

Erica Sadun



Third Edition

Covers  
iOS 5, Xcode 4.2,  
Objective-C 2.0's ARC,  
LLVM, and more!

# The iOS 5 Developer's Cookbook

Core Concepts and Essential Recipes  
for iOS Programmers

**Developer's Library**



---

## **Praise for previous editions of *The iPhone Developer's Cookbook***

“This book would be a bargain at ten times its price! If you are writing iPhone software, it will save you weeks of development time. Erica has included dozens of crisp and clear examples illustrating essential iPhone development techniques and many others that show special effects going way beyond Apple’s official documentation.”

—Tim Burks, iPhone Software Developer, TootSweet Software

“Erica Sadun’s technical expertise lives up to the Addison-Wesley name. *The iPhone Developer's Cookbook* is a comprehensive walkthrough of iPhone development that will help anyone out, from beginners to more experienced developers. Code samples and screenshots help punctuate the numerous tips and tricks in this book.”

—Jacqui Cheng, Associate Editor, *Ars Technica*

“We make our living writing this stuff and yet I am humbled by Erica’s command of her subject matter and the way she presents the material: pleasantly informal, then very appropriately detailed technically. This is a going to be the Petzold book for iPhone developers.”

—Daniel Pasco, Lead Developer and CEO, Black Pixel Luminance

“*The iPhone Developer's Cookbook* should be the first resource for the beginning iPhone programmer, and is the best supplemental material to Apple’s own documentation.”

—Alex C. Schaefer, Lead Programmer, ApolloIM, iPhone Application Development Specialist, MeLLmo, Inc.

“Erica’s book is a truly great resource for Cocoa Touch developers. This book goes far beyond the documentation on Apple’s Web site, and she includes methods that give the developer a deeper understanding of the iPhone OS, by letting them glimpse at what’s going on behind the scenes on this incredible mobile platform.”

—John Zorko, Sr. Software Engineer, Mobile Devices

“I’ve found this book to be an invaluable resource for those times when I need to quickly grasp a new concept and walk away with a working block of code. Erica has an impressive knowledge of the iPhone platform, is a master at describing technical information, and provides a compendium of excellent code examples.”

—John Muchow, 3 Sixty Software, LLC; founder, [iPhoneDeveloperTips.com](http://iPhoneDeveloperTips.com)

“This book is the most complete guide if you want coding for the iPhone, covering from the basics to the newest and coolest technologies. I built several applications in the past, but I still learned a huge amount from this book. It is a must-have for every iPhone developer.”

—Roberto Gamboni, Software Engineer, AT&T Interactive

“It’s rare that developer cookbooks can both provide good recipes and solid discussion of fundamental techniques, but Erica Sadun’s book manages to do both very well.”

—Jeremy McNally, Developer, entp

# The iOS 5 Developer's Cookbook:

Core Concepts and Essential  
Recipes for iOS Programmers

---

Third Edition

Erica Sadun

◆◆ Addison-Wesley

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

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

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

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

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

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

**International Sales**  
**international@pearsoned.com**

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

Visit us on the Web: [informit.com/aw](http://informit.com/aw)

Library of Congress Cataloging-in-Publication Data

Sadun, Erica.

The iOS 5 developer's cookbook : core concepts and essential recipes for iOS programmers / Erica Sadun. — 3rd ed.  
p. cm.

Rev. ed. of: iPhone developer's cookbook. 2009.

ISBN 978-0-321-75426-4 (pbk. : alk. paper)

1. iPhone (Smartphone)—Programming. 2. Computer software—Development. 3. Mobile computing. I. Sadun, Erica. iPhone developer's cookbook. II. Title.

QA76.8.I64S33 2011

004.16'7—dc23

2011036427

Copyright © 2012 Pearson Education, Inc.

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

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

ISBN-13: 978-0-321-75426-4

ISBN-10: 0-321-75426-3

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.

First printing November 2011

**Editor-in-Chief**  
Mark Taub

**Senior Acquisitions Editor**  
Chuck Toporek

**Senior Development Editor**  
Chris Zahn

**Managing Editor**  
Kristy Hart

**Project Editor**  
Anne Goebel

**Copy Editor**  
Bart Reed

**Indexer**  
Erika Millen

**Proofreader**  
Linda Seifert

**Technical Reviewers**  
Jon Bauer  
Joachim Bean  
Tim Burks  
Matt Martel

**Editorial Assistant**  
Olivia Basegio

**Cover Designer**  
Gary Adair

**Composition**  
Nonie Ratcliff



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



# Contents at a Glance

Preface    **xxvii**

- 1**    Introducing the iOS SDK    **1**
- 2**    Objective-C Boot Camp    **51**
- 3**    Building Your First Project    **127**
- 4**    Designing Interfaces    **191**
- 5**    Working with View Controllers    **247**
- 6**    Assembling Views and Animations    **295**
- 7**    Working with Images    **337**
- 8**    Gestures and Touches    **397**
- 9**    Building and Using Controls    **445**
- 10**    Working with Text    **491**
- 11**    Creating and Managing Table Views    **555**
- 12**    A Taste of Core Data    **611**
- 13**    Alerting the User    **633**
- 14**    Device Capabilities    **661**
- 15**    Networking    **695**

# Contents

## **Preface   xxvii**

## **1   Introducing the iOS SDK   1**

iOS Developer Programs	1
Online Developer Program	2
Standard Developer Program	2
Developer Enterprise Program	3
Developer University Program	3
Registering	3
Getting Started	3
Downloading the SDK	4
Development Devices	5
Simulator Limitations	6
Tethering	7
Understanding Model Differences	8
Screen Size	9
Camera	9
Audio	10
Telephony	10
Core Location and Core Motion Differences	10
Vibration Support and Proximity	11
Processor Speeds	11
OpenGL ES	11
Platform Limitations	12
Storage Limits	12
Data Access Limits	13
Memory Limits	13
Interaction Limits	16
Energy Limits	16
Application Limits	17
User Behavior Limits	18
SDK Limitations	18



Using the Provisioning Portal	19
Setting Up Your Team	19
Requesting Certificates	20
Registering Devices	20
Registering Application Identifiers	21
Provisioning	22
Putting Together iPhone Projects	23
The iPhone Application Skeleton	25
main.m	26
Application Delegate	28
View Controller	30
A Note about the Sample Code in This Book	31
iOS Application Components	32
Application Folder Hierarchy	32
The Executable	32
The Info.plist File	33
The Icon and Launch Images	34
Interface Builder Files	37
Files Not Found in the Application Bundle	37
IPA Archives	38
Sandboxes	38
Programming Paradigms	39
Object-Oriented Programming	39
Model-View-Controller	40
Summary	48

## **2 Objective-C Boot Camp 51**

The Objective-C Programming Language	51
Classes and Objects	52
Creating Objects	54
Memory Allocation	54
Releasing Memory	55
Understanding Retain Counts with MRR	56
Methods, Messages, and Selectors	57
Undeclared Methods	57
Pointing to Objects	58
Inheriting Methods	59

Declaring Methods	59
Implementing Methods	60
Class Methods	62
Fast Enumeration	63
Class Hierarchy	63
Logging Information	64
Basic Memory Management	66
Managing Memory with MRR	67
Managing Memory with ARC	70
Properties	71
Encapsulation	71
Dot Notation	71
Properties and Memory Management	72
Declaring Properties	73
Creating Custom Getters and Setters	74
Property Qualifiers	76
Key-Value Coding	78
Key-Value Observing	79
MRR and High Retain Counts	79
Other Ways to Create Objects	80
Deallocating Objects	82
Using Blocks	84
Defining Blocks in Your Code	85
Assigning Block References	85
Blocks and Local Variables	87
Blocks and typedef	87
Blocks and Memory Management with MRR	88
Other Uses for Blocks	88
Getting Up to Speed with ARC	88
Property and Variable Qualifiers	89
Reference Cycles	92
Autorelease Pools	94
Opting into and out of ARC	95
Migrating to ARC	95
Disabling ARC across a Target	96
Disabling ARC on a File-by-File Basis	97

