121-123

location fields, 248

lock controls, customizing, 87-90

long (L), 496
 long presses, 5
 longlong (LL), 496
 look-ups, 498
 loops, creating modal alerts with run,
 110-113

M

macros, 495
 constraints, 202-205
 NSDictionaryOfVariableBindings()
 function, 177
 view controllers, 256

managing
 constraints, 187
 controllers, 249. *See also* controllers
 subviews, 134-136
 text, attributes, 236
 views, 134-136

manual retain-release. *See* MRR

master view controllers, 263. *See also* view
 controllers

MATCH operator, 239

matching
 input, 242
 patterns, disabling, 244

mathematics
 constraints, 172-173
 touches, detecting, 16

matrices, affine, 151

maximumDate property, 401

media. *See also* images; video
 QuickTime, 316
 selecting, 303

Media folder, 301

Media Player, 313-316

members, customizing appearance, 66

memory
 tab state, 272-275
 view controllers, 249

menus
 items, customizing, 437-439
 scrolling, 117
 showFromTabBar, 115
 showFromToolBar, 115
 showInView, 115
 spoken text, accessing, 350
 viewing, 114
 views, adding, 44-46

Message UI framework, 321

messages. *See also* e-mailing images
 compose view controllers, 323-325
 content, creating, 321-322
 text, sending, 323-325

metadata, images, 303

methods
 above, 135
 activityImage, 333
 activityTitle, 332
 activityType, 332
 activityViewController, 333
 addConstraints:, 186
 alignmentRectForFrame:, 171
 application:didFinishLaunching
 WithOptions:, 122
 AudioServicesAddSystemSound
 Completion(), 125
 beginTrackingWithTouch:with
 Event:, 74
 beginUpdates, 392
 below, 135
 callback
 image picker controllers, 303-304
 video, 319

- callbacks, 3, 135-136
- cancelTrackingWithEvent:, 74
- canEditVideoAtPath:, 318
- canPerformAction: withSender:, 45, 239
- canPerformWithActivityItems:, 333
- cellForRowAtIndexPath:, 364
- CFRunLoopRun(), 112
- CGPointApplyAffineTransform(), 17
- constraint:matches:, 188
- constraintMatchingConstraint:, 188
- constraintPosition:, 194
- constraintsAffectingLayoutForAxis:, 179
- constraintSize:, 184-185
- constrainWithinSuperviewBounds, 193
- continueTrackingWithTouch:
 - withEvent:, 74
- copy:, 237
- createLabel:, 197
- currentResponder, 225
- cut:, 237
- data sources, 359, 384-385
- dealloc, 126
- delegates, 103, 385. *See also* delegates
- delete:, 237
- deleteBackwards, 228
- did-save, 319
- didAddSubview:, 135
- didMoveToSuperview:, 135
- didMoveToWindow:, 135
- dismissViewControllerAnimated:
 - completion:, 257
- drawRect:, 2, 19, 26
- enableInputClickWhenVisible, 231
- endRefreshing, 389
- endTrackingWithTouch:withEvent:, 74
- endUpdates, 392
- fileURLWithPath:, 314
- firstResponder, 225
- frameForAlignmentRect:, 171
- handlePan:, 7-8
- hasText, 228
- image:didFinishSavingWithError:
 - contextInfo:, 309
- indexPathForObject:, 451
- initWithFormat: arguments:, 113
- insertText, 228
- intersectsView:, 152
- isCameraDeviceAvailable:, 308
- isDescendantOfView:, 133
- isFlashAvailableForCameraDevice:, 308
- loadView, 392, 405, 455
- multiwheel tables, 356
- numberOfSectionsInTableView, 359, 376
- objectAtIndexPath:, 451
- paste:, 237, 239
- pathToView(), 133
- performArchive, 234
- playInputClick, 231
- pointInside: withEvent:, 16
- prepareLayout, 423
- prepareWithActivityItems:, 333
- presentViewController:animated:
 - completion:, 257
- pushViewController:animated:, 254
- reachabilityChanged, 474
- reloadData, 356
- removeMatchingConstraint:, 188
- responders, touches, 3-4
- say:, 113
- scrollRangeToVisible:, 248
- sectionForSectionIndexTitle:atIndex, 452
- sectionIndexTitleForSectionName:, 452
- select:, 237
- selectAll:, 237

sendActionsForControlEvents:, 75
 setBarButtonItem:, 369
 setMessageBody:, 321
 setOn: animated:, 72
 setSubject:, 321
 setTitleTextAttributes: forState:, 70
 setTranslation: inView:, 7
 showFromBarButtonItem: animated, 115
 showFromRect:inView:animated:, 115
 startRefreshing, 389
 stopAnimating, 167
 swap:, 161
 tableView:canMoveRowAtIndexPath:, 459
 tableView:cellForRowAtIndexPath:, 356, 359
 tableView:didSelectRowAtIndexPath:, 354, 392
 tableView:numberOfRowsInSection:, 359, 377
 tableView:willSelectRowAtIndexPath:, 392
 targetContentOffsetForProposed
 ContentOffset:, 422
 textFieldAtIndex:, 105
 textFieldShouldReturn:, 210
 toggleBoldFace:, 237
 toggleItalics:, 237
 toggleUnderline:, 237
 touchesBegan: withEvent:, 6, 8, 25
 touchesEnded: withEvent:, 3
 touchesMoved: withEvent:, 19
 updateItemAtIndexPath:withObject:, 369
 updateTransformWithOffset:, 10
 updateViewConstraints, 200
 URLWithString:, 314
 viewDidAppear:, 193, 455
 viewDidLoad, 405, 411

