

Erica Sadun
Rich Wardwell



Completely
updated for

iOS 7
and
Xcode 5

The Core iOS

Developer's Cookbook

Fifth Edition



FREE SAMPLE CHAPTER

SHARE WITH OTHERS



The Core iOS Developer's Cookbook

This page intentionally left blank

The Core iOS Developer's Cookbook

Fifth Edition

Erica Sadun
Rich Wardwell

◆◆ 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 authors 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.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the web: informit.com/aw

Library of Congress Control Number: 2013953064

Copyright © 2014 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. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.

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 United States and other countries. OpenGL and the logo are registered trademarks 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.

ISBN-13: 978-0-321-94810-6

ISBN-10: 0-321-94810-6

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.

First printing: March 2014

Editor-in-Chief:
Mark Taub

Senior Acquisitions Editor:
Trina MacDonald

Senior Development Editor:
Chris Zahn

Managing Editor:
Kristy Hart

Senior Project Editor:
Betsy Gratner

Copy Editor:
Kitty Wilson

Indexer:
Lisa Stumpf

Proofreader:
Anne Goebel

Technical Reviewers:
Collin Ruffenach
Mike Shields
Ashley Ward

Editorial Assistant:
Olivia Basegio

Cover Designer:
Chuti Prasertsith

Senior Composer:
Gloria Schurick



Erica Sadun

*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.*



Rich Wardwell

*I dedicate this book to my wife, Julie, who was relegated
to single-parent status during this endeavor,
and my children, Davis and Anne, who never stopped
asking me to play with them even after countless refusals.*



Contents

Preface xiii

1 Gestures and Touches 1

Touches 1

Recipe: Adding a Simple Direct Manipulation Interface 5

Recipe: Adding Pan Gesture Recognizers 7

Recipe: Using Multiple Gesture Recognizers
Simultaneously 9

Recipe: Constraining Movement 14

Recipe: Testing Touches 15

Recipe: Testing Against a Bitmap 17

Recipe: Drawing Touches Onscreen 20

Recipe: Smoothing Drawings 22

Recipe: Using Multi-Touch Interaction 26

Recipe: Detecting Circles 29

Recipe: Creating a Custom Gesture Recognizer 34

Recipe: Dragging from a Scroll View 37

Recipe: Live Touch Feedback 40

Recipe: Adding Menus to Views 45

Summary 47

2 Building and Using Controls 49

The `UIControl` Class 49

Buttons 53

Buttons in Interface Builder 55

Recipe: Building Buttons 56

Recipe: Animating Button Responses 60

Recipe: Adding a Slider with a Custom Thumb 62

Recipe: Creating a Twice-Tappable Segmented Control 67

Working with Switches and Steppers 70

Recipe: Subclassing `UIControl` 72

Recipe: Building a Star Slider 76

Recipe: Building a Touch Wheel 79

Recipe: Creating a Pull Control 83

Recipe: Building a Custom Lock Control 88

Recipe: Image Gallery Viewer 93

Building Toolbars 96

Summary 98

3 Alerting the User 101

Talking Directly to Your User through Alerts 101

Recipe: Using Blocks with Alerts 105

Recipe: Using Variadic Arguments with Alert Views 110

Presenting Lists of Options 112

“Please Wait”: Showing Progress to Your User 115

Recipe: Modal Progress Overlays 117

Recipe: Custom Modal Alert View 119

Recipe: Basic Popovers 124

Recipe: Local Notifications 126

Alert Indicators 128

Recipe: Simple Audio Alerts 129

Summary 133

4 Assembling Views and Animations 135

View Hierarchies 135

Recipe: Recovering a View Hierarchy Tree 137

Recipe: Querying Subviews 139

Managing Subviews 141

Tagging and Retrieving Views 142

Recipe: Naming Views by Object Association 143

View Geometry 146

Recipe: Working with View Frames 150

Recipe: Retrieving Transform Information 158

Display and Interaction Traits 164

UIView Animations 165

Recipe: Fading a View In and Out 167

Recipe: Swapping Views 168

Recipe: Flipping Views 169

Recipe: Using Core Animation Transitions 170

Recipe: Bouncing Views as They Appear 172

Recipe: Key Frame Animations 174

Recipe: Image View Animations 176

Summary 177

5 View Constraints 179

- What Are Constraints? 179
- Constraint Attributes 180
- The Laws of Constraints 182
- Constraints and Frames 184
- Creating Constraints 186
- Format Strings 189
- Predicates 194
- Format String Summary 196
- Aligning Views and Flexible Sizing 198
- Constraint Processing 198
- Managing Constraints 199
- Recipe: Comparing Constraints 201
- Recipe: Creating Fixed-Size Constrained Views 204
- Recipe: Centering Views 209
- Recipe: Setting Aspect Ratio 210
- Recipe: Responding to Orientation Changes 212
- Debugging Your Constraints 214
- Recipe: Describing Constraints 215
- Constraint Macros 218
- Summary 221

6 Text Entry 223

- Recipe: Dismissing a UITextField Keyboard 224
- Recipe: Dismissing Text Views with Custom Accessory Views 228
- Recipe: Adjusting Views Around Keyboards 230
- Recipe: Creating a Custom Input View 235
- Recipe: Making Text-Input-Aware Views 240
- Recipe: Adding Custom Input Views to Nontext Views 243
- Recipe: Building a Better Text Editor (Part I) 246
- Recipe: Building a Better Text Editor (Part II) 248
- Recipe: Text-Entry Filtering 252
- Recipe: Detecting Text Patterns 255
- Recipe: Detecting Misspelling in a UITextView 260
- Searching for Text Strings 262
- Summary 262

7 Working with View Controllers 263

View Controllers 263

Developing with Navigation Controllers and Split Views 266

Recipe: The Navigation Item Class 271

Recipe: Modal Presentation 273

Recipe: Building Split View Controllers 278

Recipe: Creating Universal Split View/Navigation Apps 283

Recipe: Tab Bars 286

Remembering Tab State 290

Recipe: Page View Controllers 293

Recipe: Custom Containers 303

Recipe: Segues 309

Summary 315

8 Common Controllers 317

Image Picker Controller 317

Recipe: Selecting Images 319

Recipe: Snapping Photos 326

Recipe: Recording Video 331

Recipe: Playing Video with Media Player 333

Recipe: Editing Video 336

Recipe: Picking and Editing Video 339

Recipe: E-mailing Pictures 341

Recipe: Sending a Text Message 344

Recipe: Posting Social Updates 347

Summary 349

9 Creating and Managing Table Views 351

iOS Tables 351

Delegation 352

Creating Tables 353

Recipe: Implementing a Basic Table 356

Table View Cells 360

Recipe: Creating Checked Table Cells 362

Working with Disclosure Accessories 364

Recipe: Table Edits 366

Recipe: Working with Sections	374
Recipe: Searching Through a Table	381
Recipe: Adding Pull-to-Refresh to Your Table	387
Recipe: Adding Action Rows	390
Coding a Custom Group Table	395
Recipe: Building a Multiwheel Table	396
Using UIPickerView	400
Summary	401

10 Collection Views 403

Collection Views Versus Tables	403
Establishing Collection Views	405
Flow Layouts	407
Recipe: Basic Collection View Flows	412
Recipe: Custom Cells	416
Recipe: Scrolling Horizontal Lists	418
Recipe: Introducing Interactive Layout Effects	422
Recipe: Scroll Snapping	424
Recipe: Creating a Circle Layout	425
Recipe: Adding Gestures to Layout	431
Recipe: Creating a True Grid Layout	433
Recipe: Custom Item Menus	440
Summary	442

11 Documents and Data Sharing 445

Recipe: Working with Uniform Type Identifiers	445
Recipe: Accessing the System Pasteboard	451
Recipe: Monitoring the Documents Folder	454
Recipe: Activity View Controller	460
Recipe: The Quick Look Preview Controller	470
Recipe: Using the Document Interaction Controller	473
Recipe: Declaring Document Support	480
Recipe: Creating URL-Based Services	486
Summary	489

12 A Taste of Core Data 491

Introducing Core Data	491
Entities and Models	492

Creating Contexts	494
Adding Data	495
Querying the Database	498
Removing Objects	500
Recipe: Using Core Data for a Table Data Source	501
Recipe: Search Tables and Core Data	505
Recipe: Adding Edits to Core Data Table Views	508
Recipe: A Core Data–Powered Collection View	514
Summary	519

13 Networking Basics 521

Recipe: Checking Your Network Status	521
Scanning for Connectivity Changes	524
The URL Loading System	526
Recipe: Simple Downloads	528
Recipe: Downloads with Feedback	533
Recipe: Background Transfers	543
Recipe: Using JSON Serialization	546
Recipe: Converting XML into Trees	549
Summary	554

14 Device-Specific Development 555

Accessing Basic Device Information	555
Adding Device Capability Restrictions	556
Recipe: Checking Device Proximity and Battery States	559
Recipe: Recovering Additional Device Information	563
Core Motion Basics	565
Recipe: Using Acceleration to Locate “Up”	566
Working with Basic Orientation	568
Recipe: Using Acceleration to Move Onscreen Objects	571
Recipe: Accelerometer-Based Scroll View	575
Recipe: Retrieving and Using Device Attitude	578
Detecting Shakes Using Motion Events	579
Recipe: Using External Screens	581
Tracking Users	587
One More Thing: Checking for Available Disk Space	588
Summary	589

15 Accessibility 591

- Accessibility 101 591
- Enabling Accessibility 593
- Traits 594
- Labels 595
- Hints 596
- Testing with the Simulator 597
- Broadcasting Updates 599
- Testing Accessibility on iOS 599
- Speech Synthesis 601
- Dynamic Type 602
- Summary 604

A Objective-C Literals 605

- Numbers 605
- Boxing 606
- Container Literals 607
- Subscripting 608
- Feature Tests 609

Index 611

Preface

Welcome to a new *Core iOS Developer's Cookbook*!

With iOS 7, Apple introduced the most significant changes to its mobile operating system since its inception. This cookbook is here to help you get started developing for the latest exciting release. This revision introduces all the new features and visual paradigms announced at the latest Worldwide Developers Conference (WWDC), showing you how to incorporate them into your applications.

For this edition, the publishing team has split the cookbook material into manageable print volumes. This book, *The Core iOS 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 provides recipes you need for working with graphics, touches, and views to create mobile applications.

And there's *Learning iOS Development: A Hands-on Guide to the Fundamentals of iOS Programming*, which covers 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 7 development from the ground up. From Objective-C to Xcode, debugging to deployment, *Learning iOS Development* 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-7-Cookbook>, all of it refreshed for iOS 7 after WWDC 2013.

If you have suggestions, bug fixes, corrections, or anything else you'd like to contribute to a future edition, please contact us at erica@ericasadun.com or rich@lifeisrich.org. We thank you all in advance. We appreciate all your feedback, which helps make this a better, stronger book.

—Erica Sadun and Rich Wardwell, January 2014

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 your applications, preferably a new model iPhone or iPad. The following list covers the basics of what you'll 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, you need to 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 or iPad 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 devices or submit to the App Store.
- **A modern Mac running Mac OS X Mountain Lion (v 10.8) or, preferably, Mac OS X Mavericks (v 10.9)**—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, 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 a cellular data 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. 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 in Figure P-1 shows the key titles offered by Pearson Education that can help provide the training you need to become a skilled iOS developer.