Creating an ARC-less Project from Xcode Templates	97
ARC Rules	98
Using ARC with Core Foundation and Toll Free Bridging	99
Casting between Objective-C and Core Foundation	99
Choosing a Bridging Approach	101
Runtime Workarounds	102
Tips and Tricks for Working with ARC	103
Crafting Singletons	103
Categories (Extending Classes)	104
Protocols	106
Defining a Protocol	106
Incorporating a Protocol	107
Adding Callbacks	107
Declaring Optional Callbacks	107
Implementing Optional Callbacks	108
Conforming to a Protocol	108
Foundation Classes	109
Strings	110
Numbers and Dates	115
Collections	117
One More Thing: Message Forwarding	123
Implementing Message Forwarding	123
House Cleaning	125
Super-easy Forwarding	126
Summary	126
<b>3 Building Your First Project</b>	<b>127</b>
Creating New Projects	127
Building Hello World the Template Way	129
Create a New Project	129
Introducing the Xcode Workspace	132
Review the Project	137
Open the iPhone Storyboard	138
Edit the View	140
Run Your Application	141

Using the Simulator	142
Simulator: Behind the Scenes	144
Sharing Simulator Applications	146
The Minimalist Hello World	146
Browsing the SDK APIs	149
Converting Interface Builder Files to Their Objective-C Equivalents	151
Using the Debugger	153
Set a Breakpoint	153
Open the Debugger	154
Inspect the Label	155
Set Another Breakpoint	156
Backtraces	157
Console	158
Add Simple Debug Tracing	158
Memory Management	158
Recipe: Using Instruments to Detect Leaks	159
Recipe: Using Instruments to Monitor Cached Object Allocations	162
Simulating Low-Memory Conditions	163
Analyzing Your Code	165
From Xcode to Device: The Organizer Interface	165
Devices	165
Summary	167
Provisioning Profiles	168
Device Logs	168
Applications	169
Console	169
Screenshots	170
Building for the iOS Device	170
Using a Development Provision	170
Enable a Device	171
Inspect Your Application Identifier	172
Set Your Device and Code Signing Identity	172
Set Your Base and Deployment SDK Targets	173
Compile and Run the Hello World Application	174
Signing Compiled Applications	175

Detecting Simulator Builds with Compile-Time Checks	175
Performing Runtime Compatibility Checks	175
Pragma Marks	177
Collapsing Methods	178
Preparing for Distribution	178
Locating and Cleaning Builds	178
Using Schemes and Actions	179
Adding Build Configurations	181
About Ad Hoc Distribution	182
Building Ad-Hoc Packages	183
Over-the-Air Ad Hoc Distribution	184
Building a Manifest	184
Submitting to the App Store	186
Summary	188

## **4 Designing Interfaces 191**

UIView and UIWindow	191
Views That Display Data	192
Views for Making Choices	193
Controls	193
Tables and Pickers	195
Bars	195
Progress and Activity	196
View Controllers	196
UIViewController	197
UINavigationController	197
UITabBarController	198
Split View Controllers	198
Page View Controller	199
Popover Controllers	199
Table Controllers	199
Address Book Controllers	200
Image Picker	200
Mail Composition	200
Document Interaction Controller	200
GameKit Peer Picker	201
Media Player Controllers	201

View Design Geometry	201
Status Bar	202
Navigation Bars, Toolbars, and Tab Bars	203
Keyboards and Pickers	205
Text Fields	207
The UIScreen Class	207
Building Interfaces	207
Walkthrough: Building Storyboard Interfaces	208
Create a New Project	208
Add More View Controllers	208
Organize Your Views	209
Update Classes	210
Name Your Scenes	211
Edit View Attributes	211
Add Navigation Buttons	211
Add Another Navigation Controller	213
Name the Controllers	213
Tint the Navigation Bars	214
Add a Button	214
Change the Entry Point	215
Add Dismiss Code	215
Run the App	216
Popover Walkthrough	216
Add a Navigation Controller	216
Change the View Controller Class	217
Customize the Popover View	217
Make the Connections	218
Edit the Code	218
Walkthrough: Building an iOS-based Temperature Converter with IB	220
Create a New Project	220
Add Media	221
Interface Builder	221
Add Labels and Views	222
Enable Reorientation	223
Test the Interface	223
Add Outlets and an Action	223

Add the Conversion Method	225
Update the Keyboard Type	225
Connecting the iPad Interface	226
Walkthrough: Building a Converter Interface by Hand	227
Putting the Project Together	230
Walkthrough: Creating, Loading, and Using Hybrid Interfaces	230
Create a New XIB Interface File	231
Add a View and Populate It	231
Tag Your Views	231
Edit the Code	232
Designing for Rotation	233
Enabling Reorientation	233
Autosizing	235
Autosizing Example	237
Evaluating the Autosize Option	238
Moving Views	239
Recipe: Moving Views by Mimicking Templates	240
One More Thing: A Few Great Interface Builder Tips	243
Summary	245

## **5 Working with View Controllers 247**

Developing with Navigation Controllers and Split Views	247
Using Navigation Controllers and Stacks	249
Pushing and Popping View Controllers	249
The Navigation Item Class	250
Modal Presentation	251
Recipe: Building a Simple Two-Item Menu	252
Recipe: Adding a Segmented Control	253
Recipe: Navigating Between View Controllers	255
Recipe: Presenting a Custom Modal Information View	258
Recipe: Page View Controllers	262
Book Properties	262
Wrapping the Implementation	263
Exploring the Recipe	264

Recipe: Scrubbing Pages in a Page View Controller	269
Recipe: Tab Bars	271
Recipe: Remembering Tab State	275
Recipe: Building Split View Controllers	278
Recipe: Creating Universal Split View/Navigation Apps	282
Recipe: Custom Containers and Segues	284
Transitioning Between View Controllers	290
One More Thing: Interface Builder and Tab Bar Controllers	291
Summary	292

## **6 Assembling Views and Animations 295**

View Hierarchies	295
Recipe: Recovering a View Hierarchy Tree	297
Recipe: Querying Subviews	298
Managing Subviews	300
Adding Subviews	300
Reordering and Removing Subviews	300
View Callbacks	301
Recipe: Tagging and Retrieving Views	301
Using Tags to Find Views	302
Recipe: Naming Views	303
Associated Objects	304
Using a Name Dictionary	305
View Geometry	308
Frames	309
Transforms	310
Coordinate Systems	310
Recipe: Working with View Frames	311
Adjusting Sizes	312
CGRects and Centers	313
Other Utility Methods	314
Recipe: Randomly Moving a Bounded View	318
Recipe: Transforming Views	319
Display and Interaction Traits	320



UIView Animations	321
Building UIView Animation Transactions	322
Building Animations with Blocks	323
Conditional Animation	324
Recipe: Fading a View In and Out	324
Recipe: Swapping Views	326
Recipe: Flipping Views	327
Recipe: Using Core Animation Transitions	328
Recipe: Bouncing Views as They Appear	329
Recipe: Image View Animations	331
One More Thing: Adding Reflections to Views	332
Summary	335

## **7 Working with Images 337**

Finding and Loading Images	337
Reading Image Data	339
Recipe: Accessing Photos from the iOS Photo Album	342
Working with the Image Picker	342
Recovering Image Edit Information	344
Recipe: Retrieving Images from Asset URLs	347
Recipe: Snapping Photos and Writing Them to the Photo Album	349
Choosing Between Cameras	351
Saving Pictures to the Documents Folder	353
Recipe: E-mailing Pictures	354
Creating Message Contents	354
Presenting the Composition Controller	356
Automating Camera Shots	358
Using a Custom Camera Overlay	358
Recipe: Accessing the AVFoundation Camera	359
Requiring Cameras	360
Querying and Retrieving Cameras	360
Establishing a Camera Session	361
Switching Cameras	363
Camera Previews	364

Laying Out a Camera Preview	364
EXIF	365
Image Geometry	365
Building Camera Helper	367
Recipe: Adding a Core Image Filter	368
Recipe: Core Image Face Detection	370
Extracting Faces	376
Recipe: Working with Bitmap Representations	377
Drawing into a Bitmap Context	378
Applying Image Processing	380
Image Processing Realities	382
Recipe: Sampling a Live Feed	384
Converting to HSB	386
Recipe: Building Thumbnails from Images	387
Taking View-based Screenshots	390
Drawing into PDF Files	390
Creating New Images from Scratch	391
Recipe: Displaying Images in a Scrollable View	392
Creating a Multi-Image Paged Scroll	395
Summary	396