views:, 176
 viewWillAppear:, 250, 386
 viewWillDisappear:, 250
 viewWithTag:, 136
 will-hide/will-show method pairs, 263
 willMoveToSuperview:, 135
 willRemoveSubview:, 136

metrics

constraints, 194
 dictionaries, 176
 predicates, 183-184

midpoints, x-axis/y-axis, 171

MIME types, 321

minimumDate property, 401

minuteInterval property, 401

mismatches, sections, 380

modal alerts, creating run loops with, 110-113

modal presentation, 257-261

ModalAlertDelegate class, 112

modalTransitionStyle property, 258

models

Core Data, 442-444
 files, building, 442-443

Model-View-Controller. See MVC

modes

content, viewing images, 308
 editing, 300
 UIViewContentModeScaleAspectFill, 308
 UIViewContentModeScaleAspectFit, 308

modifying

alerts, 107
 orientation, 200-202
 text
 accessory views, 220-222
 inserting, 350
 keyboards, 216-220

- transparencies, 160
- views
 - frames, 143
 - sizes, 144-145
- monitoring connectivity changes, 474**
- monitors, building floating progress, 109-110**
- movement, constraining, 13-14**
- movies, 314. See also video**
- moving**
 - frameworks, 301
 - indicators, 106
 - sliders, 350
 - subviews, 135
 - views, 143
- MPEG-4 Part 2 video (Simple Profile), 313**
- MRR (manual retain-release), 98**
- Multi-Touch interaction, 2, 4, 24-28**
- multiline button text, 58**
- multiple-button interfaces, 69. See also buttons**
- multiple gesture recognizers, applying simultaneously, 9-13**
- multipleTouchEnabled property, 25**
- multiplier property, 187**
- multiwheel tables, building, 396-399**
- MVC (Model-View-Controller), 354**
- touches, 1-5. *See also* gestures; touches
- views, hierarchies, 129-131
- navigation controllers, 250, 252-255**
- navigationController property, 254**
- networkActivityIndicatorVisible property, 472**
- networks, 471**
 - activity (Cocoa Touch), 123
 - asynchronous downloads, 480-486
 - connecting
 - scanning for connectivity changes, 474-475
 - testing, 472
 - JSON serialization, 487-489
 - one-call no-feedback, 486-487
 - status, checking, 471-473
 - synchronous downloads, 476-480
 - WWANs, 472
 - XML, converting into trees, 489-492
- nil layouts, 405**
- no-button alerts, 107-110**
- nonblocking feedback, 109**
- nonreciprocal relationships, detecting, 451**
- notation, JSON serialization, 487-489**
- notifications**
 - alerts, 121-123
 - keyboards, 216
 - video, playing, 314
- NSCompoundPredicate class, 449**
- NSDictionaryOfVariableBindings() function, 177**
- NSEntityDescription class, 445**
- NSIndexPath class, 404**
- NSJSONSerialization class, 487**
- NSLayoutConstraint class, 175**
- NSManagedObject class, 444**
- NSManagedObjectContext class, 444**
- NSPredicate class, 239**

N

- naming**
 - DNS, 472
 - entities, 442
 - model files, 442
 - views, 137-139, 179
- navigating**
 - gestures, 1. *See also* gestures; touches
 - horizontal lists, 416-420
 - menus, scrolling, 117

NSRegularExpression class, 243
NSStringFromCGRect() function, 140
NSTextCheckingResult class, 244
NSURLConnection class, 476, 486
NSXMLParser class, 489
numberOfPages property, 91
numberOfSectionsInTableView method, 359, 376
numbers
 entering, 239
 floating-point, 242
 Objective-C literals, 495-496

O

objectAtIndexPath: method, 451
Objective-C
 AVFoundation, 316
 literals, 495
 boxing, 496-497
 containers, 497-498
 feature tests, 499
 numbers, 495-496
 subscripting, 498
 regular expressions, 243
objects
 action sheets, 103
 associating, naming views, 137-139
 classes, building, 444
 customizing, 446
 delegates, configuring, 355
 fetch requests, 448
 JSON serialization, 487-489
 removing, 450-451
 UIActionSheet class, 117
 UIAlertView class, 102

 UICollectionViewController class, 405
 UIEvent, 2
 UIPickerView, 396
 UITouch, 2, 10
offsets, customizing target content, 422
old items, erasing, 26
on-the-fly, adding/deleting pages, 94
one-call no-feedback asynchronous downloads, 486-487
ongoing actions, displaying, 52
onscreen views, 4. See also views
opaque factors, 157
Open GL, 164
opening Identity Inspector, 137
operators, MATCH, 239
options
 action sheets, 114-117
 pull-to-refresh, adding, 388-391
 Undo, adding support, 368-369
orientation
 constraints, 177-179, 200-202
 split view controllers, 251
overlays
 alerts, customizing, 117-119
 taps, 119
 TOUCHkit, implementing, 41