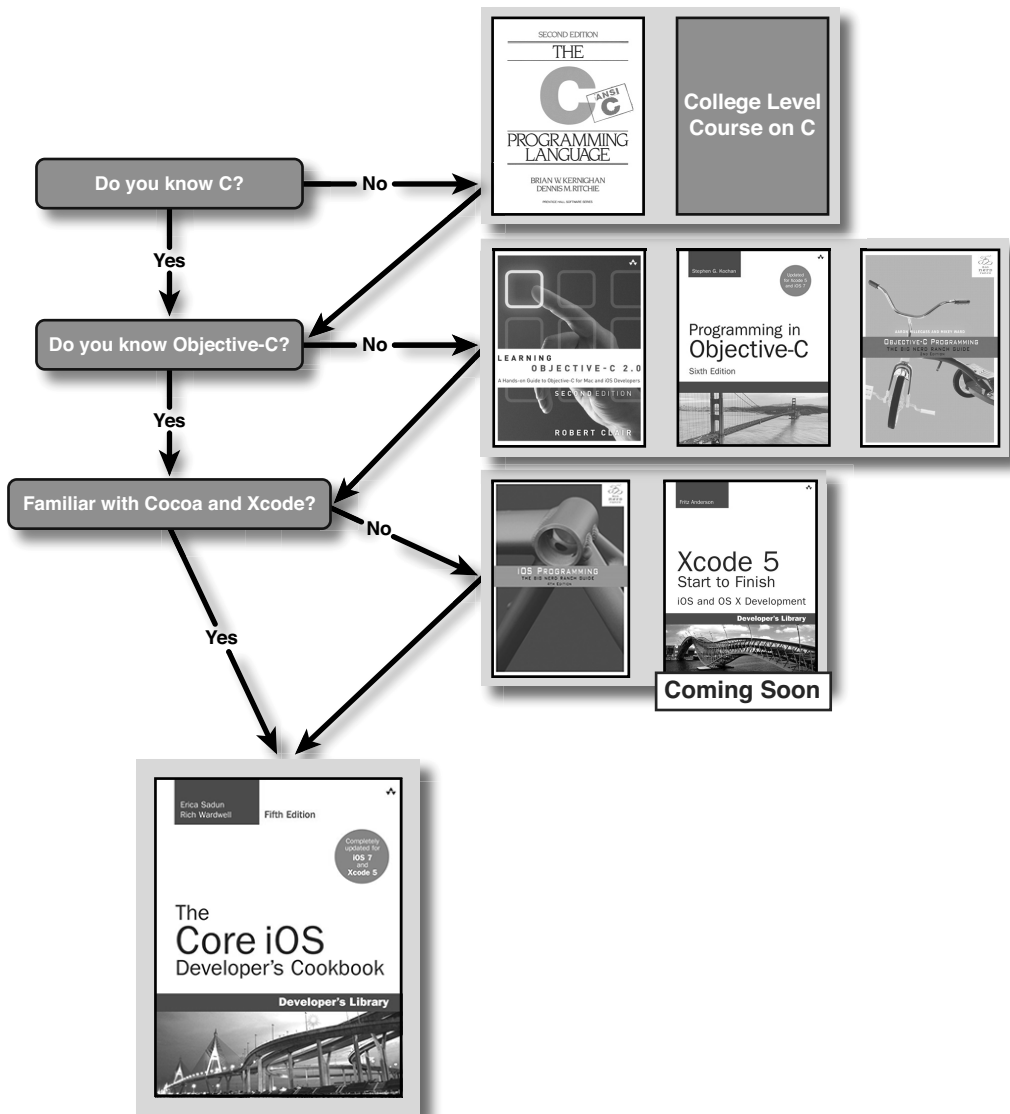


Figure P-1 A roadmap to becoming an iOS developer.

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*, 2nd edition, by Aaron Hillegass and Mikey Ward (Big Nerd Ranch, 2013)
- *Learning Objective-C 2.0: A Hands-on Guide to Objective-C for Mac and iOS Developers*, 2nd edition, by Robert Clair (Addison-Wesley, 2012)
- *Programming in Objective-C 2.0*, 6th edition, by Stephen Kochan (Addison-Wesley, 2012)

With the language under your belt, next up is tackling Cocoa (Mac) or Cocoa Touch (iOS) 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, Cocoa Touch, and Xcode (Apple Developer: developer.apple.com), or if you prefer books, you can learn from the best. Aaron Hillegass, founder of the Big Nerd Ranch in Atlanta (www.bignerdranch.com), is the coauthor of *iOS Programming: The Big Nerd Ranch Guide*, 2nd edition, and author of *Cocoa Programming for Mac OS X*, 4th 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.

Note

There are plenty of other books from other publishers on the market, including the bestselling *Beginning iOS 6 Development* by Dave Mark, Jack Nutting, Jeff LaMarche, and Fredrik Olsson (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 or iOS 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 a chance to attend WWDC, you’ll know what we’re talking about. The time you spend at 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 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 the needs of each of your apps.

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

- **Chapter 1, “Gestures and Touches”**—On iOS, touch provides the most important way for users to 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'll 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'll 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 heads-ups, from pop-up dialogs and progress bars to local notifications, popovers, and audio pings. This chapter 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 use a blocks-based API to easily handle alert interactions.
- **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'll learn how to build, inspect, and break down view hierarchies and understand how views work together. You'll discover the role that 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”**—Auto Layout revolutionized view layout in iOS. Apple's layout features 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, dynamic interfaces, rotation, or localization. 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”**—This chapter 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”**—In this chapter, you'll discover the various view controller classes that enable you to enlarge and order the virtual spaces your users interact with. You'll learn 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’ll read about selecting images from your photo library, snapping photos, and recording and editing videos. You’ll discover how to allow users to compose e-mails and text messages and how to post updates to social media such as Twitter and Facebook.
- **Chapter 9, “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. This chapter introduces tables, explaining 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 10, “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 11, “Documents and Data Sharing”**—Under iOS, applications can share information and data as well as move control from one application to another, using several system-supplied features. This chapter introduces the ways you can integrate documents and data sharing between applications. You’ll see how to add these features into your applications and use them smartly to make your apps cooperative citizens of the iOS ecosystem.
- **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. This chapter 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’ll learn how to design managed database stores, add and delete data, and query data from your code and integrate it into your UIKit table views and collection views.
- **Chapter 13, “Networking Basics”**—On Internet-connected devices, 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. This chapter surveys common techniques for network computing and offers recipes that simplify day-to-day tasks. This chapter introduces the new HTTP system in iOS 7 and provides examples for downloading data, including background downloading. You’ll also read about network reachability and web services, including examples of XML parsing and JSON serialization utilizing live services.

- **Chapter 14, “Device-Specific Development”**—Each iOS device represents a meld of unique, shared, momentary, and persistent properties. These properties include the device’s current physical orientation, its model name, its battery state, and its access to onboard hardware. This chapter looks at the device from its build configuration to its active onboard sensors. It provides recipes that return a variety of information items about the unit in use.
- **Chapter 15, “Accessibility”**—This chapter offers a brief overview of VoiceOver accessibility to extend your audience to the widest possible range of users. You’ll read about adding accessibility labels and hints to your applications and testing those features in the simulator and on the iOS device.
- **Appendix A, “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 that requires looking 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.

The examples in this book are not intended as standalone applications. Each is here 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 into your work as needed.

Contrast this 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 involving tasks that are related to but not essential to what you need to solve. Finding just the relevant portions is a lot of work, and the effort may outweigh any gains.

In this book, you’ll find exceptions to this one-file-with-the-story rule: This book 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 in 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 by 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-7-Cookbook on the open-source GitHub hosting site. There you'll 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 06 subfolder of the C05 folder.

Any project numbered 00 or that has a suffix (like 05b or 02c) refers to material that is used to create in-text coverage and figures. For example, Chapter 9's 00 – Cell Types project helped build Figure 9-2, showing system-supplied table view cell styles. Normally, we delete these extra projects. Early readers of this manuscript requested that we 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 to share them back to the main repository. If you come up with a new idea or approach, let us know. Our team is happy to include great suggestions both at the repository and in the next edition of this book.

Getting Git

You can download this book's source code by using the git version control system. Xcode 5 includes robust support for git within the IDE. The git command-line tool is also packaged with the Xcode 5 toolset. Numerous third-party free and commercial git tools are also available.

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 for project developers, it's a great place to find new code and collaborate on existing libraries. You can sign up for a free account at the GitHub website, where you can also copy and modify the repository for this book or create your own open-source iOS projects to share with others.

Contacting the Authors

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

Acknowledgments

Erica Sadun

This book would not exist without the efforts of Chuck Toporek, who was my editor and whip cracker for many years and multiple publishers. He is now at Omnigroup 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 do 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 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, Betsy Gratner, Kitty Wilson, Anne Goebel, Lisa Stumpf, Gloria Schurick, 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 to Stacey Czarnowki of Studio B, my agency of many years, and to the recently retired Neil Salkind; to tech reviewers Collin Ruffenach, Mike Shields, and Ashley Ward, 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 Blecher 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, chrallielinder, 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 Raythatha, 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.

Rich Wardwell

Although with deadlines mounting I may have versed otherwise, I give my deepest respect and appreciation to Erica for allowing me the honor of participating in the creation of this latest edition of the *Developer's Cookbook*. Through her mentoring and fish slapping, I've learned a great deal and, hopefully, at a minimum, flirted with the high standard that she has set forth.

Without the persistence of Trina MacDonald, our editor, I think I would have given up after the first chapter, screaming into the night. She has directed and encouraged through my frustration with, anxiety about, and ignorance of the book authoring process. I'm also indebted to Olivia Basegio, editorial assistant, and the team of technical editors she expertly arranged and managed. The technical editors' comprehensive efforts resulted in a much better book than we could have ever created on our own, for which I owe a great deal of gratitude: Thank you, Collin Ruffenach, Mike Shields, and Ashley Ward. The production team, including Betsy Gratner and Kitty Wilson, ensured that I appear much more adept at writing than I could ever hope to attain on my own. Many others at Addison-Wesley/Pearson to whom I've never spoken directly had a part; to each I'm immensely thankful for bringing this work to fruition.

A special thanks goes to Bil Moorhead, George Dick, and Daniel Pasco at Black Pixel, who were incredibly understanding as the demands of the book required attention and distraction from my daily responsibilities. It is an honor to work for and with the great folks at Black Pixel.

My parents, Rick and Janet, have been my greatest supporters, encouraging me in all my endeavors, including this one. My in-laws, Steve and Cary, provided a home for us during much of the writing of this book, for which I'm eternally grateful.

Finally, my wife and two children have been the true enablers of this project. I hope to reimburse in full for every honey-do item I neglected and every invite to play that I turned down. Their love and presence made it possible for me to complete this work.

Endnote

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 (Erica) and Diet Mountain Dew (Rich), who selflessly surrendered their contents in the service of this manuscript.

About the Authors

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 TUAW.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.

Rich Wardwell is a senior iOS and Mac developer at Black Pixel, with more than 20 years of professional software development experience in server, desktop, and mobile spaces. He has been a primary developer on numerous top-ranking iOS apps in the Apple App Store, including apps for *USA Today* and *Fox News*. Rich has served as a technical editor for *The Core iOS 6 Developer's Cookbook* and *The Advanced iOS 6 Developer's Cookbook*, both by author Erica Sadun, as well as many other Addison-Wesley iOS developer titles. When not knee-deep in iOS code, Rich enjoys "tractor therapy" and working on his 30-acre farm in rural Georgia with his wife and children.

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 authors as well as your name and phone or e-mail address. I will carefully review your comments and share them with the authors 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

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

This page intentionally left blank

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'll see how to create views that users can drag around the screen. You'll 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 interacting 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 `UIEvent` 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, whether 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 `UResponder` class. You decide where you process and respond to touches. Trying to implement low-level gesture control in non-responder 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 can 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 core application code 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 `UIViewController` class 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'll discover how touches work, how you can respond to them in your apps, and how to 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 happens 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:

- **`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. Notice that `UITouchPhaseStationary` does not generate a callback.

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 cancelled 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 within that view. 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'll 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 `UIApplication` instance, where it is either processed or discarded.

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, which enables you to provide 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 them yourself, and they provide a target-action feedback mechanism that hides implementation details. They also standardize how certain movements are categorized, as drags or swipes.

With gesture recognizer classes, you can trigger callbacks when iOS determines that the user has tapped, pinched, rotated, swiped, panned, or used a long press. 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.
- **Pans**—Pans occur when users drag their fingers across the screen. The recognizer determines the change in translation produced by that drag.
- **Long presses**—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

Before moving on to more modern (and commonly used) gesture recognizers, take time to understand and explore the traditional method of responding to a user's touch. You'll gain a deeper understanding of the touch interface by learning how simple `UIResponder` touch event handling works.

When you work with direct manipulation, your design focus moves from the `UIViewController` to the `UIView`. 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. Because this is 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 use touches with `UIImageView`.

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

```
@implementation DragView
{
    CGPoint startLocation;
}

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

- (void)touchesBegan:(NSSet*)touches withEvent:(UIEvent*)event
{
    // Calculate and store offset, 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-7-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, `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 the previous state.

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

```
@implementation DragView
{
    CGPoint previousLocation;
}

- (instancetype)initWithImage:(UIImage *)anImage
{
    self = [super initWithImage:anImage];
    if (self)
    {
        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-7-Cookbook> and go to the folder for Chapter 1.