## **8 Gestures and Touches 397**

Touches	397
Phases	398
Touches and Responder Methods	399
Touching Views	399
Multitouch	400
Gesture Recognizers	400
Recipe: Adding a Simple Direct Manipulation Interface	401
Recipe: Adding Pan Gesture Recognizers	402
Recipe: Using Multiple Gesture Recognizers at Once	404
Resolving Gesture Conflicts	407
Recipe: Constraining Movement	408
Recipe: Testing Touches	409
Recipe: Testing Against a Bitmap	411

Recipe: Adding Persistence to Direct Manipulation  
Interfaces 413

Storing State 413

Recovering State 415

Recipe: Persistence Through Archiving 416

Recipe: Adding Undo Support 418

Creating an Undo Manager 418

Child-View Undo Support 418

Working with Navigation Bars 419

Registering Undos 420

Adding Shake-Controlled Undo Support 422

Add an Action Name for Undo and Redo  
(Optional) 422

Provide Shake-To-Edit Support 423

Force First Responder 423

Recipe: Drawing Touches Onscreen 424

Recipe: Smoothing Drawings 426

Recipe: Detecting Circles 429

Creating a Custom Gesture Recognizer 433

Recipe: Using Multitouch 435

Retaining Touch Paths 438

One More Thing: Dragging from a Scroll View 440

Summary 443

## **9 Building and Using Controls 445**

The UIControl Class 445

Kinds of Controls 445

Control Events 446

Buttons 448

Adding Buttons in Interface Builder 449

Art 450

Connecting Buttons to Actions 451

Buttons That Are Not Buttons 452

Building Custom Buttons in Xcode 453

Multiline Button Text 455

Adding Animated Elements to Buttons 456

Recipe: Animating Button Responses 456

Recipe: Adding a Slider With a Custom Thumb	458
Customizing UISlider	459
Adding Efficiency	460
Appearance Proxies	460
Recipe: Creating a Twice-Tappable Segmented Control	465
Recipe: Subclassing UIControl	467
Creating UIControls	468
Tracking Touches	468
Dispatching Events	468
Working with Switches and Steppers	471
Recipe: Building a Star Slider	472
Recipe: Building a Touch Wheel	476
Adding a Page Indicator Control	478
Recipe: Creating a Customizable Paged Scroller	481
Building a Toolbar	486
Building Toolbars in Code	487
iOS 5 Toolbar Tips	489
Summary	489

## **10 Working with Text 491**

Recipe: Dismissing a UITextField Keyboard	491
Text Trait Properties	492
Other Text Field Properties	493
Recipe: Adjusting Views Around Keyboards	495
Recipe: Dismissing Text Views with Custom Accessory Views	498
Recipe: Resizing Views with Hardware Keyboards	500
Recipe: Creating a Custom Input View	503
Recipe: Making Text-Input-Aware Views	508
Recipe: Adding Custom Input Views to Non-Text Views	511
Adding Input Clicks	511
Recipe: Building a Better Text Editor	513
Recipe: Text Entry Filtering	516
Recipe: Detecting Text Patterns	518
Rolling Your Own Expressions	518

Enumerating Regular Expressions	519
Data Detectors	520
Adding Built-in Type Detectors	520
Recipe: Detecting Misspelling in a UITextView	522
Searching for Text Strings	523
Recipe: Dumping Fonts	524
Recipe: Adding Custom Fonts to Your App	525
Recipe: Basic Core Text and Attributed Strings	526
Using Pseudo-HTML to Create Attributed Text	532
Recipe: Splitting Core Text into Pages	536
Recipe: Drawing Core Text into PDF	537
Recipe: Drawing into Nonrectangular Paths	539
Recipe: Drawing Text onto Paths	542
Drawing Text onto Bezier Paths	543
Drawing Proportionately	544
Drawing the Glyph	545
One More Thing: Big Phone Text	551
Summary	554

## **11 Creating and Managing Table Views 555**

Introducing UITableView and UITableViewController	555
Creating the Table	556
Recipe: Implementing a Basic Table	558
Populating a Table	558
Data Source Methods	559
Reusing Cells	560
Responding to User Touches	560
Selection Color	561
Changing a Table's Background Color	561
Cell Types	562
Recipe: Building Custom Cells in Interface Builder	563
Adding in Custom Selection Traits	565
Alternating Cell Colors	565
Removing Selection Highlights from Cells	566
Creating Grouped Tables	567
Recipe: Remembering Control State for Custom Cells	567

Visualizing Cell Reuse	570
Creating Checked Table Cells	571
Working with Disclosure Accessories	572
Recipe: Table Edits	574
Displaying Remove Controls	575
Dismissing Remove Controls	575
Handling Delete Requests	576
Supporting Undo	576
Swiping Cells	576
Adding Cells	576
Reordering Cells	579
Sorting Tables Algorithmically	580
Recipe: Working with Sections	581
Building Sections	582
Counting Sections and Rows	583
Returning Cells	583
Creating Header Titles	584
Creating a Section Index	584
Delegation with Sections	585
Recipe: Searching Through a Table	586
Creating a Search Display Controller	586
Building the Searchable Data Source Methods	587
Delegate Methods	589
Using a Search-Aware Index	589
Customizing Headers and Footers	591
Recipe: Adding “Pull-to-Refresh” to Your Table	592
Coding a Custom Group Table	595
Creating Grouped Preferences Tables	595
Recipe: Building a Multiwheel Table	597
Creating the UIPickerView	598
Recipe: Using a View-based Picker	601
Recipe: Using the UIDatePicker	603
Creating the Date Picker	603
One More Thing: Formatting Dates	606
Summary	608

**12 A Taste of Core Data 611**

- Introducing Core Data 611
  - Creating and Editing Model Files 612
  - Generating Class Files 614
  - Creating a Core Data Context 615
  - Adding Objects 616
  - Querying the Database 618
  - Detecting Changes 619
  - Removing Objects 619
- Recipe: Using Core Data for a Table Data Source 620
- Recipe: Search Tables and Core Data 623
- Recipe: Integrating Core Data Table Views with Live Data Edits 625
- Recipe: Implementing Undo/Redo Support with Core Data 628
- Summary 632

**13 Alerting the User 633**

- Talking Directly to Your User Through Alerts 633
  - Building Simple Alerts 633
  - Alert Delegates 634
  - Displaying the Alert 636
  - Kinds of Alerts 636
- “Please Wait”: Showing Progress to Your User 637
  - Using UIActivityIndicatorView 638
  - Using UIProgressView 639
- Recipe: No-Button Alerts 639
  - Building a Floating Progress Monitor 642
- Recipe: Creating Modal Alerts with Run Loops 642
- Recipe: Using Variadic Arguments with Alert Views 645
- Presenting Simple Menus 646
  - Scrolling Menus 648
  - Displaying Text in Action Sheets 648
- Recipe: Building Custom Overlays 649
  - Tappable Overlays 650
- Recipe: Basic Popovers 650
- Recipe: Local Notifications 652

Alert Indicators	654
Badging Applications	654
Recipe: Simple Audio Alerts	654
System Sounds	655
Vibration	656
Alerts	656
Delays	656
One More Thing: Showing the Volume Alert	658
Summary	659

## **14 Device Capabilities 661**

Accessing Basic Device Information	661
Adding Device Capability Restrictions	662
Recipe: Recovering Additional Device Information	664
Monitoring the iPhone Battery State	666
Enabling and Disabling the Proximity Sensor	667
Recipe: Using Acceleration to Locate “Up”	668
Retrieving the Current Accelerometer Angle Synchronously	670
Calculate a Relative Angle	671
Working with Basic Orientation	671
Recipe: Using Acceleration to Move Onscreen Objects	672
Adding a Little Sparkle	675
Recipe: Core Motion Basics	676
Testing for Sensors	677
Handler Blocks	677
Recipe: Retrieving and Using Device Attitude	680
Detecting Shakes Using Motion Events	681
Recipe: Detecting Shakes via the Accelerometer	683
Recipe: Using External Screens	686
Detecting Screens	687
Retrieving Screen Resolutions	687
Setting Up Video Out	688
Adding a Display Link	688
Overscanning Compensation	688
VIDEOkit	688
One More Thing: Checking for Available Disk Space	692
Summary	693



**15 Networking 695**

Checking Your Network Status 695

Recipe: Extending the `UIDevice` Class for Reachability 697

Scanning for Connectivity Changes 700

Recovering IP and Host Information 702

Using Queues for Blocking Checks 705

Checking Site Availability 707

Synchronous Downloads 709

Asynchronous Downloads in Theory 713

Recipe: Asynchronous Downloads 715

Handling Authentication Challenges 721

Storing Credentials 722

Recipe: Storing and Retrieving Keychain Credentials 725

Recipe: Uploading Data 728

`NSOperationQueue` 728

Twitter 732

Recipe: Converting XML into Trees 733

Trees 733

Building a Parse Tree 734

Using the Tree Results 736

Recipe: Building a Simple Web-based Server 738

One More Thing: Using JSON Serialization 742

Summary 742

**Index 745**

# Acknowledgments

This book would not exist without the efforts of Chuck Toporek (my editor and whip-cracker), 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, Anne Goebel, Bart Reed, Linda Seifert, Erika Millen, Nonie Ratcliff, and Gary Adair. Thanks also to the crew at Safari for getting my book up in Rough Cuts and for quickly fixing things when technical glitches occurred.