P

Page Sheet style, 258
paged scrollers, customizing, 93
pages
 adding, 94
 deleting, 94
 indicator controls, 90-92
 view controllers, 251, 275-284

painting, 19. See also drawing

panning, 5

gesture recognizers, adding, 7-9

parallel gestures, recognizing, 10

parameters, views: method, 176

parsing

SAX parsers, 489

trees, building, 490

paste: method, 237, 239

pasting activity view controllers, 328-336

paths

Bezier, 19, 22

index access, 451

key sections, 451

pathToView() method, 133

patterns

matching, disabling, 244

presentation, image pickers, 302

target-action, 50, 74

text, deleting, 242-246

PCM audio, 124

performArchive method, 234

persistence, 272

adding, 234

phases, touches, 2-3

PhoneView (Ecam), 301

pickers

images

configuring, 307-308

controllers, 299-300

e-mailing, 321-323

selecting, 300-306

snapping, 306-310

UIDatePicker class, 399-400

video recording, creating, 311

views, applying, 397-398

pinches, 5

pipe (|) character, 179

placeholders, text, 212

Play button, 314

playing video, 313-316

playInputClick method, 231

playSound: selector, 50

Please Wait, displaying progress, 105-107

pointInside: withEvent: method, 16

points

sizes, 141

touch, 2. *See also* touches

pop-up menus, 44. See also menus

popovers

alerts, 119-121

controllers, 251-252

popping view controllers, 254

populating image collections, 301

portrait layouts, 200

positioning indicators, 106

posting social updates, 325-328

predefined callback methods, 3

predicates

constraints, 183-184

fetch requests, 455

metrics, 183-184

priorities, 184

sections, 376

view-to-view, 184

preferences, grouping tables, 395

prepareLayout method, 423

prepareWithActivityItems: method, 333

presentations

activity view controllers, 328-330

image pickers, 302-303

indexes, 279

- layouts, adding gestures, 429
- modal, 257-261
- patterns, image pickers, 302
- presentViewController:animated:completion:**
 - method, 257
- preventing keyboard dismissal, 210**
- previewing, Quick Look preview controller, 336-339**
- priorities**
 - constraints, 173
 - predicates, 184
 - priority property, 187
- processing touches, 1. See also touches**
- programming**
 - accessibility, updating, 342
 - Apple Accessibility Programming Guide for iPhone, 343
- progress**
 - displaying, 105-107
 - floating monitors, building, 109-110
 - indicators, 117
 - views, formatting, 107
- Project Navigator, moving frameworks, 301**
- properties**
 - accessibilityHint, 341-342, 346
 - accessibilityFrame, 342
 - accessibilityLabel, 342, 345
 - accessibilityTraits, 342, 344
 - accessibilityValue, 342
 - alertViewStyle, 104
 - alignment, 431
 - allowsEditing, 302, 307, 316
 - Alpha, 160
 - animationImage, 167
 - applicationIconBadgeNumber, 124
 - applicationSupportsShakeToEdit, 369
 - autocapitalizationType, 211
 - autocorrectionType, 211
 - autoresizesSubviews, 144
 - autoresizingMask, 144
 - book, 276-277
 - bounds, 140, 144
 - cameraCaptureMode, 308
 - cameraDevice, 307
 - cameraFlashMode, 308
 - cancelButtonIndex, 103, 115
 - center, 140
 - constant, 187
 - contentSizeForViewInPopover, 120
 - contentViewController, 252
 - countDownDuration, 401
 - currentMode, 145
 - customizableViewControllers, 251
 - date, 401
 - datePickerMode, 400
 - delegate, 210, 302, 318
 - enablesReturnKeyAutomatically, 211
 - firstAttribute, 187
 - firstItem, 187
 - footerReferenceSize, 409
 - geometry, 139
 - gestureRecognizers, 9
 - goesForward, 294
 - headerReferenceSize, 409
 - hidden, 157
 - highlightedTextColor, 363
 - imageView, 415
 - inputAccessoryView, 223
 - inputView, 223
 - isAccessibilityElement, 343
 - itemSize, 407
 - keyboardAppearance, 211
 - keyboardType, 211
 - maximumDate, 401

minimumDate, 401
 minuteInterval, 401
 modalTransitionStyle, 258
 multipleTouchEnabled, 25
 multiplier, 187
 navigationController, 254
 networkActivityIndicatorVisible, 472
 numberOfPages, 91
 priority, 187
 relation, 187
 returnType, 211
 row, 380
 scale, 143
 scrollDirection, 407
 secondAttribute, 187
 secondItem, 187
 section, 446
 sectionIndexTitles, 452
 sectionInset, 410
 sectionNameKeyPath, 451
 sections, 380
 secureTextEntry, 211
 selectedBackgroundView, 363
 spellCheckingType, 211
 splitViewController, 263
 subviews, 133
 tableViewFooterView, 379
 text fields, 212-213
 title, 263
 titleLabel, 58
 transform, 140, 151-152
 transitions, disabling, 158
 translatesAutoresizingMaskInto
 Constraints, 192
 UIApplication, 123
 undoManager, 368

urlconnection, 481
 userInteractionEnabled, 118, 158
 videoPath, 318

protocols

appearance, 68
 UIAccessibility, 341
 UIActivityItemSource, 328
 UIAppearance, 269
 UIInputViewAudioFeedback, 231
 UIKeyInput, 228
 UIResponderStandardEditActions, 236
 UITextFieldDelegate, 210
 UITextInputTraits, 211