Recipe: Using Multiple Gesture Recognizers Simultaneously

Recipe 1-3 builds on 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>
@end

@implementation DragView
{
    CGFloat tx; // x translation
    CGFloat ty; // y translation
    CGFloat scale; // zoom scale
    CGFloat theta; // rotation angle
}

- (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;
    scale = self.scaleX;
    theta = self.rotation;
}
```

```
- (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);

    // Guard against scaling too low, by limiting the scale factor
    if (self.scale > 0.5f)
    {
        self.transform = CGAffineTransformScale(self.transform, scale, scale);
    }
    else
    {
        self.transform = CGAffineTransformScale(self.transform, 0.5f, 0.5f);
    }
}

- (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;
}

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

```

Get This Recipe's Code

To find this recipe's full sample project, point your browser to <https://github.com/erica/iOS-7-Cookbook> and go to the folder for Chapter 1.

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 gesture recognizer method takes one argument, another gesture recognizer. This call creates a dependency between the two gesture recognizers. For the first gesture to trigger, the second gesture must fail. If the second gesture is recognized, the first gesture will not be.

iOS 7 introduces new APIs that offer more flexibility in providing runtime failure conditions via gesture recognizer delegates and subclasses. You implement `gestureRecognizer:shouldRequireFailureOfGestureRecognizer:` and `gestureRecognizer:shouldBeRequiredToFailByGestureRecognizer:` in recognizer delegates and `shouldRequireFailureOfGestureRecognizer:` and `shouldBeRequiredToFailByGestureRecognizer:` in subclasses.

Each method returns a Boolean result. A positive response requires the failure condition specified by the method to occur for the gesture to succeed. These `UIGestureRecognizer` delegate methods are called by the recognizer once per recognition attempt and can be set up between recognizers across view hierarchies, while implementations provided in subclasses can define class-wide failure requirements.

In real life, failure requirements typically mean 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 can limit interface slowdowns to where they're absolutely needed.

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.

Note

iOS 7 introduces UIKit Dynamics, for modeling real-world physical behaviors including physics simulation and responsive animations. By using the declarative Dynamics API, you can model gravity, collisions, force, attachments, springs, elasticity, and numerous other behaviors and apply them to UIKit objects. While this recipe presents a traditional approach to moving and constraining UI objects via gesture recognizers and direct frame manipulation, you can construct a much more elaborate variant with Dynamics.

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 offscreen. Recipe 1-4's code is general and can adapt to parent bounds and child views of any size.

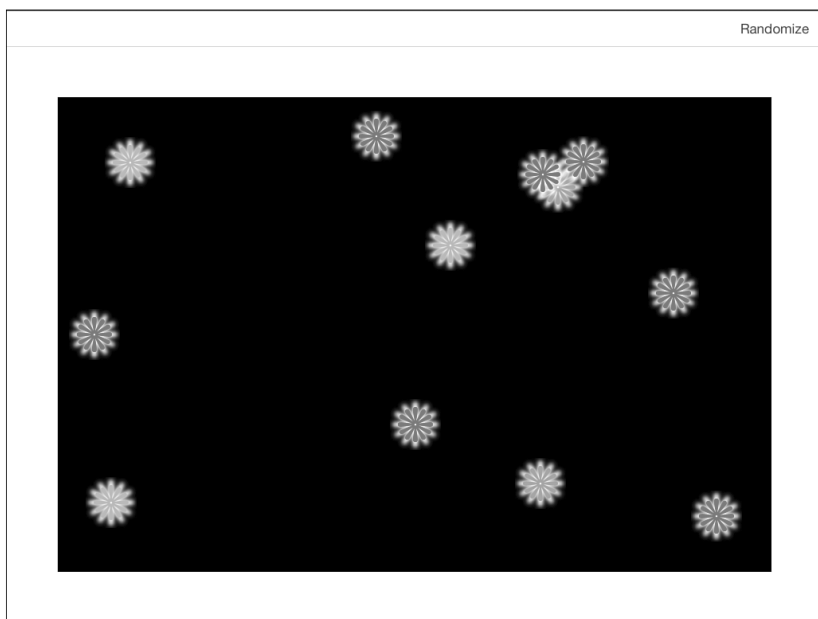


Figure 1-1 The movement of these flowers is constrained within the black rectangle.

Recipe 1-4 Constrained Movement

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

    // Restrict movement within the 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;
}
```

Get This Recipe's Code

To find this recipe's full sample project, point your browser to <https://github.com/erica/iOS-7-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 screen shot 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.

Visualize your actual view bounds by setting its background color, like this:

```
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.

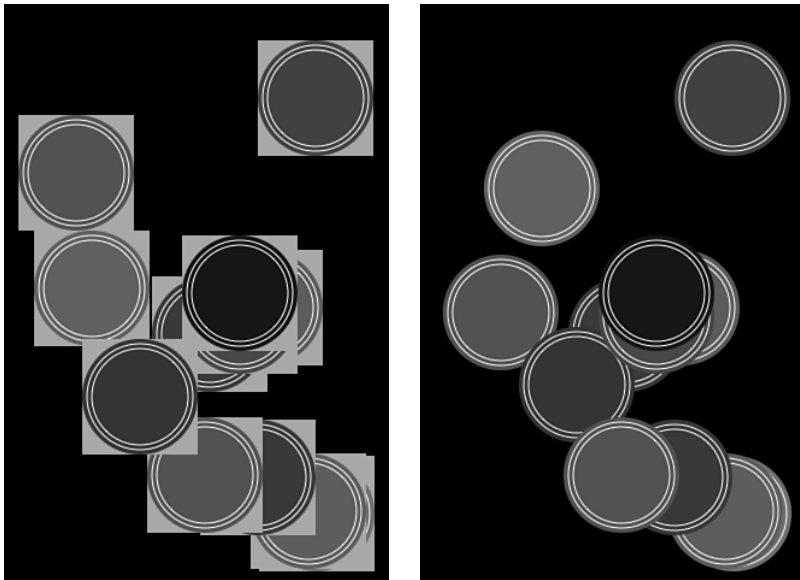


Figure 1-2 The application should ignore touches to the gray areas that surround each circle (left). The actual interface (right) uses a clear background (zero alpha values) to hide the parts of the view that are not used.

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'll see in Recipe 1-6, which follows in the next section, you can expand that test for much finer control.

Be aware that the math for touch detection on Retina display devices remains the same as that for older units, using the normalized points coordinate system rather than actual pixels. 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 `pointInside` 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 = kSideLength / 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-7-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. This 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)
{
    if (!sourceImage) return nil;

    // Establish color space
    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    if (colorSpace == NULL)
    {
        NSLog(@"Error creating RGB color space");
        return nil;
    }

    // Establish context
    int width = sourceImage.size.width;
    int height = sourceImage.size.height;
    CGContextRef context =
        CGContextCreate(NULL, width, height, 8,
            width * 4, colorSpace,
            (CGBitmapInfo) kCGImageAlphaPremultipliedFirst);
    CGColorSpaceRelease(colorSpace);
    if (context == NULL)
    {
```

```
        NSLog(@"Error creating context");
        return nil;
    }

    // Draw source into context bytes
    CGRect rect = (CGRect){.size = sourceImage.size};
    CGContextDrawImage(context, rect, sourceImage.CGImage);

    // Create NSData from bytes
    NSData *data =
        [NSData dataWithBytes:CGBitmapContextGetData(context)
                     length:(width * height * 4)];
    CGContextRelease(context);

    return data;
}

// Store the bitmap data into an NSData instance variable
- (instancetype)initWithImage:(UIImage *)anImage
{
    self = [super initWithImage:anImage];
    if (self)
    {
        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-7-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. With touch and drawing, you can 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.

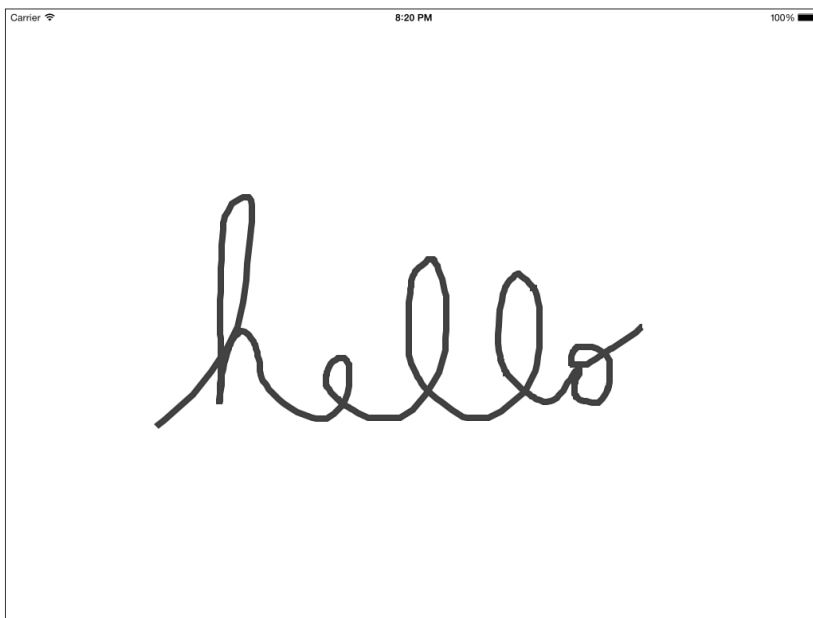


Figure 1-3 A simple painting tool for iOS requires little more than collecting touches along a path and painting that path with UIKit/Quartz 2D calls.

Although you could adapt this recipe to use gesture recognizers, there's really no point to it. The touches are essentially meaningless; they're only provided to create a pleasing tracing. The basic responder methods (that is, `touchesBegan`, `touchesMoved`, 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

@implementation TouchTrackerView

```
{
    UIBezierPath * path;
}

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

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    // Initialize a new path for the user gesture
    path = [UIBezierPath bezierPath];
    path.lineWidth = IS_IPAD ? 8.0f : 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];  
}  
@end
```

Get This Recipe's Code

To find this recipe's full sample project, point your browser to <https://github.com/erica/iOS-7-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.

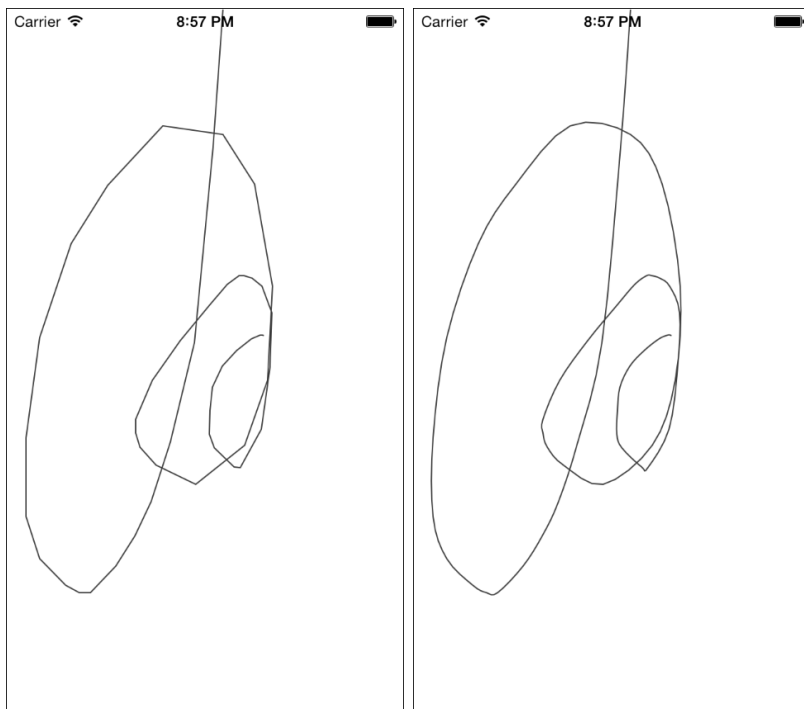


Figure 1-4 Catmull-Rom smoothing can be applied in real time to improve arcs between touch events. The images shown here are based on identical gesture input, with (right) and without (left) smoothing applied.

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 is 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.

Note

More extensive `UIBezierPath` utilities similar to `getPointsFromBezier` can be found in Erica Sadun's *iOS Drawing: Practical UIKit Solutions* (Addison-Wesley, 2013). For many excellent graphics-related recipes, including more advanced smoothing algorithms, check out the *Graphics Gems* series of books published by Academic Press and available at www.graphicsgems.org.

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-7-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 Recipe 1-9 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. 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 object, as 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 starts a new drawing at the end of each touch cycle. That works well for application bookkeeping but fails when it comes 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
- (void) clear;
@end

@implementation TouchTrackerView
{
    NSMutableArray *strokes;
    NSMutableDictionary *touchPaths;
}

// Establish new views with storage initialized for drawing
- (instancetype)initWithFrame:(CGRect)frame
{
    self = [super initWithFrame:frame];
    if (self)
    {
        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[key] = path;
    }
}

// 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-7-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.

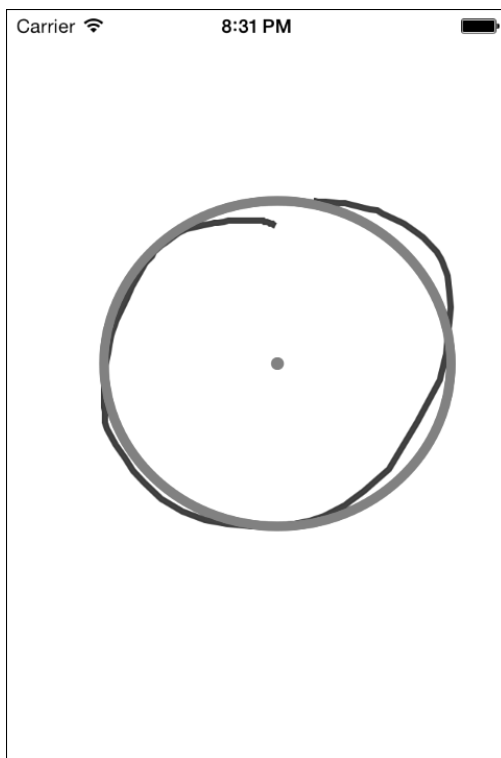


Figure 1-5 The dot and the outer ellipse show the key features of the detected circle.

In this implementation, detection uses a multistep test. A time test checks that the stroke is not lingering. A circle gesture should be quickly drawn. An inflection test checks that the touch does 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.

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 (NSUInteger 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 deltx = 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(deltx) != 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-7-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, but Recipe 1-11 does it. Subclassing `UIGestureRecognizer` enables you to build your own circle recognizer that you can add to views in your applications.

Start by importing `UIGestureRecognizerSubclass.h` from `UIKit` 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 the 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. 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 recognizer should save any detected circle so users know where the gesture occurred.

Recipe 1-11 Creating a Gesture Recognizer Subclass

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

// Called automatically by the runtime after the gesture state has
// been set to UIGestureRecognizerStateEnded. 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 = UIGestureRecognizerStatePossible;
}

// mirror of the touch-delivery methods on UIResponder
// UIGestureRecognizerSubclass aren't in the responder chain, but observe
// touches hit-tested to their view and their view's subviews.
```

```

// UIGestureRecognizer receives 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 = UIGestureRecognizerStateFailed;
        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-7-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 developer Alex Hosgrove encountered while he 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 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.

The implementation in Recipe 1-12 detects swipes that move down at least 16 vertical pixels without straying more than 12 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 gives 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)
```

```
const NSInteger kSwipeDragMin = 16;
const NSInteger kDragLimitMax = 12;
```