Thanks go as well to Neil Salkind, my agent of many years, to the tech reviewers (Jon Bauer, Joachim Bean, Tim Burks, and Matt Martel) 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 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 Watnough, Youssef Francis, Bryan Henry, William DeMuro, Jeremy Sinclair, Arshad Tayyeb, Daniel Peebles, ChronicProductions, Greg Hartstein, Emanuele Vulcano, Sean Heber, Josh Bleacher Snyder, Eric Chamberlain, Steven Troughton-Smith, Dustin Howett, Dick Applebaum, Kevin Ballard, Hamish Allan, Kevin McAllister, Jay Abbott, Tim Grant Davies, Chris Greening, Landon Fuller, Wil Macaulay, Stefan Hafeneger, Scott Yelich, chrallelinder, John Varghese, Andrea Fanfani, J. Roman, jtbandes, Artissimo, Aaron Alexander, Christopher Campbell Jensen, rincewind42, 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 Demarest, Yanik Magnan, ashikase, Shane Zatezalo, Tito Ciuro, Jonah Williams of Carbon Five, Joshua Weinberg, biappi, Eric Mock, Jay Spencer, and everyone at the iPhone developer channels at [irc.saurik.com](http://irc.saurik.com) and [irc.freenode.net](http://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 last few months by testing applications, offering suggestions, and just being awesome people. I try to remind myself on a daily basis how lucky I am that these kids are part of my life.

## About the Author

**Erica Sadun** is the bestselling author, coauthor, and contributor to several dozen books on programming, digital video and photography, and web design, including the widely popular *The iPhone Developer's Cookbook: Building Applications with the iPhone 3.0 SDK, Second Edition*. She currently blogs at TUAUW.com, and has blogged in the past at O'Reilly's Mac DevCenter, Lifehacker, and Ars Technica. In addition to being the author of dozens of iOS-native applications, Erica holds a Ph.D. in Computer Science from Georgia Tech's Graphics, Visualization and Usability Center. A geek, a programmer, and an author, she's never met a gadget she didn't love. When not writing, she and her geek husband parent three geeks-in-training, who regard their parents with restrained bemusement, when they're not busy rewiring the house or plotting global dominance.

# Preface

This is the iOS Cookbook you've been waiting for!

Last year, when iOS 4 debuted, my editor and I had a hard decision to make: Publish the book on iOS 4 and don't include Xcode 4 material, or hold off until Apple released Xcode 4. We chose to hold off for Xcode 4, feeling that many people would expect to see it covered in the book. What we couldn't anticipate, however, is that Apple's NDA would last until Spring 2011, and we knew iOS 5 was right around the corner.

Stuck between a rock and an iOS release, we decided to update the book to iOS 4.3 and to release that as an ebook-only version (that is, we aren't planning to print that edition—ever). The reason for doing an electronic-only edition on iOS 4.3 was so developers who wanted that info could still have access to it. Once that update was finished and iOS 5 was introduced at WWDC, I quickly turned my attention to updating—and expanding—the cookbook for iOS 5. This is the version you're currently reading. Finally!

This edition, *The iOS 5 Developer's Cookbook*, carries through with the promise of the subtitle: *Core Concepts and Essential Recipes for iOS Programmers*. That means this book covers what you need to know to get started. For someone who's just starting out as an iOS developer, this is the ideal book because it covers the tools (Xcode and Interface Builder), the language (Objective-C), and the basic elements common to pretty much every iOS app out there (table views, custom controls, split views, and the like).

But we're not stopping there. Mid-October 2011 is our cutoff date for getting the book to production this year. While the book is in production, I'll continue writing and adding more advanced material to *The iOS 5 Developer's Cookbook*, along with a bunch of new chapters that won't make it to print.

Our plan is to combine all this material to create *The iOS 5 Developer's Cookbook: Expanded Electronic Edition*, which will release in electronic-only form (namely, ePub for iBooks, Kindle, and PDF for desktops). It will hit the virtual electronic shelf at the same time this printed book hits the stands. The Expanded Electronic Edition will include the equivalent of what would amount to several hundred pages of printed material. You can see our reason for not wanting to print all that. There *is* an electronic version of the very book you hold in your hands, but if you want access to the entire *The iOS 5 Developer's Cookbook: Expanded Electronic Edition*, you will need to purchase that edition separately.

As in the past, sample code can be found at github. The repository for this cookbook is located at <https://github.com/erica/iOS-5-Cookbook>, all of it written after WWDC 2011 and during the time when Apple was routing iOS 5 betas to developers.

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

—Erica Sadun, November 2011

## 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 of those iOS devices to test out your application, preferably a 3GS or later, a third-gen iPod touch or later, or any iPad. The following list covers the basics of what you need to begin:

- **Apple's iOS SDK**—The latest version of the iOS SDK can be downloaded from Apple's iOS Dev Center ([developer.apple.com/ios](http://developer.apple.com/ios)). If you plan to sell apps through the App Store, you will need to become a paid iOS developer, which costs \$99/year for individuals and \$299/year for enterprise (that is, corporate) developers. Registered developers receive certificates that allow them to “sign” and download their applications to their iPhone/iPod touch for testing and debugging.

### University Student Program

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

- **An Intel-based Mac running Mac OS X Snow Leopard (v 10.6) or Lion (v 10.7)**—You need plenty of disk space for development, and your Mac should have at least 1GB RAM, preferably 2GB or 4GB to help speed up compile time.
- **An iOS device**—Although the iOS SDK and Xcode include a simulator for you to test your applications in, you really do need to have an iPhone, iPad, and/or iPod touch if you're going to develop for the platform. You can use the USB cable to 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 you can test on the same platforms your target audience will use.
- **At least one available USB 2.0 port**—This enables you to tether a development iPhone or iPod touch to your computer for file transfer and testing.
- **An Internet connection**—This connection enables you to test your programs with a live Wi-Fi connection as well as with an EDGE or 3G service.
- **Familiarity with Objective-C**—To program for the iPhone, you need to know Objective-C 2.0. The language is based on ANSI C with object-oriented extensions, which means you also need to know a bit of C too. If you have programmed with Java or C++ and are familiar with C, making the move to Objective-C is pretty easy. Chapter 2, “Objective-C Boot Camp,” helps you get up to speed.

## Your Roadmap to Mac/iOS Development

As mentioned earlier, one book can't be everything to everyone. And try as I might, if we were to pack everything you'd 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 us 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. To help you along the way, my editor Chuck Toporek and I put together the flowchart shown in Figure P-1 to point you at some books of interest.

Once you know C, you've got a few options for learning how to program with Objective-C. For a quick-and-dirty overview of Objective-C, you can turn to Chapter 2 of this book and read the "Objective-C Boot Camp." However, if you want a more in-depth view of the language, you can either read Apple's own documentation or pick up one of these books on Objective-C:

- *Objective-C Programming: The Big Nerd Ranch Guide*, by Aaron Hillegass (Big Nerd Ranch, 2012).
- *Learning Objective-C: A Hands-on Guide to Objective-C for Mac and iOS Developers*, by Robert Clair (Addison-Wesley, 2011).
- *Programming in Objective-C 2.0, Fourth Edition*, by Stephen Kochan (Addison-Wesley, 2012).