proxies, appearance, 66-68

pull controls, 83-87

pull-to-refresh option, adding, 388-391

pushbuttons, 55. *See also* buttons

pushing view controllers, 254

pushViewController:animated: method, 254

Q

Quartz 2D calls, 19

querying

databases, 448-450
 subviews, 133-134

Quick Look, 336

 preview controller, 336-339

QuickTime, 316

R

ranges, assigning controls, 73

ratios, aspect, 197-199

reachabilityChanged method, 474

readiness, tables, 452

reassigning gestures, 36

recipes

- activity view controllers, 328-336
- alerts
 - audio, 124-128
 - creating modal alerts with run loops, 110-113
 - customizing overlays, 117-119
 - local notifications, 121-123
 - popovers, 119-121
 - variadic arguments with alert views, 113-114
- buttons
 - adding sliders with custom thumbs, 61-66
 - animating responses, 59-61
 - appearance proxies, 66-68
 - building, 55-59
 - custom lock controls, 87-90
 - customizing paged scrollers, 93
 - page indicator controls, 90-92
 - pull controls, 83-87
 - star sliders, 77-80
 - switches/steppers, 72-73
 - toolbars, 99
 - touch wheels, 80-83
 - twice-tappable segmented controls, 69-72
- collection views
 - adding gestures to layouts, 429-430
 - basic flows, 411-414
 - creating circle layouts, 423-429
 - custom cells, 415-416
 - customizing item menus, 437-439
 - formatting true grid layouts, 431-437
 - interactive layout effects, 420-422
 - scrolling horizontal lists, 416-420
 - scroll snapping, 422-423

constraints

- centering views, 196-197
- comparing, 187-189
- creating fixed-size views, 192-196
- describing, 189-191
- formatting aspect ratios, 197-199
- responding to orientation changes, 200-202
- Core Data
 - collection views, 464-468
 - editing tables, 457-463
 - searching tables, 455-457
 - table data sources, 451-454
- direct manipulation interfaces, adding, 5-7
- image picker controllers
 - e-mailing images, 321-323
 - selecting images, 300-306
 - snapping images, 306-310
- message compose view controller, 323-325
- movement, constraining, 13-14
- multiple gesture recognizers, applying simultaneously, 9-13
- networks
 - asynchronous downloads, 480-486
 - checking status, 471-473
 - converting XML into trees, 489-492
 - synchronous downloads, 476-480
- no-button alerts, 107-110
- pan gesture recognizers, adding, 7-9
- Quick Look preview controller, 336-339
- social updates, posting, 325-328
- table views
 - action rows, 391-394
 - adding pull-to-refresh option, 388-391

- creating checked, 363-365
 - editing, 367-375
 - multiwheel tables, 396-399
 - searching, 382-388
 - sections, 375-382
- tables, implementing, 358-361
- tests, against bitmap images, 17-19
- text
 - adding custom input views to
 - nontext views, 230-233
 - building text editors, 233-239
 - customizing input views, 223-227
 - deleting patterns, 242-246
 - detecting misspelling in UITextView, 246-247
 - dismissing UITextField keyboards, 209-213
 - dismissing views, 213-216
 - formatting text-input-aware views, 227-230
 - modifying views around accessory views, 220-222
 - modifying views around keyboards, 216-220
 - text-entry filtering, 239-241
- touches
 - adding menus to views, 44-46
 - customizing gesture recognizers, 33-35
 - detecting circles, 28-33
 - dragging scroll views from, 35-39
 - drawing, 19-21
 - live feedback, 39-44
 - Multi-Touch interaction, 24-28
 - smooth drawings, 21-24
 - testing, 15-17
- UIControl class, subclassing, 73-77
- video
 - editing, 316-318
 - playing, 313-316
 - recording, 310-313
 - selecting, 318-320
- view controllers
 - building split view controllers, 261-266
 - custom containers, 286-292
 - modal presentation, 257-261
 - page view controllers, 275-284
 - scrubbing pages in page view controllers, 285-286
 - segues, 292-297
 - tab bars, 268-272
 - UINavigationController class, 255-257
 - universal split view/navigation apps, 266-268
- views
 - bouncing, 165-133
 - Core Animation transitions, 163-165
 - fading in and out, 159-160
 - flipping, 162-163
 - frames, 143-150
 - image animations, 166-167
 - naming by object association, 137-139
 - retrieving transform information, 151-157
 - swapping, 161
- recognizers, 5**
 - gestures, 5, 33-35
 - multiple gesture, applying simultaneously, 9-13
 - pan gesture
 - adding, 7-9

- recording video, 310-313**
- recovering views, hierarchies, 131-132**
- rectangles**
 - alignment, 170-171
 - bounding, 29
 - utility functions, 140-141
- Redo buttons, 233**
- Reflection, 314**
- refreshing, 388-391**
- Regex Pal, 244**
- registering cells**
 - classes, 356-357
 - for search display controllers, 384
- Regular Expression Library, 244**
- regular expressions, 242-243**
 - enumerating, 243
 - websites, 244
- relation property, 187**
- relationships, 443**
 - constraints, 173. *See also* constraints
 - customizing, 443
 - inverse, 451
 - nonreciprocal, detecting, 451
- reliability, 5**
- reloadData method, 356**
- remove controls, displaying, 369**
- removeMatchingConstraint: method, 188**
- removing**
 - badges, 124
 - buttons, 106, 108, 255
 - gestures, recognizers, 13
 - objects, 450-451
 - subviews, 135
 - view controllers, 287
- reordering**
 - cells, 370
 - subviews, 135
- repeating**
 - formatting, 398
 - steppers, 73
- replacing keyboards, 223**
- requests**
 - delete, 369-370
 - fetch, 448
 - applying, 449-450
 - configuring, 449
 - predicates, 455
 - synchronous downloads, 476-480
- resizing views, 183, 216**
- resolving conflicts, gestures, 12-13**
- resources, regular expressions, 244**
- responders**
 - chains, 1
 - functionality, 237
 - methods, touches, 3-4
- responding to touches, tables, 359**
- responses, animating buttons, 59-61**
- restricting alerts, interaction, 118**
- retaking images, 307**
- retapping, 69. *See also* taps**
- retrieving**
 - keyboard bounds, 218
 - labels, 137
 - transforms
 - properties, 151-152
 - views, 151-157
 - views, 136-137
- returning**
 - cells, 377-378
 - values, action sheets, 115
- returnKeyType property, 211**
- ribbon controls, dragging, 85**
- right attribute, 171**
- rolling regular expressions, 243**