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

@implementation PullView

```
// Create a new view with an embedded pan gesture recognizer
- (instancetype)initWithImage:(UIImage *)anImage
{
    self = [super initWithImage:anImage];
    if (self)
    {
        self.userInteractionEnabled = YES;
        UIPanGestureRecognizer *pan =
            [[UIPanGestureRecognizer alloc] initWithTarget:self
             action:@selector(handlePan:)];
        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 > kSwipeDragMin) && (ABS(dy) < kDragLimitMax))
            touchtype = TouchSwipeLeft;
        else if ((-dx > kSwipeDragMin) && (ABS(dy) < kDragLimitMax))
            touchtype = TouchSwipeRight;
        else if ((dy > kSwipeDragMin) && (ABS(dx) < kDragLimitMax))
            touchtype = TouchSwipeUp;
        else if ((-dy > kSwipeDragMin) && (ABS(dx) < kDragLimitMax))
            touchtype = TouchSwipeDown;
        else
            finished = NO;

        // If unhandled and a downward swipe, produce a new draggable view
        if (!gestureWasHandled && finished &&
            (touchtype == TouchSwipeDown))
        {
```

```

        dragView = [[DragView alloc] initWithImage:self.image];
        dragView.center = touchLocation;
        [supersuper addSubview: dragView];
        scrollView.scrollEnabled = NO;
        gestureWasHandled = YES;
    }
    else if (gestureWasHandled)
    {
        // allow continued dragging after detection
        dragView.center = touchLocation;
    }
}

if(uiqr.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-7-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 and 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 past the overlay and 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

```

@implementation TOUCHkitView
{
    NSSet *touches;
    UIImage *fingers;
}

+ (instancetype)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
- (instancetype)initWithFrame:(CGRect)frame

```

```

{
    self = [super initWithFrame:frame];
    if (self)
    {
        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 *)touches withEvent:(UIEvent *)event
{
    touches = theTouches;
    [self setNeedsDisplay];
}

- (void)touchesEnded:(NSSet *)touches 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-7-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.

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 to 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 the state. For example, you don't want to offer a copy command if the view has no content to copy.

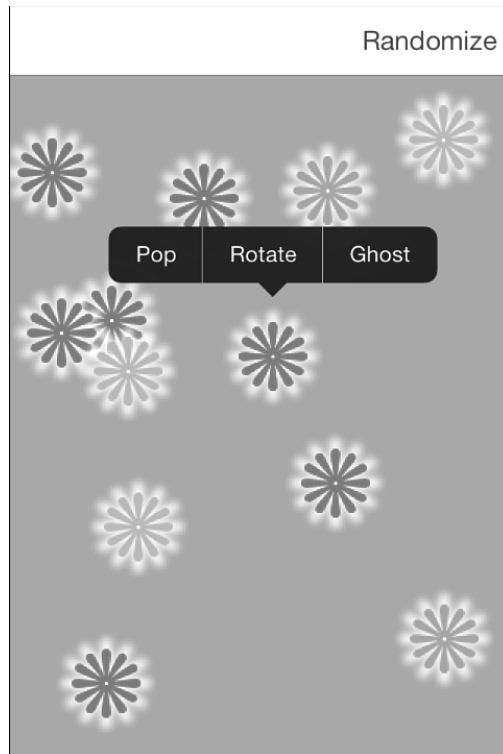


Figure 1-6 Contextual pop-up menus allow you to add interactive actions to first responder views.