With the language behind you, next up is tackling Cocoa and the developer tools, otherwise known as Xcode. For that, you have a few different options. Again, you can refer to Apple's own documentation on Cocoa and Xcode,<sup>1</sup> or if you prefer books, you can learn from the best. Aaron Hillegass, founder of the Big Nerd Ranch in Atlanta,<sup>2</sup> is the coauthor of *iOS Programming: The Big Nerd Ranch Guide, Second Edition* and author of *Cocoa Programming for Mac OS X*, soon to be in its fourth edition. Aaron's book is highly regarded in Mac developer circles and is the most-recommended book you'll see on the *cocoa-dev* mailing list. To learn more about Xcode, look no further than Fritz Anderson's *Xcode 4 Unleashed* from Sams Publishing.

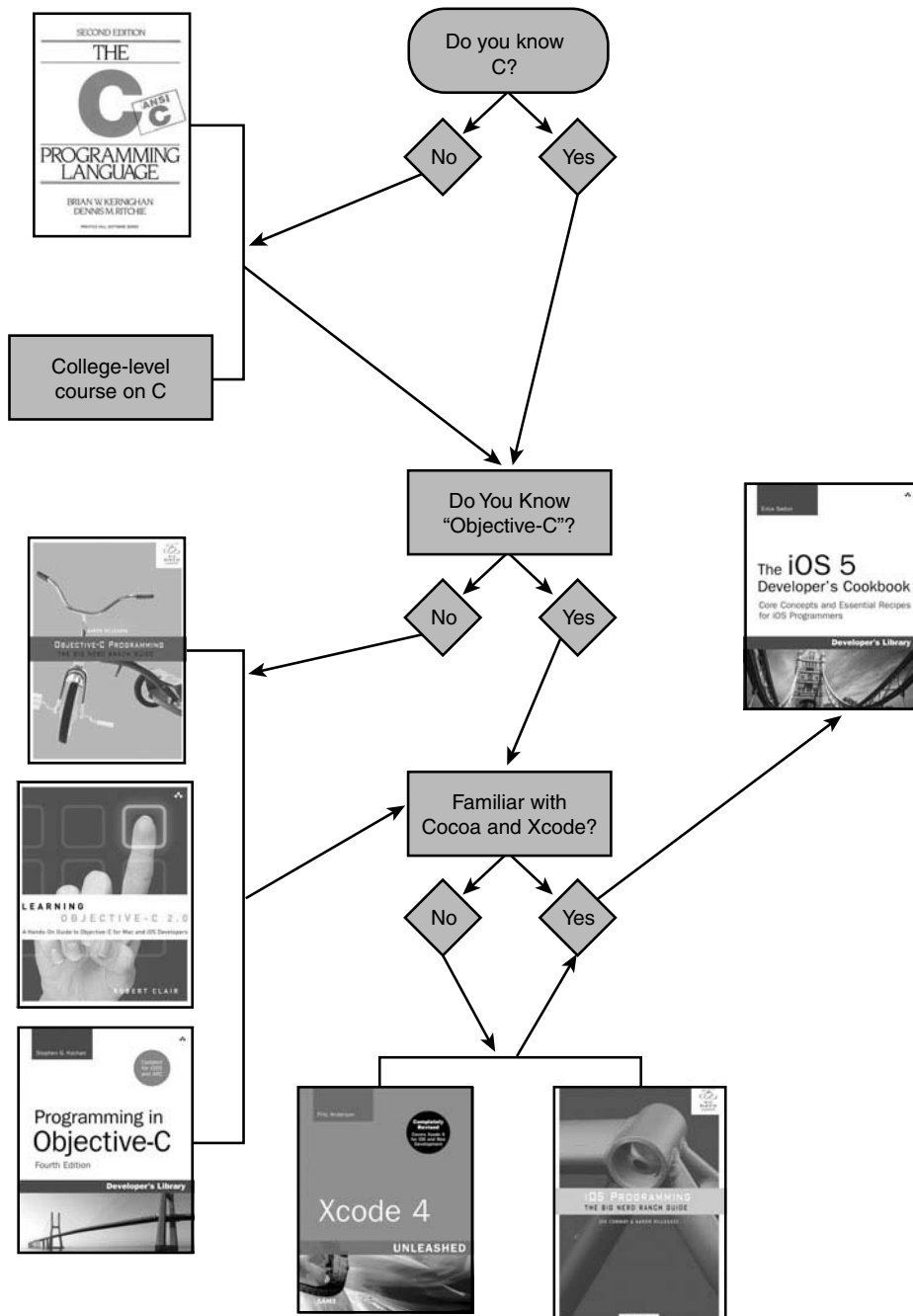


Figure P-1 What it takes to be an iOS programmer.

## Note

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

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

## How This Book Is Organized

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

*The iOS 5 Developer's Cookbook* offers you “cut-and-paste convenience,” which means you can freely reuse the source code from recipes in this book for your own applications and then tweak the code to suit your app's needs.

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

- **Chapter 1, “Introducing the iOS SDK”**—Chapter 1 introduces the iOS SDK and explores iOS as a delivery platform, limitations and all. It explains the breakdown of the standard iOS application and helps you get started with the iOS Developer Portal.
- **Chapter 2, “Objective-C Boot Camp”**—If you're new to Objective-C as well as to iOS, you'll appreciate this basic skills chapter. Objective-C is the standard programming language for both iOS and for Mac OS X. It offers a powerful object-oriented language that lets you build applications that leverage Apple's Cocoa and Cocoa Touch frameworks. Chapter 2 introduces the language, provides an overview of its object-oriented features, discusses memory management skills, and adds a common class overview to get you started with Objective-C programming.
- **Chapter 3, “Building Your First Project”**—Chapter 3 covers the basics for building your first Hello World-style applications. It introduces Xcode and Interface Builder, showing how you can use these tools in your projects. You read about basic debugging tools, walk through using them, and pick up some tips about handy compiler directives. You'll also discover how to create provisioning



profiles and use them to deploy your application to your device, to beta testers, and to the App Store.

- **Chapter 4, “Designing Interfaces”**—Chapter 4 introduces iOS’s library of visual classes. It surveys these classes and their geometry. In this chapter, you learn how to work with these visual classes and discover how to handle tasks such as device reorientation. You’ll read about solutions for laying out and customizing interfaces and learn about hybrid solutions that rely both on Interface Builder–created interfaces and Objective-C–centered ones.
- **Chapter 5, “Working with View Controllers”**—The iOS paradigm in a nutshell is this: small screen, big virtual worlds. In Chapter 5, you discover the various view controller classes that enable you to enlarge and order the virtual spaces your users interact with. You learn how to let these powerful objects perform all the heavy lifting when navigating between iOS application screens or breaking down iPad applications into master-detail views.
- **Chapter 6, “Assembling Views and Animations”**—Chapter 6 introduces iOS views, objects that live on your screen. You see how to lay out, create, and order your views to create backbones for your applications. You read about view hierarchies, geometries, and animations, features that bring your iOS applications to life.
- **Chapter 7, “Working with Images”**—Chapter 7 introduces images, specifically the UIImage class, and teaches you all the basic know-how you need for working with iOS images. You learn how to load, store, and modify image data in your applications. You see how to add images to views and how to convert views into images. And you discover how to process image data to create special effects, how to access images on a byte-by-byte basis, and how to take photos with your device’s built-in camera.
- **Chapter 8, “Gestures and Touches”**—On iOS, the touch provides the most important way that users communicate their intent to an application. Touches are not limited to button presses and keyboard interaction. Chapter 8 introduces direct manipulation interfaces, multitouch, and more. You see how to create views that users can drag around the screen and read about distinguishing and interpreting gestures, as well as how to create custom gesture recognizers.
- **Chapter 9, “Building and Using Controls”**—Control classes provide the basis for many of iOS’s interactive elements, including buttons, sliders, and switches. This chapter introduces controls and their use. You read about standard control interactions and how to customize these objects for your application’s specific needs. You even learn how to build your own controls from the ground up, as Chapter 9 creates custom switches, star ratings controls, and a virtual touch wheel.
- **Chapter 10, “Working with Text”**—From text fields and text views to iOS’s new and powerful Core Text abilities and inline spelling checkers, Chapter 10 introduces everything you need to know to work with iOS text in your apps.