rotate recognizers, 429

rotations, 5

Rounded Rectangle button, 53-54

rounding buttons, 56

rows, 354. *See also* tables

actions, adding, 391-394

counting, 376

properties, 380

rules, 169. *See also* constraints

geometry, 181

layouts, 181

predicates. *See* predicates

separating, 183

views, centering, 196

run loops

creating modal alerts with, 110-113

modal alerts, creating with, 110-113

runtime

constraints, 174

errors, 192

User Defined Runtime Attributes,
assigning, 137

S

save-to-library feature, 319

Saved Photos album, 299

saving

did-save method, 319

images to albums, 309

video, 311

SAX parsers, 489

say: method, 113

scale property, 143

scaling, 145, 151, 173

scanning for connectivity changes, 474-475

scheduling local notifications, 122

SCNetworkReachabilityGetFlags flag, 472

ScreenCurtain, 350

screens, sensing taps, 15

scroll views, dragging from touches, 35-39

scrollDirection property, 407

scrollers, customizing paged, 93

scrolling

horizontal lists, 416-420

menus, 117

snapping, 422-423

text, 350

scrollRangeToVisible: method, 248

scrubbing pages in page view controllers,
285-286

SDKs (software development kits), 5

buttons, 53

collection views, 406

constraints, 170

controllers, 299. *See also* controllers

gesture conflicts, resolving, 12

remove controls, displaying, 369

UIKit header files, 67

UIWindow class, 130

views

controllers, 250

moving, 143

searching

Core Data, 455-457

data source methods, 384-385

indexes, 386

tables, 382-388

text strings, 247-248

views, 136-137

second-tap feedback, 70

secondAttribute property, 187

secondItem property, 187

- section property, 446**
- sectionForSectionIndexTitle:atIndex method, 452**
- sectionIndexTitleForSectionName: method, 452**
- sectionIndexTitles property, 452**
- sectionInset property, 410**
- sectionNameKeyPath property, 451**
- sections**
 - counting, 376
 - delegates, 380
 - groups, 452
 - indexes, 379-380, 455
 - insets, 410
 - keys, paths, 451
 - mismatches, 380
 - properties, 380
 - tables, 375-382, 453
- secureTextEntry property, 211**
- segments, twice-tappable segmented controls, 69-72**
- segues, 292-297**
- select: method, 237**
- selectAll: method, 237**
- selectedBackgroundView property, 363**
- selecting**
 - colors, cells, 362
 - images, 300-306
 - items, 350
 - media, 303
 - text, 350
 - video, 318-320
- selectors, 50**
 - playSound:, 50
- self.view instance, 186**
- sendActionsForControlEvents: method, 75**
- sending**
 - images, e-mailing, 321-323
 - text messages, 323-325
- sensing taps, 15**
- separating rules, 183**
- services**
 - adding, 331-332
 - items, activities, 335-336
- setBarButtonItems method, 369**
- setMessageBody: method, 321**
- setOn: animated: method, 72**
- setSubject: method, 321**
- setTitleTextAttributes: forState: method, 70**
- setTranslation: inView: method, 7**
- shadows, images, 171**
- sharing menu controllers, 45**
- showFromBarButtonItem: animated method, 115**
- showFromRect:inView:animated: method, 115**
- showFromTabBar menu, 115**
- showFromToolBar menu, 115**
- showInView menu, 115**
- simulators**
 - images, adding, 301
 - testing, 347
- Sina Weibo**
 - activity view controllers, 328-336
 - social updates, posting, 325-328
- Siri Assistant, 341**
- sizing**
 - alerts, 102
 - constraints, 193-196
 - content, 120
 - fixed size views, creating constraints, 192-196
 - flexible, 199-200

- footers, 409-410
- frames, 143
- headers, 409-410
- items, 407-408
- points, 141
- views, modifying, 144-145
- skinnable input elements, creating, 224**
- Slide style, 258**
- sliders, 49. See also controls**
 - Auto-Hide, adding, 285
 - buttons, adding with custom thumbs, 61-66
 - moving, 350
 - star, 77-80
 - thumbs, building dynamic, 63
- smooth drawings, 21-24**
- snapping**
 - images, 306-310
 - scrolling, 422-423
- social updates, posting, 325-328**
- software, 5**
- software development kits. See SDKs**
- sorting**
 - constraints, 173
 - fetch requests, 449
- Sound Effects, 124**
- sounds**
 - alerts, 126
 - building, 124-125
 - delays, 126
 - disposing of, 126-127
 - vibration, 125-126
- sources**
 - data source. *See* data sources
 - images, 299-300
- spacing lines, 407-408**
- spell checking, 246-247**
- spellCheckingType property, 211**
- spelling out characters, 350**
- split view controllers, 251-255**
 - building, 261-266
 - iPads, 365
 - universal split view/navigation apps, 266-268
- splitViewController property, 263**
- spoken text menus, accessing, 350**
- SQL tables, 448**
- SQLite files, 447**
- stacks, navigation controllers, 253-254**
- star sliders, 77-80**
- startRefreshing method, 389**
- state**
 - buttons, adding to, 59
 - storing, 364
 - tabs, memory, 272-275
- status, checking network, 471-473**
- steppers, 72-73**
- stopAnimating method, 167**
- store coordinators, connecting, 445**
- storing**
 - constraints, 186-187
 - state, 364
 - tab state, 273
- storyboards, 259**
- strings**
 - format, 177-182, 184-185
 - text, searching, 247-248
- structures, centering CGRect, 146**
- styles**
 - borders, 212
 - Contacts Add button, 93
 - modal presentations, 258
 - tables, 355
 - UIAlertViewStyleDefault, 104