Recipe 1-14 Adding Menus to Interactive Views

```
- (BOOL)canBecomeFirstResponder
{
    // Menus 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];
}

- (instancetype)initWithImage:(UIImage *)anImage
{
    self = [super initWithImage:anImage];
    if (self)
    {
        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-7-Cookbook> and go to the folder for Chapter 1.

Summary

UIViews 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, 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 recognizer 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 touches 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

A

acceleration

- monitoring, 566-568
- moving onscreen objects, 571-575

accelerometer, calculating, orientation, 569-570

accelerometer-based scroll view, 575-578

accessibility, 591-592

- broadcasting, updates, 599
- Dynamic Type, 601-604
- enabling, 593-594
- hints, 596
- IB, 592
- labels, 595-596
- speech synthesis, 600-601
- testing
 - on iOS, 599-601
 - with simulators, 597-598
- traits, 594-595

Accessibility Inspector, 597-598

accessibilityFrame, 592

accessibilityHint, 592

accessibilityLabel, 592

accessibilityLabel property, 595

accessibilityPath, 592

accessibilityTraits, 592

accessibilityValue, 592

accessing

- basic device information, 555-556
- sensor data, 566
- system pasteboards, 451-452

accessory views, 363-364

accessoryType property, 363

accumulating, user tracings for composite drawings, 27-29

action rows, adding to tables, 390-394

action sheets, 112

- displaying, text, 114-115
- values, 113

actions, connecting, to buttons, 55

activating, refresh controls, 388

activities

- activity view controller, 469-470
- excluding, 470

activity item sources, 462

activity view controller, 460-461

- activity item sources, 462
- adding services, 464-465
- code, 463-464
- excluding activities, 470
- item providers, 462
- item source callbacks, 462-463
- items and activities, 469-470
- presenting, 461

activityImage, 465

activityTitle, 465

activityType, 465

activityViewController, 465

adapting, table edits to Core Data, 510-514

adding

- action rows, to tables, 390-394
- animated elements, to buttons, 59
- cells to tables, 369
- child view controllers, 304
- custom buttons to keyboards, 229-230
- custom input views to nontext views, 243
- custom selection traits, to table view cells, 361-362
- data, Core Data, 495-496
- device capability restrictions, 556-557
- display links, 583
- edits to Core Data table views, 508, 510-514
 - undo/redo support, 508-509
 - undo transactions, 509-510
- efficiency to sliders, 64
- extra state to buttons, 59-60
- gestures to layout, collection views, 431-433
- handler method, URL-based services, 488
- input clicks, 243
- keyboard input to nontext views, 241-243
- menus, to views, 45-47
- motion effects, 85
- page indicator controls, 92
- pan gesture recognizers, 7-9
- persistence to text views, 246-248
- photos to simulators, 319
- pull-to-refresh to tables, 387-390

services, activity view controller, 464-465

simple direct manipulation interfaces, 5-7

sliders, 62-67

subviews, 141

undo support

table edits, 367

to text views, 246-248

adjusting

sizes, frames, 151-152

views, around keyboards, 230-234

advertisingIdentifier, 588

alert indicators, 128-129

badging applications, 129

alert sounds, 131

alert views, variadic arguments, 110-111

alerts, 101

audio alerts, 129, 131

delays, 131

disposing of system sounds, 132

system sounds, 129-130

vibration, 130-131

blocks, 105-107

blocks-based alerts, creating, 108-110

custom modal alert view, 119-120

frosted glass effect, 120-124

delegates, 103-104

displaying, 104-105

kinds of alerts, 104-105

lists of options, 112-114

local notifications, 126-127

best practices, 127-128

modal progress overlays, 117-119

popovers, 124-126

simple alerts, building, 101-102

tappable overlays, 119

variadic arguments, 110-111

- alerts, audio alert modules for system frameworks, 130
- alertVisualStyle property, 104**
- aligning views and flexible sizing with constraints, 198
- alignment rectangles, 185-186
 - declaring, 186
- allowsEditing property, 336**
- Alpha property, 167-168**
- animated elements, adding to buttons, 59
- animating
 - button responses, 60-62
 - constraints, 203
- animation
 - creation and deletion animation, collection views, 426
 - image view animations, 176
 - key frame animations, 174-175
 - UIView animations, 165-166
 - building with blocks, 166-167
- animation blocks, adding to controls (UIView), 60-61
- annotation, document interaction controller, 476
- Apple, checking network status, 521
- application activities, code, 466-469
- array literals, 608
- ASIdentifierManager, 588**
- aspect ratio, 210-211
- assets library module, image picker controllers, 319-320
- assigning
 - data sources, tables, 354
 - delegates, 356
- attitude, devices, 578-579
- attributed text, enabling, 249

- attributes
 - constraints, 180-181
 - controlling, 249
 - controls and, 69-70
 - Core Data, 493
- audio alerts, 129, 131**
 - delays, 131
 - disposing of system sounds, 132
 - modules for system frameworks, 130
 - system sounds, 129-130
 - vibration, 130-131
- Audio Services, 131**
 - playing sounds, alerts, and vibrations, 132-133
- Audio Toolbox framework, 130**
- Auto Layout, 179**
 - alignment rectangles, 185-186
 - constraint processing, 198-199
 - constraints, 183
- Auto Layout system, 151**
- autocapitalizationType, 225**
- autocorrectionType, 225**
- autoresizesSubviews property, 151**
- autoresizing constraints, disabling, 204-205**
- autoresizingMask property, 152**
- AV Foundation**
 - Core Media and, 337-338
 - video, trimming, 338-339

B

- back buttons, navigation item class, 271**
- background transfers, 543-544**
 - testing, 544-545
 - web services, 546
 - converting XML into trees, 549-551
 - JSON serialization, 546-548

- badging applications, 129
- bar buttons, navigation controllers, 269
- basic constraint declarations, creating, 187
- battery state, monitoring, 560-562
- `beginTrackingWithTouch:withEvent:`, 73
- Bezier paths, creating smooth with Catmull-Rom splining, 23-25
- bitmap alpha levels, testing, touches against, 18-19
- bitmaps, testing against, 17-19
- blocks
 - alerts, 105-107
 - creating blocks-based alerts, 108-110
 - building animations, 166-167
 - retain cycles, 107-108
- blocks-based alerts, 108-110
- book properties, page view controllers, 294-295
- border style, 227
- `bottomLayoutGuide`, 270
- bouncing, views, 172-174
- bounds, 152
- boxed expressions, 606
 - enums, 607
- broadcasting, updates, accessibility, 599
- built-in type detectors, 257
- button responses, animating, 60-62
- buttons, 53-54
 - adding, to keyboards, 229-230
 - animating button responses, 60-62
 - building, 56-60
 - adding animated elements, 59
 - adding extra state to buttons, 59-60
 - buttons that toggle on and off, 57-58
 - multiline button text, 59

- connecting to actions, 55
- Interface Builder, 55

C

- calculating
 - orientation, from accelerometer, 569-570
 - relative angle, 570-571
- callbacks, views, 142
- cameras, `UIImagePickerController` class, 328
- Cancel button, 114
- `cancelTrackingWithEvent:`, 73
- `canPerformWithActivityItems`, 465
- caret symbols, 106
- Catmull-Rom, 22-23
 - creating smoothed Bezier paths using Catmull-Rom splining, 23-25
- cell classes, registering, 355
- cells, 354
 - adding to tables, 369
 - checked table cells, creating, 362-364
 - custom cells, collection views, 416-417
 - dequeuing, 355-356
 - registering for search display controllers, 383
 - reordering, 369
 - returning in sections, 376-377
 - swiping, 369
 - table view cells, 360
 - adding custom selection traits, 361-362
 - selection style, 361
 - tables, 354
- centering views, constraints, 209-210
- centers, frames, 153
- `CFBundleURLTypes`, 487

CGAffineTransform, 159
CGPoint, 148
CGRect, 147
CGRectApplyAffineTransform, 148
CGRectDivide(), 148
CGRectEqualToRect, 148
CGRectFromString(aString), 147
CGRectGetMidX, 147
CGRectInset(aRect, xinset, yinset), 147
CGRectIntersectsRect, 148
CGRectOffset, 147
CGRects, 148
 frames, 153
CGRectZero, 148
CGSize, 148
 checked table cells, creating, 362-364
 checking
 for available disk space, 588
 network status, 521-524
 chevrons, 364
 child view controllers, adding/removing, 304
 circle layout, collection views, 425-426
 creation and deletion animation, 426
 powering, 426-427
 circles
 detecting, 29-34
 laying out views in, 428-431
 circular hit tests, 17
 Clang specification, number literals, 606
 classes, gesture recognizer subclasses, 35-36
 clear button, 227
 clipsToBounds flag, 152

code
 accessory views and stored state, 363-364
 accumulating user tracings for a composite drawing, 27-29
 activity view controller, 463-464
 adapting table edits to Core Data, 510-514
 adding action drawers to tables, 392-394
 adding custom buttons to keyboards, 229-230
 adding gestures to collection view layouts, 432-433
 adding keyboard input to nontext views, 241-243
 adding menus to interactive views, 46-47
 adding motion effects, 85
 adding UIViewAnimation blocks to controls, 60-61
 adding undo support and persistence to text views, 246-248
 adding universal support for split view alternatives, 284-285
 animating transitions with Core Animation, 172
 animating transparency changes to a view's alpha property, 167-168
 application activities, 466-469
 automatically copying text to the pasteboard, 454
 background transfers, 545
 basic collection view controller with flow layout, 412-416
 basic navigation drilling, 272-273
 basic popovers, 125-126
 basic size constraints, 207-208
 bouncing views, 172-174
 building a basic table, 358-360

- building a custom color control, 74-76
- building a discrete-valued star slider, 76-79
- building a draggable ribbon control, 86-88
- building a pull-to-refresh into your tables, 390
- building a sectioned table with Core Data, 503-505
- building a touch wheel control, 80-82
- building a UIButton that toggles on and off, 57-58
- building detail and master view for a split view controller, 280-283
- building dynamic slider thumbs, 64-67
- centering views with constraints, 209-210
- combining multiple view changes in animation blocks, 168-169
- comparing constraints, 202-203
- constrained movement, 15
- constraint macros, 219-221
- Core Data collection view, 516-519
- creating a custom input controller for a nontext view, 244-245
- creating a custom input view, 238-240
- creating a custom view controller segue, 311-314
- creating a dedicated keyboard spacer, 233-234
- creating a draggable view, 6-7
- creating a gesture recognizer subclass, 35-36
- creating a lock control, 89-91
- creating a page view controller wrapper, 298-303
- creating a segmented control subclass that responds to a second tap, 70
- creating a tab bar view controller, 287-290
- creating a touch feedback overlay view, 43-45
- creating a view controller container, 306-308
- creating aspect ratio constraints, 210-211
- creating blocks-based alerts, 108-110
- creating smoothed Bezier paths using Catmull-Rom splining, 23-25
- creating the illusion of a repeating cylinder, 398-400
- creating toolbars in code, 97-98
- custom alert, 121-124
- custom collection view cell menus, 441-442
- custom collection view cells, 417
- customizing the target content offset, 424-425
- describing constraints, 216-218
- detecting circles, 31-33
- detecting text patterns using predicates and regular expressions, 258-259
- to-do list view hierarchy, 137
- document interaction controllers, 477-480
- downloads with feedback, 535-542
- dragging items out of scroll views, 38-40
- editing tables, 370-373
- enhanced text editor, 251-252
- extending device information gathering, 563-564
- extracting a view hierarchy tree, 138
- grid layout customization, 434-439
- handling acceleration events, 567-568
- handling incoming documents, 483-486
- horizontal scroller collection view, 419-421
- image gallery viewer, 94-95

- interactive layout effects, 423-424
- JSON data, 547-548
- key frame animation, 175
- laying out views in a circle, 428-431
- monitoring connectivity changes, 525-526
- monitoring proximity and battery, 561-562
- naming views, 145
- playing sounds, alerts, and vibrations
 - using audio services, 132-133
- posting social updates, 348-349
- presenting and dismissing a modal controller, 276-278
- presenting and hiding a custom alert overlay, 118-119
- programmatically updating accessibility information, 592
- providing a circular hit test, 17
- providing URL scheme support, 488-489
- pull controls, testing touches, 85-88
- quick look, 472-473
- recognizing gestures in parallel, 10-12
- recording video, 332-333
- recovering file system size and file system free size, 588
- retrieving transform values, 160-164
- scheduling local notifications, 127-128
- searching for misspellings, 260-261
- selecting images, 323-326
- sending images by e-mail, 343-344
- sending texts, 345-346
- simple downloads, 530-533
- sliding an onscreen object based on accelerometer feedback, 573-575
- snapping pictures, 329-330
- spell checker protocol, 261
- storing tab state to user defaults, 291-293
- subview utility functions, 140-141
- supporting a table with sections, 379-380
- testing conformance, 450-451
- testing a network connection, 523-524
- testing touches against bitmap alpha levels, 18-19
- tilt scroller, 576-578
- touch-based painting in a UIView, 21-22
- trimming video with AV Foundation, 338-339
- UIViewFrame geometry category, 154-157
- updating view constraints, 213-214
- using a kqueue file monitor, 458-460
- using a pan gesture recognizer to drag views, 8-9
- using a variadic method for UIAlertView creation, 111
- using device motion updates to fix an image in space, 578-579
- using fetch requests with predicates, 505-508
- using search features, 386-387
- using the done key to dismiss a text field keyboard, 227-228
- using the video editor controller, 340-341
- using transitions with UIView animations, 170
- using UIImageView animation, 176
- utilizing text-to-speech in iOS 7, 601-602
- video playback, 335-336
- VIDEOkit, 584-587
- XMLParser helper class, 552-553

collection views, 403, 405-406

- adding gestures to layout, 431-433
- circle layout, 425-426
 - creation and deletion animation, 426
 - powering, 426-427
- controllers, 406
- Core Data, 514-519
- custom cells, 416-417
- data sources, 407
- delegates, 407
- flow layouts, 407, 412-416
 - header and footer sizing, 410
 - insets, 410
 - item size and line spacing, 408
 - scroll direction, 407
- grid layouts, creating, 433-439
- interactive layout effects, 422-424
- item menus, 440-442
- scroll snapping, 424-425
- scrolling horizontal lists, 418-421
- versus tables, 403-405