- **Chapter 11, “Creating and Managing Table Views”**—Tables provide a scrolling interaction class that works particularly well on a small, cramped device. Many, if not most, apps that ship with the iPhone and iPod touch center on tables, including Settings, YouTube, Stocks, and Weather. Chapter 11 shows how iPhone tables work, what kinds of tables are available to you as a developer, and how you can use table features in your own programs.
- **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 iPhone development. Chapter 12 introduces Core Data. It provides just enough recipes to give you a taste of the technology, offering a jumping-off point for further Core Data learning. You learn how to design managed database stores, add and delete data, and query that data from your code and integrate it into your UIKit table views.
- **Chapter 13, “Alerting the User”**—iOS offers many ways to provide users with a heads-up, from pop-up dialogs and progress bars to local notifications, popovers, and audio pings. Chapter 13 shows how to build these indications into your applications and expand your user-alert vocabulary. It introduces standard ways of working with these classes and offers solutions that allow you to craft linear programs without explicit callbacks.
- **Chapter 14, “Device Capabilities”**—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, battery state, and access to onboard hardware. Chapter 14 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. You read about testing for hardware prerequisites at runtime and specifying those prerequisites in the application’s Info.plist file. You discover how to solicit sensor feedback (including using Core Motion) and subscribe to notifications to create callbacks when those sensor states change. This chapter covers the hardware, file system, and sensors available on the iPhone device and helps you programmatically take advantage of those features.
- **Chapter 15, “Networking”**—As an Internet-connected device, iOS is particularly suited to subscribing to web-based services. Apple has lavished the platform with a solid grounding in all kinds of network computing services and their supporting technologies. Chapter 15 surveys common techniques for network computing and offers recipes that simplify day-to-day tasks. You read about network reachability, synchronous and asynchronous downloads, using operation queues, working with the iPhone’s secure keychain to meet authentication challenges, XML parsing, JSON serialization, the new Twitter APIs, and more.

## About the Sample Code

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

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

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

- First, application-creation walkthroughs use the full file structure created by Xcode to mirror the reality of what you'd expect to build on your own. The walkthrough folders may therefore contain a dozen or more files at once.
- Second, standard class and header files are provided when the class itself is the recipe or provides a precooked utility class. Instead of highlighting a technique, some recipes offer these precooked class implementations and categories (that is, extensions to a preexisting class rather than a new class). For those recipes, look for separate `.m` and `.h` files in addition to the skeletal `main.m` that encapsulates the rest of the story.

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

## Getting the Sample Code

The source code for this book can be found at the open-source GitHub hosting site at <https://github.com/erica/iOS-5-Cookbook>. There, you find a chapter-by-chapter collection of source code that provides working examples of the material covered in this book.

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

## Getting Git

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

## Getting GitHub

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

## Contacting the Author

If you have any comments or questions about this book, please drop me an e-mail message at [erica@ericasadun.com](mailto:erica@ericasadun.com), or stop by [www.ericasadun.com](http://www.ericasadun.com) for updates about the book and news for iOS developers. Please feel free to visit, download software, read documentation, and leave your comments.

## Endnotes

- <sup>1</sup> See the *Cocoa Fundamentals Guide* (<http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/CocoaFundamentals/CocoaFundamentals.pdf>) for a head start on Cocoa, and for Xcode, see *A Tour of Xcode* ([http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/A\\_Tour\\_of\\_Xcode/A\\_Tour\\_of\\_Xcode.pdf](http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/A_Tour_of_Xcode/A_Tour_of_Xcode.pdf)).
- <sup>2</sup> Big Nerd Ranch: <http://www.bignerdranch.com>.

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

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

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

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

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

Email: chuck.toporek@pearson.com  
Mail: Chuck Toporek  
Senior Acquisitions Editor  
Addison-Wesley/Pearson Education, Inc.  
75 Arlington St., Ste. 300  
Boston, MA 02116

# Working with View Controllers

**V**iew controllers simplify view management for many iOS applications. They allow you to build applications that centralize many tasks, including view management, orientation changes, and view unloading during low-memory conditions. Each view controller owns a hierarchy of views, which presents a complete element of a unified interface.

In the previous chapter, you built view-controller-based applications using Xcode and Interface Builder. Now it's time to take a deeper look at using view-controller-based classes and how to apply them to real-world situations for both iPhone/iPod and iPad design scenarios. In this chapter you discover how to build simple menus, create view navigation trees, design tab-bar-based and page-view-based applications, and more. This chapter offers hands-on recipes for working with a variety of controller classes.

## Developing with Navigation Controllers and Split Views

The `UINavigationController` class offers one of the most important ways of managing interfaces on a device with limited screen space such as the iPhone and iPod touch. It creates a way for users to drill up and down a hierarchy of interface presentations to create a virtual GUI that's far larger than the device. Navigation controllers fold their GUIs into a neat tree-based scheme. Users travel through that scheme using buttons and choices that transport them around the tree. You see navigation controllers in the Contacts application and in Settings, where selections lead to new screens and “back” buttons move to previous ones.

Several standard GUI elements identify the use of navigation controllers in applications, as seen in Figure 5-1 (left). These include their large navigation bars that appear at the top of each screen, the backward-pointing button at the top-left that appears when the user drills into hierarchies, and option buttons at the top-right that offer other application functionality such as editing. Many navigation controller applications are built around scrolling lists, where elements in that list lead to new screens, indicated by grey and blue chevrons found on the right side of each table cell.



Figure 5-1 The iPhone's navigation controller uses chevrons to indicate that detail views will be pushed onscreen when their parents are selected. On the iPad, split view controllers use the entire screen, separating navigation elements from detail presentations.

The iPad, with its large screen size, doesn't require the kind of space-saving shortcuts that navigation controllers leverage on the iPhone and iPod touch, along with their cousins the tab view controller and modal view controller. iPad applications can use navigation controllers directly, but the `UISplitViewController` shown in Figure 5-1 (right) offers a presentation that's far better suited for the more expansive device.

Notice the differences between the iPhone implementation on the left and the iPad implementation on the right of Figure 5-1. The iPad's split view controller contains no chevrons. When items are tapped, their data appears on the same screen using the large right-hand detail area. The iPhone, lacking this space, presents chevrons that indicate new views will be pushed onscreen. Each approach takes device-specific design into account in its presentation.

Both the iPhone and iPad Inbox views use similar navigation controller elements, including the back button (iPad Book/Gmail for Book), an options button (Edit), and a status in the title bar (with its one unread message). Each of these elements is created using navigation controller API calls working with a hierarchy of e-mail accounts and mailboxes. The difference lies at the bottom of the navigation tree, at the level of individual messages that form the leaves of the data structure. On the iPhone, leaves are indicated by chevrons and, when viewed, are pushed onto the navigation stack, which accumulates the trace of a user's progress through the interface. On the iPad, leaves are presented in a separate view without those chevrons that otherwise indicate that users have reached the extent of the hierarchy traversal.

iPhone-style navigation controllers play roles as well on the iPad. When iPad applications use standard (iPhone-style) navigation controllers, they usually do so in narrow contexts such as transient popover presentations, where the controller is presented onscreen

in a small view with a limited lifetime. Otherwise, iPad applications are encouraged to use the split view approach that occupies the entire screen.

## Using Navigation Controllers and Stacks

Every navigation controller owns a root view controller. This controller forms the base of its stack. You can programmatically push other controllers onto the stack as the user makes choices while navigating through the model's tree. Although the tree itself may be multi-dimensional, the user's path (essentially his history) is always a straight line representing the choices already made to date. Moving to a new choice extends the navigation breadcrumb trail and automatically builds a back button each time a new view controller gets pushed onto the stack.

Users can tap a back button to pop controllers off the stack. The name of each button represents the title of the most recent view controller. As you return through the stack of previous view controllers, each back button previews the view controller that can be returned to. Users can pop back until reaching the root. Then they can go no further. The root is the root, and you cannot pop beyond that root.

This stack-based design lingers even when you plan to use just one view controller. You might want to leverage the `UINavigationController`'s built-in navigation bar to build a simple utility that uses a two-button menu, for example. This would disregard any navigational advantage of the stack. You still need to set that one controller as the root via `initWithRootViewController:`. Storyboards simplify using navigation controllers for one- and two-button utilities, as you read about in Chapter 4, "Designing Interfaces."