UIButtonTypeCustom, 55
 UITableViewCellStyleDefault, 362
 UITableViewCellStyleSubtitle, 362
 UITableViewCellStyleValue1, 362
 UITableViewCellStyleValue2, 362

subclassing, UIControl class, 73-77

subscripting Objective-C literals498

subviews, 129. See also views

adding, 134-135
 managing, 134-136
 querying, 133-134
 removing, 135
 reordering, 135
 tagging, 136

sufficient constraints, 203

superviews, 183

support

Application Support folder, 301
 classes, appearance proxies, 67
 fingers, 25
 section-based tables, 380
 undo, adding, 234, 368-369, 458

swap: method, 161

swapping views, 161

swipes, 5

cells, 370
 detecting, 36
 dragging, converting, 36

switches, 49, 72-73. See also controls

synchronizing images, 299

synchronous downloads, 476-480

System Audio, 124

System Configuration framework, 472

system sounds

building, 124-125
 disposing of, 126-127

T

tableView:cellForRowAtIndexPath: method, 379

tables

building, 359
 collection views, comparing, 403-405
 Core Data
 as data sources, 451-454
 editing, 457-463
 searching, 455-457
 editing, 371
 implementing, 358-361
 iOS, 353-354
 readiness, 452
 sections, building, 453
 SQL, 448
 styles, 355
 touches, responding to, 359
 views, 353
 action rows, 391-394
 adding pull-to-refresh option,
 388-391
 cells, 343, 361-363
 coding custom group tables, 394-395
 creating checked, 363-365
 delegates, 354-355
 disclosure accessories, 365-367
 editing, 367-375
 formatting, 355-358
 multiwheel tables, 396-399
 searching, 382-388
 sections, 375-382
 UIDatePicker class, 399-400

**tableView:canMoveRowAtIndexPath:
 method, 459**

**tableView:cellForRowAtIndexPath:
 method, 356, 359**

tableView:didSelectRowAtIndexPath:
method, 354, 392

tableView:numberOfRowsInSection:
method, 359, 377

tableView:willSelectRowAtIndexPath:
method, 392

tabs

bars, 251, 268-272

state, 272-275

tags

UI_APPEARANCE_SELECTOR, 67

views, 136-137

taps, 5

overlays, 119

second-tap feedback, 70

sensing, 15

twice-tappable segmented controls, 69-72

target-action design patterns, 50, 74

target content offsets, customizing, 422

**targetContentOffsetForProposed
ContentOffset: method, 422**

templates, Core Data, 442

testing

accessibility, 349-350

against bitmap images, 17-19

Bluetooth connections, 471

features, Objective-C literals, 499

hit, 4

keyboards, 221-222

networks

checking status, 471

connecting, 472

pull controls, 85

regular expressions, 244

simulators, 347

touches, 15-17

views, intersections, 152

text

accessibility, 345

action sheets, displaying, 117

alerts, 113. *See also* alerts

attributed, enabling, 236

attributes, managing, 236

buttons, multiline, 58

editors, building, 233-239

entry, 209

fields, 49, 212-213. *See also* controls

file types, 337

formatting, 237

inserting, modifying, 350

keyboards, dismissing, 209-213

labels, 345-346

messages, sending, 323-325

patterns, deleting, 242-246

scrolling, 350

selecting, 350

spoken text menus, accessing, 350

strings, searching, 247-248

text-entry filtering, 239-241

titles, assigning, 255

traits, 211

UITextView class, detecting misspelling
in, 246-247

views

adding custom input views to
nontext views, 230-233

customizing input, 223-227

dismissing, 213-216

editing, 246

formatting text-input-aware, 227-230

modifying around accessory views,
220-222

modifying around keyboards,
216-220

- textFieldAtIndex: method, 105**
- textFieldShouldReturn: method, 210**
- thumbs. See also images**
 - customizing, adding sliders, 61-66
 - sliders, building dynamic, 63
- tilde (~), 382**
- titleLabel property, 58**
- titles**
 - assigning, 255
 - Back button, 256
 - indexes, 452
 - properties, 263
- toggle-style buttons, 55. See also buttons**
- toggleBoldFace: method, 237**
- toggleItalics: method, 237**
- tooggles, VoiceOver, 349-350. See also VoiceOver**
- toggleUnderline: method, 237**
- toolbars, building, 99**
- tools, painting, 19. See also drawing**
- top attribute, 171**
- touches, 1-5**
 - bitmap alpha levels, testing against, 17
 - circles, detecting, 28-33
 - direct manipulation interfaces,
 - adding, 5-7
 - drawing, 19-21
 - gestures. *See also* gestures
 - customizing recognizers, 33-35
 - recognizers, 5
 - limiting, 25
 - live feedback, 39-44
 - mathematics, detecting, 16
 - movement, constraining, 13-14
 - Multi-Touch interaction, 2, 4, 24-28
 - phases, 2-3
 - pull controls, 85
 - responder methods, 3-4
 - scroll views, dragging from, 35-39
 - smooth drawings, 21-24
 - tables, responding to, 359
 - testing, 15-17
 - touch wheels, 80-83
 - tracking, 3, 74
 - views, 4
 - adding menus, 44-46
 - handling, 2
- touchesBegan: withEvent: method, 6, 8, 25**
- touchesEnded: withEvent: method, 3**
- touchesMoved: withEvent: method, 19**
- TOUCHkit, 39**
 - overlays, implementing, 41
- TouchTrackView class, 19**
- traces, adding, 26**
- tracking touches, 3, 74**
- trailing attribute, 172**
- traits**
 - accessibility, 344-345
 - customizing, 363
 - text, 211
- transactions, creating undo, 459**
- transforms**
 - building, 33
 - properties, 140
 - retrieving, 151-152
 - views, 142, 151-157
- transitions**
 - animations, 164
 - Core Animation, 164
 - core animation, 163-165
 - modal presentations, 258
 - properties, disabling, 158
 - UIView class animations, 162
 - view controllers, 288-289