- views, 406-407

colorWithPatternImage: method, 165**common keys, 559****common types, storing, on pasteboards, 452-453****comparing constraints, 201-203****composite drawings, accumulating user tracings, 27-29****composition view controller, 347-349****compression resistance, 185****configurations, URL Loading System, 527****conformance, testing, 450-451****conformance lists, retrieving, 449-451****conformance trees, UTIs (Uniform Type Identifiers), 446****connecting buttons, to actions, 55****connections, format strings, 191-193****connectivity changes**

- monitoring, 525-526
- scanning for, 524-526

constrained movement, code, 15**constraining**

- movement, 14-15
- sizes, constraints, 206

constrainPosition:, 206**constraint multipliers, 210****constraints, 179-180**

- aligning views and flexible sizing, 198
- animating, 203
- aspect ratio, 210-211
- attributes, 180-181
- Auto Layout, 183
- centering views, 209-210
- comparing, 201-203
- constraining size, 206
- creating, 186
 - basic constraint declarations, 187
 - fixed-size constrained views, 204
 - variable bindings, 188-189
 - visual format constraints, 187-188

debugging, 214-215**describing, 215-218****disabling autoresizing constraints, 204-205****fixed-size constrained views, 206-208****format strings, 189, 196-197****connections, 191-193****orientation, 189-191****frames, 184****alignment rectangles, 185-186****intrinsic content size, 184-185****laws of, 182-184****macros, 218-221****managing, 199-201**

- math, 181
- orientation changes, 212-214
- predicates, 194-195
 - metrics, 195-196
 - priorities, 196
 - view-to-view predicates, 196
- priorities, 182
- processing, 198-199
- starting within view bounds, 205
- updating, 213-214

Contact Add button, 54

container literals, 607-608

containers, custom containers, 303

- adding/removing child view controllers, 304

content hugging, 185

contentMode property, 152

contentViewController property, 266

contexts, creating (Core Data), 494-495

continueTrackingWithTouch:withEvent:, 73

control events, UIControl class, 51-53

controllers

- activity view controller. *See* activity view controller
- collection views, 406
- composition view controller, 347-349
- document interaction controller. *See* document interaction controller
- image picker controllers
 - adding photos to simulators, 319
 - assets library module, 319-320
 - delegate callbacks, 321-322
 - image sources, 317-318
 - iPhone/iPad, 318
 - presenting pickers, 320-321
- Media Player, 333-336
- Message UI framework, 341-344

- navigation controllers. *See* navigation controllers

- Quick Look preview controller. *See* Quick Look preview controller

- Social framework, 347-349

- video editor controller, 340-341

controlling attributes, 249

controls

- attributes and, 69-70
- building, custom color controls, 74-76
- buttons. *See* buttons
- creating, UIControl class, 73
- custom lock controls, building, 88-92
- draggable ribbon controls, building, 86-88
- image picker controllers, selecting images, 323-326
- lock controls, creating, 89-91
- motion effects, adding, 85
- page indicator controls, adding, 92
- pull controls
 - creating, 83
 - discoverability, 84-85
- sliders, adding, 62-67
- star sliders, building, 76-79
- steppers, 70-72
- switches, 70-72
- touch wheels, building, 79-82
- twice-tappable segmented controls, creating, 67-70
- UIControl class, 49-50
 - control events, 51-53
 - target-actions, 49-50

converting XML into trees, 549-551

coordinate systems, view geometry, 149-150

copying text to pasteboards, 454

Core Animation transitions, 170-172

Core Data, 491

- adding data, 495-496
- adding edits to table views, 508, 510-514
 - undo/redo support, 508-509
 - undo transactions, 509-510
- collection views, 514-519
- contexts, creating, 494-495
- entities, 492
 - attributes and relationships, 493
 - building model files, 492-493
 - building object classes, 494
- examining, data files, 497-498
- model files, building, 492-493
- models, 492
- querying database, 498
 - fetch requests, 499
- removing, objects, 500-501
- search tables, 505-508
- table data sources, 501
 - index path access, 501
 - index titles, 502
 - section groups, 502
 - section key paths, 502
 - table readiness, 502-503

Core Data querying database, performing fetch requests, 499-500**Core Graphics, 62-64****Core Media, AV Foundation, 337-338****Core Motion, 565**

- monitoring accelerometer, 566
- testing, sensors, 565-566

counting sections and rows, 375-376**creation animation, circle layout (collection views), 426****curls, modal presentations, 274****current context style, 275****custom accessory views, dismissing text views, 228-230****custom alert overlays, 118-119****custom buttons, adding to keyboards, 229-230****custom cells, collection views, 416-417****custom color controls, building, 74-76****custom containers, 303**

- adding/removing child view controllers, 304

custom document types, creating, 481-482**custom gesture recognizers, creating, 34-36****custom group tables, 395****custom input view, creating, 235-240****custom input views to nontext views, adding to nontext views, 243****custom lock controls, building, 88-92****custom modal alert view, 119-120**

- frosted glass effect, 120-124

custom modal information view, 275-278**custom presentation style, 275****custom selection traits, adding to table view cells, 361-362****custom view controller segues, creating, 311-314****customizing**

- grids, collection views, 434-439
- headers and footers, sections, 377-378
- sliders, 62-64
- target content offsets, collection views, 424-425

D
data

- adding (Core Data), 495-496
- retrieving from system pasteboards, 453
- storing on pasteboards, 452

- data detectors, 257**
 - built-in type detectors, 257
- data files, examining (Core Data), 497-498**
- data source methods**
 - multiwheel tables, 397
 - tables, 357-358
- data sources**
 - assigning to tables, 354
 - collection views, 407
 - tables, 352
- databases (Core Data), querying, 498**
 - fetch requests, 499-500
- dataSource property, 354**
- date pickers, creating, 400**
- debugging constraints, 214-215**
- declaring**
 - alignment rectangles, 186
 - document support, 480-481
 - creating custom document types, 481-482
 - schemes, URL-based services, 487
- delays, audio alerts, 131**
- delegate callbacks, image picker controllers, 321-322**
- delegate methods**
 - multiwheel tables, 397
 - searching tables, 385
- delegates, 352**
 - alerts, 103-104
 - assigning, 356
 - collection views, 407
 - sections, 379
 - table views, 352-353
- delete requests, table edits, 369**
- deletion animation, circle layout (collection views), 426**
- dequeuing cells, 355-356**

- describing constraints, 215-218**
- Deselect button, 356**
- Detail Disclosure button, 53**
- detail views, split view controllers, 280-283**
- DetailViewController class, 280**
- detecting**
 - circles, 29-34
 - misspelling in UITextView, 260-261
 - retina support, 562-563
 - screens, 582
 - shakes, using motion events, 579-581
 - text patterns, 255
 - built-in type detectors, 257
 - data detectors, 257
 - enumerating regular expressions, 256-257
 - expressions, 255-256
 - predicates and regular expressions, 258-259
- device capability restrictions, adding, 556-557**
- device information**
 - accessing, 555-556
 - recovering, 563-564
- devices**
 - accelerometer-based scroll view, 575-578
 - attitude, 578-579
 - battery state, 560-562
 - calculating relative angle, 570-571
 - checking for available disk space, 588
 - common keys, 559
 - detecting
 - retina support, 562-563
 - screens, 582
 - external screens, 581
 - adding display links, 583
 - overscanning compensation, 583

- retrieving screen resolutions, 582
- video out, 583
- VIDEOkit, 584-587
- moving onscreen objects, with acceleration, 571-575
- orientation, 568-569
- proximity sensors, 559-560
- required device capabilities, 557-558
- tracking, users, 587-588
- user permissions, 558
- diacritics, 381**
- didAddSubview:, 142**
- didMoveToSuperview:, 142**
- didMoveToWindow, 142**
- direct manipulation**
 - adding interfaces, 5-7
 - multiple gesture recognizers, using simultaneously, 9-13
 - pan gesture recognizers, adding, 7-9
 - touches, 1-2
 - drawing onscreen, 20-22
 - gesture recognizers, 4-5
 - Multi-Touch, 4, 26-29
 - phases, 2-3
 - responder methods, 3-4
 - testing, 15-17
 - views, 4
- disabling autosizing constraints, 204-205**
- disclosure accessories, table views, 364-366**
- discoverability, pull controls, 84-85**
- disk space, checking for available disk space, 588**
- dismissing**
 - text views with custom accessory views, 228-230
 - UITextField keyboard, 224-225
 - text trait properties, 225-228

- dispatching events, UIControl class, 73-74**
- display links, adding, 583**
- display traits, 164-165**
- displaying**
 - alerts, 104-105
 - images, image picker controllers, 328
 - remove controls, table edits, 368
 - text, in action sheets, 114-115
- disposing of system sounds, 132**
- document interaction controller, 473-474**
 - checking, for open menu, 476-477
 - code for, 477-480
 - creating instances, 475
 - properties, 475-476
 - providing Quick Look support, 476
- document support**
 - declaring, 480-481
 - creating custom document types, 481-482
 - implementing, 483
- document types, creating, 481-482**
- documents**
 - document interaction controller. *See* document interaction controller
 - handling incoming, 483-486
 - Quick Look preview controller. *See* Quick Look preview controller
 - scanning for, 456-457
- Documents folder, monitoring, 454-455**
 - file sharing, 455
 - scanning for new documents, 456-457
 - user control, 455-456
 - Xcode, 456
- double-tap gesture, 440**
- doubleSided property, 294**

downloads

- with feedback, 533-542

- running, 543

- simple downloads, 528-533

draggable ribbon controls, building, 86-88**draggable views, creating, 6-7****dragging, from scroll view, 37-40****DragView, 6****drawing touches, onscreen, 20-22****drawings**

- composite drawings, accumulating user tracings, 27-29

- smoothing, 22-25

dynamic sliders, building, 64-67**Dynamic Type, 602-604****Dynamics, 120**

E

e-mailing pictures, Message UI framework, 341-344**edge-to-edge layout, navigation controllers, 269-271****editing video, 336-341****efficiency, adding to sliders, 64****enablesReturnKeyAutomatically, 226****endTrackingWithTouch:withEvent:, 73****entities, Core Data, 492**

- attributes and relationships, 493

- building model files, 492-493

- building object classes, 494

Entity editor, 493**enumerating regular expressions, 256-257****enums, 607****establishMotionManager, 572****events, dispatching (UIControl class), 73-74****examining, data files (Core Data), 497-498****exceptions, 608****excluding activities, 470****exclusive touch, views, 4****expressions**

- detecting text patterns, 255-256

- enumerating, 256-257

- regular expressions, resources for, 258

extensions, UTIs (Uniform Type Identifiers), 447-448**external screens, 581**

- adding display links, 583

- detecting, 582

- overscanning compensation, 583

- retrieving, screen resolutions, 582

- video out, 583

- VIDEOkit, 584-587

extracting view hierarchy trees, 138

F

fades, modal presentations, 274**fading in and out, views, 167-168****feature tests, 609****feedback**

- downloads, 533-542

- second-tap feedback, 68

fetch requests

- Core Data, 499

- performing, Core Data, 499-500

- predicates, 505-508

file sharing

- Documents folder, 455

- Xcode, 456

filtering, text entry, 252-255**finding views with tags, 143****fixed-size constrained views, 206-208**

- creating, 204

flexible sizing, aligning with constraints, 198

- flipping, views, 169-170**
- flips, modal presentations, 274**
- flow layouts, collection views, 407, 412-416**
 - header and footer sizing, 410
 - insets, 410
 - item size and line spacing, 408
 - scroll direction, 407
- footer sizing, flow layouts (collection views), 410**
- footers, customizing in sections, 377-378**
- form sheet style, 275**
- format strings, 189, 196-198**
 - connections, 191-193
 - orientation, 189-191
- forwarding touch events, 41-43**
- frames**
 - constraints, 184
 - alignment rectangles, 185-186
 - intrinsic content size, 184-185
 - view geometry, 147
 - views, 150-151
 - centers, 153
 - CGRects, 153
- frames views, adjusting sizes, 151-152**
- frosted glass effect, 120-124**
- full-screen presentation, 274**

G

- geometry, views, 146, 154-157**
 - coordinate systems, 149-150
 - frames, 147
 - points and sizes, 148
 - rectangle utility functions, 147-148
 - transforms, 149
- gesture recognizer subclasses, creating, 35-36**

- gesture recognizers, 4-5**
 - creating custom, 34-36
 - using multiple gesture recognizers simultaneously, 9-13

- gestures**
 - adding to layout, collection views, 431-433
 - double-tap gesture, 440
 - recognizing in parallel, 10-12
 - resolving conflicts, 13
 - VoiceOver, 600-601

- GitHub, xx**

- gray disclosure indicators, 366**

- grid layouts, collection views, 433-439**

- grids, customizing (collection views), 434-439**

- grouped preferences tables, creating, 395-396**

- GUIs, navigation controllers, 266**

H

- handlePan: method, 8**

- handler method, adding to URL-based services, 488**

- Hartstein, Greg, 428**

- header sizing, flow layouts (collection views), 410**

- header titles, sections, 377**

- headers, customizing in sections, 377-378**

- hierarchies**

- view hierarchy trees, recovering, 137-139
 - views, 135-137

- HIG (Human Interface Guidelines), 53**

- hints, accessibility, 596**

- Holleman, Matthijs, 84**

- horizontal lists, scrolling (collection views), 418-421**

Hosgrove, Alex, 37

Human Interface Guidelines (HIG), 53

I

IB (Interface Builder)

accessibility, 592

segues, 314

IB Identity Inspector, accessibility, 592

icons, document interaction controller, 475

identifierForVendor property, 588

image gallery viewer, 93-95

image picker controllers, 317

adding photos to simulators, 319

assets library module, 319-320

delegate callbacks, 321-322

displaying images, 328

image sources, 317-318

iPhone/iPad, 318

presenting pickers, 320-321

recording video, 331-333

saving images, 329

selecting images, 323-326

snapping photos, 326-330

video

editing, 336-339

picking and editing, 339-341

recording, 332-333

saving, 332

image sources, image picker controllers, 317-318

image view animations, 176

imageData, 514

images

displaying, image picker controllers, 328

saving, image picker controllers, 329

selecting, image picker controllers, 323-326

implementing

document support, 483

page view controllers, 295-296

Quick Look preview controller, 471-472

tables, 356

data source methods, 357-358