## Pushing and Popping View Controllers

Add new items onto the navigation stack by pushing a new controller with `pushViewController:animated:`. Send this call to the navigation controller that owns a `UIViewController`. This is normally called on `self.navigationController` when you're working with a primary view controller class. When pushed, the new controller slides onscreen from the right (assuming you set `animated` to `YES`). A left-pointing back button appears, leading you one step back on the stack. The back button uses the title of the previous view controller.

There are many reasons you'd push a new view. Typically, these involve navigating to specialty views such as detail views or drilling down a file structure or preferences hierarchy. You can push controllers onto the navigation controller stack after your user taps a button, a table item, or a disclosure accessory.

There's little reason to ever subclass `UINavigationController`. Perform push requests and navigation bar customization (such as setting up a bar's right-hand button) inside `UIViewController` subclasses. For the most part, you don't access the navigation controller directly. The two exceptions to this rule include managing the navigation bar's buttons and changing the bar's look.



You might change a bar style or its tint color by accessing the `navigationBar` property directly:

```
self.navigationController.navigationBar.barStyle =
    UIBarStyleBlackTranslucent;
```

To add a new button, you modify your `navigationItem`, which provides an abstract class that describes the content shown on the navigation bar, including its left and right bar button item and its title view. Here's how you can assign a button to the bar. To remove a button, assign the item to `nil`.

```
self.navigationItem.rightBarButtonItem = [[[UIBarButtonItem alloc]
    initWithTitle:@"Action" style:UIBarButtonItemStylePlain target:self
    action:]] autorelease];
```

Bar button items are not views. They are abstract classes that contain titles, styles, and callback information that are used by navigation items and toolbars to build actual buttons into interfaces. iOS does not provide you with access to the button views built by bar button items and their navigation items.

## The Navigation Item Class

The objects that populate the navigation bar are put into place using the `UINavigationController` class, which is an abstract class that stores information about those objects. Navigation item properties include the left and right bar button items, the title shown on the bar, the view used to show the title, and any back button used to navigate back from the current view.

This class enables you to attach buttons, text, and other UI objects into three key locations: the left, the center, and the right of the navigation bar. Typically, this works out to be a regular button on the right, some text (usually the `UIViewController`'s title) in the middle, and a Back-styled button on the left. But you're not limited to that layout. You can add custom controls to any of these three locations. You can build navigation bars with search fields, segment controls, toolbars, pictures, and more.

You've already seen how to add custom bar button items to the left and right of a navigation item. Adding a custom view to the title is just as simple. Instead of adding a control, assign a view. This code adds a custom `UILabel`, but this could be a `UIImageView`, a `UIStepper`, or anything else:

```
self.navigationItem.titleView = [[[UILabel alloc]
    initWithFrame:CGRectMake(0.0f, 0.0f, 120.0f, 36.0f)] autorelease];
```

The simplest way to customize the actual title is to use the `title` property of the child view controller rather than the navigation item:

```
self.title = @"Hello";
```

When you want the title to automatically reflect the name of the running application, here is a little trick you can use. This returns the short display name defined in the bundle's Info.plist file. Limit using application-specific titles (rather than view-related titles) to simple utility applications.

```
self.title = [[[NSBundle mainBundle] infoDictionary]
              objectForKey:@"CFBundleName"];
```

## Modal Presentation

With normal navigation controllers, you push your way along views, stopping occasionally to pop back to previous views. That approach assumes that you're drilling your way up and down a set of data that matches the tree-based view structure you're using. Modal presentation offers another way to show a view controller. After sending the `presentModalViewController:animated:` message to a navigation controller, a new view controller slides up into the screen and takes control until it's dismissed with `dismissModalViewControllerAnimated:`. This enables you to add special-purpose dialogs into your applications that go beyond alert views.

Typically, modal controllers are used to pick data such as contacts from the Address Book or photos from the Library or to perform a short-lived task such as sending e-mail or setting preferences. Use modal controllers in any setting where it makes sense to perform a limited-time task that lies outside the normal scope of the active view controller.

You can present a modal dialog in any of four ways, controlled by the `modalTransitionStyle` property of the presented view controller. The standard, `UIModalTransitionStyleCoverVertical`, slides the modal view up and over the current view controller. When dismissed it slides back down.

`UIModalTransitionStyleFlipHorizontal` performs a back-to-front flip from right to left. It looks as if you're revealing the back side of the currently presented view. When dismissed, it flips back left to right. `UIModalTransitionStyleCrossDissolve` fades the new view in over the previous one. On dismissal, it fades back to the original view. Use `UIModalTransitionStylePartialCurl` to curl up content (in the way the Maps application does) to reveal a modal settings view "underneath" the primary view controller.

On the iPhone and iPod touch, modal controllers always fully take over the screen. The iPad offers more nuanced presentations. You can introduce modal items using three presentation styles. In addition to the default full-screen style (`UIModalPresentationFullScreen`), use `UIModalPresentationFormSheet` to present a small overlay in the center of the screen or `UIModalPresentationPageSheet` to slide up a sheet in the middle of the screen. These styles are best experienced in landscape mode to visually differentiate the page sheet presentation from the full-screen one.

## Recipe: Building a Simple Two-Item Menu

Although many applications demand serious user interfaces, sometimes you don't need complexity. A simple one- or two-button menu can accomplish a lot in many iOS applications. Navigation controller applications easily lend themselves to a format where instead of pushing and popping children, their navigation bars can be used as basic menus. Use these steps to create a hand-built interface for simple utilities:

1. Create a `UIViewController` subclass that you use to populate your primary interaction space.
2. Allocate a navigation controller and assign an instance of your custom view controller to its root view.
3. In the custom view controller, create one or two button items and add them to the view's navigation item.
4. Build the callback routines that get triggered when a user taps a button.

Recipe 5-1 demonstrates these steps. It creates a simple view controller called `TestBedViewController` and assigns it as the root view for a `UINavigationController`. In the `viewDidLoad` method, two buttons populate the left and right custom slots for the view's navigation item. When tapped, these update the controller's title, indicating which button was pressed. This recipe is not feature rich, but it provides an easy-to-build two-item menu. Figure 5-1 shows the interface in action.

This code uses a handy bar-button-creation macro. When passed a title and a selector, this macro returns a properly initialized bar button item ready to be assigned to a navigation item. (Add `autorelease` to this macro if you're working in MRR code.)

```
#define BARBUTTON(TITLE, SELECTOR) \
    [[UIBarButtonItem alloc] initWithTitle:TITLE \
    style:UIBarButtonItemStylePlain target:self action:SELECTOR]
```

If you're looking for more complexity than two items can offer, consider having the buttons trigger `UIActionSheet` menus and popovers. Action sheets, which are discussed in Chapter 13, "Alerting the User," let users select actions from a short list of options (usually between two and five options, although longer scrolling sheets are possible) and can be seen in use in the Photos and Mail applications for sharing and filing data.

### Note

You can add images instead of text to the `UIBarButtonItem` instances used in your navigation bar. Use `initWithImage:style:target:action:` instead of the text-based initializer.

### Recipe 5-1 Creating a Two-Item Menu Using a Navigation Controller

```
@implementation TestBedViewController
- (void) rightAction: (id) sender
{
    self.title = @"Pressed Right";
```

```

}

- (void) leftAction: (id) sender
{
    self.title = @"Pressed Left";
}

- (void) loadView
{
    [super loadView];
    self.view.backgroundColor = [UIColor whiteColor];
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Right", @selector (rightAction:));
    self.navigationItem.leftBarButtonItem =
        BARBUTTON(@"Left", );
}
@end

```

---

### Get This Recipe's Code

To get the code used for this recipe, go to <https://github.com/erica/iOS-5-Cookbook>, or if you've downloaded the disk image containing all the sample code from the book, go to the folder for Chapter 5 and open the project for this recipe.

## Recipe: Adding a Segmented Control

The preceding recipe showed how to use the two available button slots in your navigation bar to build mini menus. Recipe 5-2 expands on that idea by introducing a six-item `UISegmentedControl` and adding it to a navigation bar's custom title view, as shown in Figure 5-2. When tapped, each item updates the main view with its number.

The key thing to pay attention to in this recipe is the `momentary` attribute assigned to the segmented control. This transforms the interface from a radio button style into an actual menu of options, where items can be selected independently and more than once. So after tapping item three, for example, you can tap it again. That's an important behavior for menu interaction.