translatesAutoresizingMaskIntoConstraints property, 192

transparencies, modifying, 160

tree-based hierarchies, 129-131

recovering, 131-132

trees

parsing, building, 490

XML, converting into, 489-492

trimming video, 317-318

troubleshooting. See also errors; testing

gestures

applications, 347

conflicts, 12-13

network connection loss, 474

true grid layouts, formatting, 431-437

Tweetbot, 391

twice-tappable segmented controls, 69-72

Twitter, 299

activity view controllers, 328-336

social updates, posting, 325-328

two-dimensional arrays, 376

two-state buttons, 59. See also buttons

txt2re generator, 244

types

of alerts, 104-105

of controls, 50

data, 335-336

of files, 124, 337

of layouts, 200

MIME, 321

typing text (VoiceOver), 350

UIActivityIndicatorView class, 105-106

UIActivityIndicatorViewStyleGray class, 106

UIActivityIndicatorViewStyleWhite class, 106

UIActivityItemSource protocol, 328

UIActivityViewController class, 329

UIAlertView class, 101-102

UIAlertViewStyleDefault style, 104

UIAppearance protocol, 269

UI_APPEARANCE_SELECTOR tag, 67

UIApplication property, 123

UIButton class, 53-54. See also buttons

UIButtonTypeCustom style, 55

UICollectionViewCell class, 415

UICollectionView class, 403

UICollectionViewController class, 405

UICollectionViewFlowLayout class, 406-407, 420

UIColor class, 158

UIControl class, 49-53. See also controls

events, 51-53

subclassing, 73-77

UIDatePicker class, 399-400

UIDevice class, 231, 474

UIEvent objects, 2

UIGestureRecognizer, 7, 33

UIGestureRecognizerDelegate, 36

UIImage class, 55

UIImagePickerController class, 299-300, 308, 318

UIImageView class, 167

UIImageWriteToSavedPhotosAlbum() function, 309

UIInputViewAudioFeedback protocol, 231

UIKeyboardFrameEndUserInfoKey class, 218

UIKeyInput protocol, 228

U

U (unsigned), 496

UIAccessibility protocol, 341

UIActionSheet class, 101, 114, 117

UIKit class

calls, 19

item behaviors, 344

UIKit header files, 67

UILayoutContainerView class, 130

UIMenuController class, 44

UINavigationController class, 252

UINavigationController class, 254

UINavigationController class, 255-257

UIPageControl class, 90

UIPickerViewController class, 396

UIPickerView objects, 396

UIProgressView class, 105, 107

UIResponder class, 3, 213

UIResponderStandardEditActions
protocol, 236

UIScreen class, 143

UISegmentedControl class, 69

UISlider class, 62-63, 77. *See also* sliders

UISplitViewController class, 250-252

UIStepper class, 68, 73

UISwitch class, 72. *See also* switches

UITabBarController class, 251, 268

UITableView class, 353

UITableViewCellStyleDefault style, 362

UITableViewCellStyleSubtitle style, 362

UITableViewCellStyleValue1 style, 362

UITableViewCellStyleValue2 style, 362

UITableViewController class, 388

UITableViewIndexSearch constant, 386

UITapGestureRecognizer, 10

UITextAttributeFont class, 70

UITextAttributeShadowColor class, 71

UITextAttributeTextColor class, 70

UITextAttributeTextShadowOffset class, 71

UITextChecker class, 246

UITextField class, 209-213, 239

UITextFieldDelegate protocol, 210

UITextInputTraits protocol, 211

UITextView class, 246-247

UIToolbar class, 98. *See also* toolbars

UITouch objects, 2, 10

UITouchPhaseBegan, 2

UITouchPhaseCancelled, 3

UITouchPhaseEnded, 3

UITouchPhaseMoved, 2

UITouchPhaseStationary, 3

UIVideoEditorController class, 318

UIView class, 2, 3, 129

animation blocks, adding, 60

animations, 158-159

frame geometry categories, 147

UIViewContentModeScaleAspectFill
mode, 308UIViewContentModeScaleAspectFit
mode, 308

UIViewController class, 3, 250

UIWindow class, 130

umlaut (¨), 382

undo

support (Core Data), adding, 458

transactions, creating, 459

Undo buttons, 233

support, adding, 368-369

undoManager property, 368

unlocking iPhones, 350

unsigned (U), 496

updateItemAtIndexPath:withObject:
method, 369

updateTransformWithOffset: method, 10

updateViewConstraints method, 200

updating

- accessibility, 342, 347
- broadcasting, 348
- constraints, 186-187
- responding to, 272
- size components, 144
- social updates, posting, 325-328
- views
 - centers, 6
 - constraints, 201