responding to user touches, 358

UIKit overlay view, 43

undo, table edits, 367

Inbox (iTunes), 456

incoming, documents, handling, 483-486

index path access, Core Data (table data sources), 501

index titles, Core Data (table data sources), 502

indexes

presentation indexes, page view controllers, 297-298

search-aware index, 385-386

indicators

alert indicators, 128-129

badging applications, 129

showing progress, 115

Info Dark button, 53

Info Light button, 53

inheritance, UTIs (Uniform Type Identifiers), 446

input clicks, adding, 243

inputView property, 235

insertSubview:aboveSubview:, 141

insertSubview:atIndex:, 141

insertSubview:belowSubview:, 141

insets, flow layouts, collection views, 410

instances, creating for document interaction controllers, 475

interaction traits, 164-165

interactive layout effects, collection views, 422-424

intercepting touch events, 41-43

Interface Builder, 355

 buttons, 55

 naming views, 144-145

interfaces, adding simple direct manipulation interfaces, 5-7

internationalizing applications, constraint attributes, 180

intrinsic content size, frames, constraints, 184-185

iOS tables, 351-352

 testing accessibility, 599-601

iPad, image picker controllers, 318

iPhone, image picker controllers, 318

iPhone-style navigation controllers, 267

item menus, collection views, 440-442

item providers, activity view controller, 462

item size, flow layouts, collection views, 408

item source callbacks, activity view controller, 462-463

items, activity view controller, 469-470

J-K

JSON serialization, 546-548

key frame animations, views, 174-175

keyboard dismissal, preventing, 225

keyboardAppearance, 226

keyboards

 adding custom buttons to, 229-230

 adding input to nontext views, 241-243

 adjusting views, 230-234

 creating dedicated keyboard spacers, 233-234

keyboardType, 226

kqueue file monitor, 458-460

kSCNetworkReachabilityFlagsConnectionOnTraffic, 522

kSCNetworkReachabilityFlagsIsDirect, 522

kSCNetworkReachabilityFlagsIsWWAN, 522

kUTTypeConformsToKey, 449

kUTTypeDescriptionKey, 449

kUTTypeIdentifierKey, 449

kUTTypeTagSpecificationKey, 449

L

labels, accessibility, 595-596

laws of constraints, 182-184

laying out views in circles, 428-431

line spacing, flow layouts (collection views), 408

List Items, 465

lists of options, alerts, 112-114

literals, 605

 array literals, 608

 boxed expressions, 606

 enums, 607

 container literals, 607-608

 feature tests, 609

 number literals, 605-606

 subscripting, 608

local notifications, 126-127

 best practices, 127-128

 scheduling, 127-128

lock controls, creating, 89-91

long presses, 5

M

macros

 constraints, 218-221

 navigation item class, 272

managing, constraints, 199-201

master view controller, 280

master views, split view controllers, 280-283

math, constraints, 181

Media Player, playing video, 333-336

menu support, item menus, 440

menus, 112

adding to views, 45-47

scrolling, 114

showFromBarButtonItem:animated,
112

showFromRect:inView:animated, 112

showFromTabBar, 112

showFromToolBar, 112

showInView, 112

message contents, creating, 342-343

**Message UI framework, e-mailing pictures,
341-344**

metrics, predicates, 195-196

metrics dictionary, 188

**MFMessageComposeViewController-
Delegate protocol, 344**

**MIME helper, UTIs (Uniform Type
Identifiers), 448**

MIME type

message contents, 342-343

UTIs (Uniform Type Identifiers),
447-448

minwidth, 196

mismatches, sections, 378-379

**misspelling in UITextView, detecting,
260-261**

modal controllers, code, 276-278

modal presentations

custom modal information view,
275-278

navigation controllers, 273-275

modal progress overlays, 117-119

modalPresentationStyle property, 274-275

model, 556

model files, building (Core Data), 492-493

Model-View-Controller (MVC), 352

models, Core Data, 492

**modules for system frameworks, audio
alerts, 130**

monitoring

acceleration, 566-568

battery state, 560-562

connectivity changes, 525-526

Documents folder, 454-455

file sharing, 455

scanning for new documents,
456-457

user control, 455-456

Xcode, 456

motion effects, adding, 85

motion events, detecting shakes, 579-581

movement, constraining, 14-15

moving

onscreen objects with acceleration,
571-575

UTIs (Uniform Type Identifiers) to
extensions or MIME types, 447-448

Multi-Touch, 4, 26-29

multiline button text, 59

**multiple gesture recognizers, using simulta-
neously, 9-13**

multiwheel tables, 396-397

data source and delegate methods, 397

picker views, 397-398

UIPickerView, 397

MVC (Model-View-Controller), 352, 491

myView.alpha property, 164

N

name, 556

document interaction controller, 475

naming views

in Interface Builder, 144-145

by object association, 143-144

navigation apps, creating, 283-285**navigation controllers, 264-265**

bar buttons, 269

custom containers, 303

adding/removing child view controllers, 304

edge-to-edge layout, 269-271

modal presentations, 273-275

custom modal information view, 275-278

navigation drilling, 272-273

navigation item class, 271

macros, 272

titles and back buttons, 271

page view controllers, 265, 293, 296-297

book properties, 294-295

implementing, 295-296

presentation indexes, 297-298

popover controllers, 266

segues, 309-314

split view controllers, 265

creating, 278-283

split views, 266-267

stacks, 268

tab bar controllers, 286-290

tab state, 290-293

universal split view/navigation apps, creating, 283-285

view controllers

pushing and popping, 268-269

transitioning between, 304-308

navigation item class, 271

macros, 272

titles and back buttons, 271

navigationOrientation property, 295**network connections, testing, 523-524****network status, checking, 521-524****network timeouts, 527**

networkAvailable method, 522

nontext views, adding custom input views, 243

notifications, local notifications, 126-127

best practices, 127-128

NSAttributedString, 469**NSDataDetector class, 257****NSDictionary, 469****NSDictionaryOfVariableBindings() macro, 189****NSEntityDescription, 495****NSFontAttributeName, 70****NSForegroundColorAttributeName, 70****NSJSONSerialization, 546****NSNotFound, 262****NSNumber, 605-606****NSOperationQueue, 522****NSProgress, 534****NSRegularExpression class, 256****NSShadowAttributeName, 70****NSSQLiteStoreType, 495****NSString, 469****NSStringFromCGRect, 147****NSUnderlineStyleAttribute Name, 70****NSURL, 469****NSURLSession, 528****NSURLSessionConfiguration object, 527****NSURLSessionDownloadTask, 543****NSURLSessionTask, 527****number literals, 605-606****numberOfSectionsInTableView, 357, 375, 395**

O

object association, naming views, 143-144

object classes, building (Core Data), 494

objects, removing (Core Data), 500-501

onscreen objects, moving with acceleration, 571-575

Open in options, 476-477

open menus, checking for in document interaction controller, 476-477

open-source llamasettings project, 396

orientation

calculating, from accelerometer, 569-570

devices, 568-569

format strings, 189-191

orientation changes, constraints, 212-214

overlays

modal progress overlays, 117-119

tappable overlays, 119

overscanning compensation, 583

P

page indicator controls, adding, 92

page sheet style, 274

page view controllers, 265, 293, 296-297

book properties, 294-295

implementing, 295-296

presentation indexes, 297-298

wrappers, creating, 298-303

pan gesture recognizers, adding, 7-9

pans, 5

parse trees, building, 551-553

pasteboards. *See* system pasteboards

performing fetch requests, Core Data, 499-500

permissions, user permissions, 558

persistence, adding to text views, 246-248

phases, touches, 2-3

photos

adding to simulators, 319

snapping, image picker controllers, 326-330

picker views, multiwheel tables, 397-398

pickers, presentation indexes (image picker controllers), 320-321

picking video, 339-341

pictures, e-mailing (Message UI framework), 341-344

pinches, 5

placeholders, 226

playing video with Media Player, 333-336

points, view geometry, 148

popover controllers, 266

popoverArrowDirection property, 125

popovers, 124-126

posting social updates, 347-349

powering, circle layout (collection views), 426-427

predicates, 194-195

detecting data patterns, 258-259

fetch requests, 505-508

metrics, 195-196

priorities, 196

view-to-view predicates, 196

prepareWithActivityItems, 465

presentation indexes, page view controllers, 297-298

presenting

activity view controller, 461

pickers, image picker controllers, 320-321

preventing keyboard dismissal, 225

priorities

constraints, 182

predicates, 196

processing constraints, 198-199

progress, showing, 115

UIActivityIndicatorView, 116

UIProgressView, 116

properties

- accessoryType property, 363
- document interaction controller, 475-476
- inputView property, 235
- scrollDirection property, 407
- text trait properties, 225-228
- transforms, retrieving, 158-159

providing

- Quick Look support, document interaction controller, 476
- URL scheme support, 488-489

proximity sensors, devices, 559-560**pull controls**

- creating, 83
- discoverability, 84-85
- testing, touches, 85-88

pull-to-refresh, adding to tables, 387-390**pushing view controllers, 268-269**

Q

QLPreviewController, 349**querying**

- databases (Core Data), 498
 - fetch requests, 499
 - performing fetch requests, 499-500
- subviews, 139-141

Quick Look, providing support for document interaction controllers, 476**Quick Look preview controller, 470-471**

- code, 472-473
- implementing, 471-472

R

reachabilityChanged, 524**recording video, image picker controllers, 331-333****recovering**

- device information, 563-564
- view hierarchy trees, 137-139

rectangle utility functions, view geometry, 147-148**refresh controls, activating, 388****registering**

- cell classes, 355
- cells for search display controllers, 383

regular expressions

- detecting data patterns, 258-259
- resources for, 258

relationships, Core Data, 493**relative angle, calculating, 570-571****remove controls, displaying in table edits, 368****removing**

- child view controllers, 304
- objects, Core Data, 500-501
- subviews, 141-142

reordering

- cells, 369
- subviews, 141-142

repeating cylinders, creating illusion of, 398-400**resolving, gesture conflicts, 13****resources for regular expressions, 258****responder methods, touches, 3-4****responders, text editors, 250****responding to user touches, 358****restrictions, adding device capability restrictions, 556-557****retain cycles, blocks, 107-108****retina support, detecting, 562-563****retrieving**

- conformance lists, UTIs (Uniform Type Identifiers), 449-451
- data, system pasteboards, 453

- device attitude, 578-579
- screen resolutions, 582
- transform information, 158
 - properties, 158-159
 - testing for view intersection, 159-164

returning cells, sections, 376-377

returnKeyType, 226

root view controllers, 268

rotations, 5

rows, counting, 375-376

running downloads, 543

S

saving

- images, image picker controllers, 329
- video, image picker controllers, 332

scanning

- for connectivity changes, 524-526
- for new documents, 456-457

scheduling, local notifications, 127-128

schemes

- declaring, URL-based services, 487
- URL scheme support, providing, 488-489

SCNetworkReachabilityGetFlags, 522

screen resolutions, retrieving, 582

screens

- detecting, 582
- external screens, 581
 - adding display links, 583
 - overscanning compensation, 583
 - retrieving screen resolutions, 582
 - video out, 583
 - VIDEOkit, 584-587

scroll direction, flow layouts (collection views), 407

scroll snapping, collection views, 424-425

scroll view, dragging from, 37-40

scrollDirection property, 407

scrolling

- horizontal lists, collection views, 418-421
- menus, 114

search-aware index, 385-386

search bars, 381

search display controllers

- creating, 382-383
- registering cells, 383

search features, 386-387

search tables, Core Data, 505-508

searchable data source methods, building, 383-385

searching

- tables, 381
 - building searchable data source methods, 383-385
 - creating search display controllers, 382-383
 - delegate methods, 385
 - registering cells for the search display controller, 383
 - search-aware index, 385-386
 - for text strings, 262

second-tap feedback, 68

section groups, Core Data (table data sources), 502

section index, creating, 378

section key paths, Core Data (table data sources), 502

sectioned tables, creating with Core Data, 503-505

sectionIndexTitlesForTableView, 378

sections, 374

- counting, 375-376
- creating, 374-375
 - section index, 378
- customizing headers and footers, 377-378
- delegates, 379
- header titles, 377
- mismatches, 378-379
- returning cells, 376-377
- supporting tables with sections, code, 379-380

secureTextEntry, 226**segmented control subclasses, second taps, 70****segues, 309-314**

- IB, 314

selecting images, image picker controllers, 323-326**selection style, table view cells, 361****sending**

- text messages, 344-346
- texts, 345-346

sensor data, accessing, 566**sensors**

- proximity sensors, devices, 559-560
- testing, Core Motion, 565-566

services, adding (activity view controller), 464-465**setBarButtonItems method, 367****shakes, detecting using motion events, 579-581****sharing data, system pasteboards, 451-452****showFromRect:inView:animated, 112****showFromTabBar, 112****showFromToolBar, 112****showing progress, 115**

- UIActivityIndicatorView, 116
- UIProgressView, 116

showInView, 112**shutdownMotionManager, 572****simple alerts, building, 101-102****simple downloads, 528-533****simulators**

- adding photos to, 319
- testing, accessibility, 597-598

sizes

- adjusting frames, 151-152
- constraining, constraints, 206
- view geometry, 148

sliders

- adding, 62-67
- building, 64-67
- customizing, 62-64
- efficiency, adding, 64
- star sliders, building, 76-79

slides, modal presentations, 274**smoothing, drawings, 22-25****snapping photos, image picker controllers, 326-330****Social framework, 347-349****social updates, posting, 347-349****speech synthesis, 600-601****spell checker protocol, 261****spellCheckingType, 225****spineLocation property, 294****splining, Catmull-Rom (creating smoothed Bezier paths), 23-25****split view alternatives, adding universal support for, 284-285****split view controllers, 265**

- creating, 278-283
- detail views, 280-283
- master views, 280-283

split views, navigation controllers, 266-267**stacks, navigation controllers, 268**

- star sliders, building, 76-79
- starting within view bounds, constraints, within view bounds, 205
- steppers, 70-72
- stored state, 363-364
- storing
 - common types, on pasteboards, 452-453
 - data, system pasteboards, 452
 - tab states to user defaults, 291-293