URLs (uniform resource locators)

- asset libraries, moving, 301
- video, editing, 316

URLConnection property, 481**URLWithString: method, 314****User Defined Runtime Attributes, assigning, 137****userInteractionEnabled property, 118, 158****users**

- alerts, 101. *See also* alerts
- defaults, storing tab state, 273
- interaction, blocking, 109

utilities

- functions, rectangles, 140-141
- subview functions, 133

V

values

- action sheets, returning, 115
- Boolean, 310
- transforms, retrieving, 153

variables, bindings, 177**versions**

- constraints, 170
- macros, 256

vertical axis, 176**vertical flow, 407****vibration, 125-126****video**

- capturing, 300
- editing, 316-320
- exporting, 317
- playing, 313-316
- recording, 310-313
- saving, 311
- selecting, 318-320
- trimming, 317-318

videoPath property, 318**viewDidAppear: method, 193, 455****viewDidLoad method, 405, 411****viewing**

- alerts, 104
- data, 446-448
- HUD, 107
- images, 308
- menus, 114
- text in action sheets, 117

views, 129

- accessory, 364
- alerts, applying variadic arguments, 113-114
- animations, 158-159
- bouncing, 165-133
- bounds, 15, 193
- callbacks, 135-136
- centers, updating, 6
- collections, 403
 - adding gestures to layouts, 429-430
 - basic flows, 411-414
 - Core Data, 464-468
 - creating circle layouts, 423-429
 - custom cells, 415-416
 - customizing item menus, 437-439

- establishing, 405-407
 - flow layouts, 407-410
 - formatting true grid layouts, 431-437
 - interactive layout effects, 420-422
 - overview of, 403-405
 - scroll snapping, 422-423
 - scrolling horizontal lists, 416-420
- connecting, 179-182, 183
- constraints, 169. *See also* constraints
 - aligning, 199-200
 - centering, 196-197
 - creating fixed-size, 192-196
 - updating, 201
- controllers, 249
 - activity, 328-336
 - building split view controllers, 261-266
 - custom containers, 286-292
 - developing navigation controllers/split views, 252-255
 - message view, 323-325
 - modal presentation, 257-261
 - overview of, 249-252
 - page view controllers, 275-284
 - pushing/popping, 254
 - scrubbing pages in page view controllers, 285-286
 - segues, 292-297
 - tab bars, 268-272
 - tab state memory, 272-275
 - transitions, 288-289
 - UINavigationController class, 255-257
 - universal split view/navigation apps, 266-268
- Core Animation transitions, 163-165
- decoration, 404
- displaying, 157-158
- distribution, 199-200
- dragging, formatting, 6
- embedding, 406
- fading in and out, 159-160
- finding, 136-137
- flipping, 162-163
- frames, 140, 143-150
- geometry, 139-143, 147
- hierarchies, 129-132
- image animations, 166-167
- interaction, 157-158
- intersections, testing, 152
- menus, adding, 44-46
- moving, 143
- naming, 137-139, 179
- pickers, applying, 397-398
- progress, formatting, 107
- resizing, 183
- retrieving, 136-137
- rounding, 56
- scroll, dragging, 35-39
- shadows, 171
- sizes, modifying, 144-145
- subviews
 - adding, 134-135
 - managing, 134-136
 - querying, 133-134
 - removing, 135
 - reordering, 135
- swapping, 161
- tables, 353
 - action rows, 391-394
 - adding pull-to-refresh option, 388-391
 - cells, 343, 361-363
 - coding custom group tables, 394-395
 - Core Data, 457-463
 - creating checked, 363-365

- delegates, 354-355
- disclosure accessories, 365-367
- editing, 367-375
- formatting, 355-358
- implementing, 358-361
- iOS, 353-354
- multiwheel tables, 396-399
- searching, 382-388
- sections, 375-382
- UIDatePicker class, 399-400
- tagging, 136-137
- text
 - adding custom input views to
 - nontext views, 230-233
 - customizing input, 223-227
 - dismissing, 213-216
 - editing, 246
 - formatting text-input-aware, 227-230
 - modifying around accessory views, 220-222
 - modifying around keyboards, 216-220
 - scrolling, 350
- touches, 2, 4
- TOUCHkit overlays, implementing, 41
- transforms, 142, 151-157
- zooming, 161

views: method, 176

view-to-view gaps, 181

view-to-view predicates, 184

viewWillAppear: method, 250, 386

viewWillDisappear: method, 250

viewWithTag: method, 136

visual format constraints, 176, 185

Voice Control, 341

VoiceOver, 341. See also accessibility

- gestures, 350
- hints, 346
- labels, 345-346
- simulators, 347
- testing, 349-350
- traits, 344-345

W

warnings, Xcode, 451

WAV files, 124

websites, regular expressions, 244

wheels, touch, 80-83

width attribute, 172

Wi-Fi. See also networks

- availability, 471
- indicators, 123

wiggles, 84

will-hide/will-show method pairs, 263

willMoveToSuperview: method, 135

willRemoveSubview: method, 136

windows, 133. See also views

wireless wide area networks. See WWANs

wrapping implementations, 277-278

WWANs (wireless wide area networks), 472

X

x-axis, midpoints, 171

xcdatamodel file, 442

Xcode

- constraints, debugging, 205-207
- model files, building, 442-443
- number literals, 495
- warnings, 451

XIB (Xcode Interface Builder), 132

XML (Extensible Markup Language),
converting trees into, 489-492

XMLParser class, 490

Y

y-axis, midpoints, 171

Z

zooming

- items, 420

- views, 161