- storyboard views, XIB, 139
- storyboards, 314
- styles, table styles, 353
- subclassing, UIControl class, 72-76
- subscripting, 608
- subview utility functions, 140-141
- subviews, 135
 - adding, 141
 - querying, 139-141
 - removing, 141-142
 - reordering, 141-142
- subviews property, 139
- suffixes, for number literals, 606
- swapping views, 168-169
- swipes, 5
- swiping cells, 369
- switches, 70-72
- sysctl(), 563
- sysctlbymname(), 563
- System button, 54
- System Configuration framework, 522
- system pasteboards
 - accessing, 451-452
 - copying text to, 454
 - retrieving data, 453

- storing
 - common types, 452-453
 - data, 452
 - updating, 453-454
- system sounds, 129-130
 - disposing of, 132
- systemName, 555
- systemVersion, 555

T

- tab bar controllers, 265, 286-290
 - creating, 287-290
- tab state, 290-293
 - storing to user default, 291-293
- table data sources, Core Data, 501
 - index path access, 501
 - index titles, 502
 - section groups, 502
 - section key paths, 502
 - table readiness, 502-503
- table edits, 366-367
 - adding
 - cells, 369
 - undo support, 367
 - code, 370-373
 - delete requests, 369
 - displaying remove controls, 368
 - implementing undo, 367
 - reordering cells, 369
 - swiping cells, 369
- table readiness, Core Data (table data sources), 502-503
- table styles, 353
- table view cells, 360
 - accessibility, 593
 - adding custom selection traits, 361-362
 - selection style, 361

table views, 353-354

- Core Data, 508
- delegates, 352-353
- iOS tables, 351-352
- sections. *See* sections

table views disclosure accessories, 364-366**tables**

- adding
 - action rows, 390-394
 - pull-to-refresh, 387-390
- versus collection views, 403-405
- Core Data
 - table views, 510-514
 - undo/redo support, 508-509
 - undo transactions, 509-510
- creating, 353
 - assigning data sources, 354
 - assigning delegates, 356
 - building basic tables, 358-360
 - cells, 354
 - dequeuing cells, 355-356
 - registering cell classes, 355
 - table styles, 353
 - views, 353-354
- creating checked table cells, 362-364
- custom group tables, 395
- Deselect button, 356
- grouped preferences tables, creating, 395-396
- implementing, 356
 - data source methods, 357-358
 - responding to user touches, 358
- iOS tables, 351-352
- multiwheel tables, 396-397
 - data source and delegate methods, 397
 - picker views, 397-398
 - UIPickerView, 397

search tables, Core Data, 505-508

searching, 381

- building searchable data source methods, 383-385
- creating search display controllers, 382-383
- delegate methods, 385
- registering cells for the search display controller, 383
- search-aware index, 385-386
- section tables, creating with Core Data, 503-505
- table view cells, 360
 - adding custom selection traits, 361-362
 - selection style, 361
- UIDatePicker, 400
 - creating, 400

tableView:cellForRowAtIndexPath:, 358, 395

tableView:didSelectRowAtIndexPath, 395

tableView:heightForRowAtIndexPath, 395

tableView:numberOfRowsInSection:, 357, 376, 395

tableView:titleForHeaderInSection, 395

tagging views, 142-143

tags, finding views, 143

tappable overlays, 119

taps, 5

target-actions, UIControl class, 49-50

tasks, URL Loading System, 527-528

testing

- accessibility
 - on iOS, 599-601
 - with simulators, 597-598
- background transfers, 544-545
- against bitmaps, 17-19

conformance, 450-451

UTIs (Uniform Type Identifiers),
448-449

network connections, 523-524

sensors, Core Motion, 565-566

touches, 15-17

against bitmap alpha levels, 18-19

pull controls, 85-88

URLs, 488

for view intersection, transforms,
159-164

tests, circular hit tests, 17

text

copying, 454

displaying in action sheets, 114-115

text editors

attributed text, 249

attributes, controlling, 249

building, 246-248

responders, 250

text entry, 223

adding

custom input views to nontext
views, 243

input clicks, 243

building text editors, 246-252

creating, custom input view, 235-240

detecting

misspelling in UITextView, 260-261

text patterns, 255

dismissing UITextField keyboard,
224-225

filtering, 252-255

keyboards. *See* keyboards

text-input-aware views, 240-243

text strings, searching for, 262

text-input-aware views, 240-243

Text Kit, 223, 258

text messages, sending, 344-346

text patterns, detecting, 255

built-in type detectors, 257

data detectors, 257

enumerating regular expressions,
256-257

expressions, 255-256

predicates and regular expressions,
258-259

text strings, searching for, 262

text trait properties, 225-228

text views, 246

dismissing, with custom accessory
views, 228-230

persistence, adding, 246-248

undo support, adding, 246-248

textFieldAtIndex: method, 105

textFieldShouldReturn:, 225

texts, sending, 345-346

tintColor property, 269

titles, navigation item class, 271

to-do list view hierarchy, 137

toolbars

building, 96-98

creating, code, 97-98

topLayoutGuide, 270

touch-based painting, UIView, 21-22

**touch events, intercepting/forwarding,
41-43**

touch feedback, 40

enabling, 41

overlay view, creating, 43-45

touch events, intercepting/forwarding,
41-43

touch wheels, building, 79-82

touches, 1-2

dragging from scroll view, 37-40

drawing, onscreen, 20-22

- feedback
 - enabling, 41
 - intercepting/forwarding, 41-43
 - TOUCHkit overlay view, 43
- gesture recognizers, 4-5
 - creating custom, 34-36
- Multi-Touch, 4, 26-29
- phases, 2-3
- responder methods, 3-4
- testing, 15-17
 - against bitmap alpha levels, 18-19
 - pull controls, 85-88
- tracking, UIControl instances, 73
- views, 4
- touchesBegan:withEvent:, 3**
- touchesCancelled:WithEvent:, 3**
- touchesEnded:withEvent:, 3**
- touchesMoved:withEvent:, 3**
- TOUCHkit, 40**
 - implementing, 43
- TOUCHOverlayWindow class, 41**
- tracking
 - touches, UIControl instances, 73
 - users, 587-588
- traits, accessibility, 594-595**
- transfers, background transfers, 543-544**
 - testing, 544-545
 - web services, 546
- transforms**
 - defined, 158
 - retrieving information, 158
 - properties, 158-159
 - testing for view intersection, 159-164
 - view geometry, 149
- transitioning between view controllers, 304-308**

transitions

- Core Animation transitions, 170-172
- flipping views, 169-170

transitionStyle property, 294

trees, 550-551

- converting XML into, 549-551
- parse trees, building, 551-553

trimming video with AV Foundation, 338-339

twice-tappable segmented controls, creating, 67-70

U

UIAccessibility protocol, 592

UIAccessibilityLayoutChangeNotification, 599

UIAccessibilityPageScrolledNotification, 599

UIAccessibilityTraitAdjustable, 595

UIAccessibilityTraitAllowsDirectInteraction, 595

UIAccessibilityTraitButton, 594

UIAccessibilityTraitCausesPageTurn, 595

UIAccessibilityTraitHeader, 594

UIAccessibilityTraitImage, 594

UIAccessibilityTraitKeyboardKey, 594

UIAccessibilityTraitLink, 594

UIAccessibilityTraitNone, 594

UIAccessibilityTraitNotEnabled, 594

UIAccessibilityTraitPlaysSound, 595

UIAccessibilityTraitSearchField, 594

UIAccessibilityTraitSelected, 594

UIAccessibilityTraitStartsMediaSession, 595

UIAccessibilityTraitStaticText, 594

UIAccessibilityTraitSummaryElement, 595

UIAccessibilityTraitUpdatesFrequently, 595

UIAccessibilityZoomFocusChange, 599

UIActionSheet, 101

UIActionSheet instances, 112

- UIActivityIndicatorView, 115
 - modal view, 117
 - showing progress, 116
- UIActivityItemProvider, 462
- UIActivityItemSource, 462
- UIAlertView, 101-102
 - variadic methods, 111
- UIAlertViewStyleLoginAndPasswordInput, 104
- UIAlertViewStylePlainTextInput, 104
- UIAlertViewStyleSecureTextInput, 104
- UIAppearance protocol, 286
- UIAppFonts, 559
- UIApplication property, 128
- UIApplicationExitsOnSuspend, 559
- UIBarButtonItem, 96-98
- UIBarButtonItemSystemItemFlexibleSpace, 97
- UIButton instances, 53-54
- UIButtonTypeCustom, 54, 56
- UICollectionView instances, 403
- UICollectionViewFlowLayout, 406
- UIColor, 469
- UIControl class, 49
 - control events, 51-53
 - dispatching events, 73-74
 - kinds of controls, 50
 - subclassing, 72-76
 - target-actions, 49-50
- UIControl event types, 52
- UIControl instances, tracking touches, 73
- UIControlEventsValueChanged, 73
- UIDatePicker, 400
 - creating, 400
- UIDevice class, 555-556
- UIDocumentInteractionController, 473
- UIFileSharingEnabled, 559
- UIImage, 56, 469
- UIImagePickerController class, cameras, 328
- UIImagePickerControllerCropRect, 322
- UIImagePickerControllerEditedImage, 322
- UIImagePickerControllerMediaMetadata, 322
- UIImagePickerControllerMediaType, 322, 337
- UIImagePickerControllerMediaURL, 337
- UIImagePickerControllerOriginalImage, 322
- UIImagePickerControllerReferenceURL, 322
- UIImagePickerControllers class, 317
- UIImagePickerControllerSourceType-
Cameras, 318
- UIImagePickerControllerSourceTypePhoto
Library, 317
- UIImagePickerControllerSourceTypeSaved-
PhotosAlbum, 317
- _UIImagePickerControllerVideoEditingEnd, 337
- _UIImagePickerControllerVideoEditingStart, 337
- UIImageView, animation, 176
- UIKeyInput protocol, 241
- UIKit Dynamics, 14
- UIModalTransitionStyleCrossDissolve, 274
- UIModalTransitionStyleFlipHorizontal, 274
- UIModalTransitionStylePartialCurl, 274
- UINavigationController, hierarchies, 136
- UIPageControl class, 92
- UIPasteboard, 451
- UIPickerView, 396, 397
- UIPopoverController, 125
- UIPrintFormatter, 469
- UIPrintInfo, 469
- UIPrintPageRenderer, 469
- UIProgressView, 115, 533
 - showing progress, 116

- `UIRequiredDeviceCapabilities`, 556
- `UIRequiresPersistentWifi`, 559
- `UIResponder` class, 3
- `UIScreen`, 562-563, 582
- `UIScrollView` instance, 93
- `UISegmentedControl` class, 67
- `UISlider`, 62
- `UISplitViewController`, 264, 265
- `UIStatusBarHidden`, 559
- `UIStatusBarStyle`, 559
- `UIStepper` class, 71
- `UISwitch` instances, 71
- `UISwitch` object, 71
- `UITabBarController` class, 265
- `UITabBarControllerDelegate` protocol, 290
- `UITableView` delegate method, 352
- `UITableViewCellAccessoryDetailDisclosureButton`, 365
- `UITableViewCellAccessoryDisclosureIndicator`, 365
- `UITableViewCellStyleDefault`, 360
- `UITableViewCellStyleSubtitle`, 361
- `UITableViewCellStyleValue1`, 360
- `UITableViewCellStyleValue2`, 361
- `UITableViewControllers`, 354
- `UITextChecker`, 261
- `UITextField` keyboard, dismissing, 224-225
 - text trait properties, 225-228
- `UITextView`, detecting misspelling, 260-261
- `UITouch` objects, 10
- `UITouchPhaseBegan`, 2
- `UITouchPhaseCancelled`, 3
- `UITouchPhaseEnded`, 3
- `UITouchPhaseMoved`, 2
- `UITouchPhaseStationary`, 2
- `UIView` animations, 165-166
 - building with blocks, 166-167
 - transitions, 170
- `UIView` class
 - adding animation blocks to controls, 60-61
 - touch-based painting, 21-22
 - touches, 2
- `UIViewContentModeScaleAspectFill` mode, 328
- `UIViewContentModeScaleAspectFit` mode, 328
- `UIViewController` class, 269-271
 - view controllers, 264
- `UIViewFrame` geometry category, 154-157
- undo, implementing in table edits, 367
- undo/redo support, Core Data table views, 508-509
- undo support, adding
 - to table edits, 367
 - to text views, 246-248
- undo transactions, Core Data table views, 509-510
- Uniform Type Identifiers. See UTIs (Uniform Type Identifiers)
- universal split view/navigation apps, creating, 283-285
- updates, broadcasting, accessibility, 599
- `updateTransformWithOffset:` method, 10
- updating
 - system pasteboards, 453-454
 - view constraints, 213-214
- URL, 475
 - testing, 488
- URL-based services
 - adding handler method, 488
 - creating, 486-487
 - declaring schemes, 487
 - testing URLs, 488

URL Loading System, 526

configurations, 527

NSURLSession, 528

tasks, 527-528

URL scheme support, providing, 488-489**user control, Documents folder, 455-456****user permissions, 558****user touches, responding to user touches, 358****user tracings, accumulating for composite drawings, 27-29****userInterfaceldiom, 556****users, tracking, 587-588****UTI declarations, 449****UTIs (Uniform Type Identifiers), 445-446, 475**

conformance lists, retrieving, 449-451

conformance trees, 446

file extensions, 446-447

inheritance, 446

MIME helper, 448

moving to extensions or MIME types, 447-448

testing conformance, 448-449

UTTypeCopyPreferredTagWithClass(), 447**UUIDs (Universally Unique Identifiers), 588**

V

variable bindings, 188-189**variadic arguments, alerts, 110-111****verbose logging, 180****vibration, 130-131****video**

editing, 336-339

picking and editing, 339-341

playing with Media Player, 333-336

recording, image picker controllers, 331-333

saving, image picker controllers, 332

trimming with AV Foundation, 338-339

video editor controller, 340-341**video out, external screens, 583****video-recording picker, creating, 331-332****VIDEOkit, 584-587****view bounds, starting constraints, 205****view controller containers, creating, 306-308****view controllers, 263-264**

navigation controllers, 264-265

pushing and popping, 268-269

segues, 309-314

tab bar controllers, 265

transitioning between, 304-308

UIViewController class, 264

view hierarchy trees

extracting, 138

recovering, 137-139

view intersection, testing for, 159-164**view-to-view predicates, 196****viewDidLoad method, 368, 391****views**

accelerometer-based scroll view, 575-578

adding menus to, 45-47

adjusting around keyboards, 230-234

aligning with constraints, 198

bouncing, 172-174

callbacks, 142

centering in constraints, 209-210

collection views, 406-407

Core Animation transitions, 170-172

display traits, 164-165

exclusive touch, 4

fading in and out, 167-168

finding with tags, 143

flipping, 169-170

frames, 150-151

adjusting sizes, 151-152

centers, 153

CGRects, 153

geometry, 146, 154-157

coordinate systems, 149-150

frames, 147

points and sizes, 148

rectangle utility functions, 147-148

transforms, 149

hierarchies, 135-137

image view animations, 176

interaction traits, 164-165

key frame animations, 174-175

laying out in circles, 428-431

naming

in Interface Builder, 144-145

by object association, 143-144

swapping, 168-169

tables, 353-354

tagging, 142-143

text-input-aware views, 240-243

touching, 4

views: parameter, 188

visual format constraints, creating, 187-188

VoiceOver, 591

gestures for apps, 600-601

speech synthesis, 601

testing accessibility, 599-601

UIAccessibilityLayoutChange-
Notification, 599

VoiceOver toggle, 599

W

web services, background transfers, 546

converting XML into trees, 549-551

JSON serialization, 546-548

willMoveToSuperview:, 142

willMoveToWindow:, 142

willRemoveSubview, 142

wrappers, page view controllers, 298-303

X

.xcdatamodeld files, 492

Xcode

file sharing, 456

verbose logging, 180

Xcode5, accessibility, 597

XIB, storyboard views, 139

XML, converting into trees, 549-551

XMLParser class, 551-553

XMLParser helper class, 551-553

Y-Z

y coordinate, 149