Unlike Recipe 5-1, all items in the segmented control trigger the same action (in this case, `segmentAction:`). Determine which action to take by querying the control for its `selectedSegmentIndex` and use that value to create the needed behavior. This recipe updates a central text label. You might want to choose different options based on the segment picked.

### Note

If you want to test this code with the `momentary` property disabled, set the `selectedSegmentIndex` property to match the initial data displayed. In this case, `segment 0` corresponds to the displayed number 1.



Figure 5-2 Adding a segmented control to the custom title view allows you to build a multi-item menu. Notice that no items remain highlighted even after an action takes place. (In this case, the Four button was pressed.)

Segmented controls use styles to specify how they should display. The example here, shown in Figure 5-2, uses a bar style. It is designed for use with bars, as it is in this example. The other two styles (`UISegmentedControlStyleBordered` and `UISegmentedControlStylePlain`) offer larger, more metallic-looking presentations. Of these three styles, only `UISegmentedControlStyleBar` can respond to the `tintColor` changes used in this recipe.

#### Recipe 5-2 Adding a Segmented Control to the Navigation Bar

---

```

- (void) segmentAction: (UISegmentedControl *) segmentedControl
{
    // Update the label with the segment number
    NSString *segmentNumber = [NSString stringWithFormat:@"%0d",
                               segmentedControl.selectedSegmentIndex + 1];
    [(UITextView *)self.view setText:segmentNumber];
}
- (void) loadView
{
    [super loadView];

```

```

// Create a central text view
UITextView *textView = [[UITextView alloc]
    initWithFrame:self.view.frame];
textView.font = [UIFont fontWithName:@"Futura" size:96.0f];
textView.textAlignment = UITextAlignmentCenter;
self.view = textView;

// Create the segmented control
NSArray *buttonNames = [NSArray arrayWithObjects:
    @"One", @"Two", @"Three", @"Four", @"Five", @"Six", nil];
UISegmentedControl* segmentedControl = [[UISegmentedControl alloc]
    initWithItems:buttonNames];
segmentedControl.segmentedControlStyle = UISegmentedControlStyleBar;
segmentedControl.momentary = YES;
[segmentedControl addTarget:self action:
    forControlEvents:UIControlEventValueChanged];

// Add it to the navigation bar
self.navigationItem.titleView = segmentedControl;
}

```

### Get This Recipe's Code

To get the code used for this recipe, go to <https://github.com/erica/iOS-5-Cookbook>, or if you've downloaded the disk image containing all the sample code from the book, go to the folder for Chapter 5 and open the project for this recipe.

## Recipe: Navigating Between View Controllers

In addition to providing menus, navigation controllers do the job they were designed to do: managing hierarchy as you navigate between views. Recipe 5-3 introduces the navigation controller as an actual navigation controller, pushing views on the stack.

The views in this recipe present a number, indicating how many view controllers have been pushed onto the stack. An instance variable stores the current depth number, which is used to both show the current level and decide whether to display a further push option. The maximum depth in this example is 6. In real use, you'd use more meaningful view controllers or contents. This example demonstrates things at their simplest level.

The navigation controller automatically creates the Level 2 back button shown in Figure 5-3 (left) as an effect of pushing the new Level 3 controller onto the stack. The rightmost button (Push) triggers navigation to the next controller by calling `pushViewController:animated:.` When pushed, the next back button reads Level 3, as shown in Figure 5-3 (right).

Back buttons pop the controller stack for you, releasing the current view controller as you move back to the previous one. Make sure your memory management allows that view controller to return all its memory upon being released. Beyond basic memory management, you do not need to program any popping behavior yourself. Note that back

buttons are automatically created for pushed view controllers but not for the root controller itself, because it is not applicable.

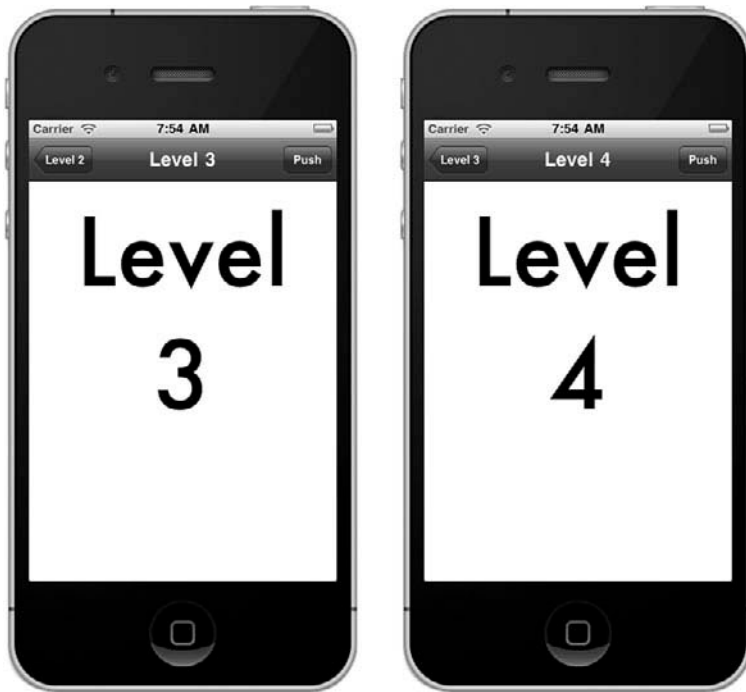


Figure 5-3 The navigation controller automatically creates properly labeled back buttons. After the Level 4 button is selected in the left interface, the navigation controller pushes the Level 4 view controller and creates the Level 3 back button in the right interface.

### Recipe 5-3 Drilling through Views with UINavigationController

---

```
#define IS_IPAD (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad)
```

```
@interface NumberViewController : UIViewController
@property (nonatomic, assign) int number;
@property (nonatomic, strong, readonly) UITextView *textView;
+ (id) controllerWithNumber: (int) number;
@end
```

```
@implementation NumberViewController
@synthesize number, textView;
```

```
// Return a new view controller at the specified level number
+ (id) controllerWithNumber: (int) number
{
    NumberViewController *viewController = [[NumberViewController alloc] init];
    viewController.number = number;
    viewController.textView.text =
        [NSString stringWithFormat:@"Level %d", number];
    return viewController;
}

// Increment and push a controller onto the stack
- (void) pushViewController: (id) sender
{
    NumberViewController *nvc =
        [NumberViewController controllerWithNumber:number + 1];
    [self.navigationController pushViewController:nvc animated:YES];
}

// Set up the text and title as the view appears
- (void) viewDidLoad: (BOOL) animated
{
    self.navigationController.navigationBar.tintColor = COOKBOOK_PURPLE_COLOR;

    // match the title to the text view
    self.title = self.textView.text;
    self.textView.frame = self.view.frame;

    // Add a right bar button that pushes a new view
    if (number < 6)
        self.navigationItem.rightBarButtonItem =
            BARBUTTON(@"Push", );
}

// Create the text view at initialization, not when the view loads
- (id) init
{
    if (!(self = [super init])) return self;

    textView = [[UITextView alloc] initWithFrame:CGRectZero];
    textView.frame = [[UIScreen mainScreen] bounds];
    textView.font =
        [UIFont fontWithName:@"Futura" size:IS_IPAD ? 192.0f : 96.0f];
    textView.textAlignment = UITextAlignmentCenter;
    textView.editable = NO;
    textView.autoresizingMask = self.view.autoresizingMask;
```



```

        return self;
    }

    - (void) loadView
    {
        [super loadView];
        [self.view addSubview:textView];
    }

    - (void) dealloc
    {
        [textView removeFromSuperview];
        textView = nil;
    }
@end

```

---

### Get This Recipe's Code

To get the code used for this recipe, go to <https://github.com/erica/iOS-5-Cookbook>, or if you've downloaded the disk image containing all the sample code from the book, go to the folder for Chapter 5 and open the project for this recipe.

## Recipe: Presenting a Custom Modal Information View

Modal view controllers slide onscreen without being part of your standard view controller stack. Modal views are useful for picking data, updating settings, performing an orthogonal function, or presenting information—tasks that might not match well to your normal hierarchy. Any view controller, including navigation controllers, can present a modal controller as demonstrated in the Chapter 4 walkthroughs. This recipe introduces modal controllers more from a code point of view.

Presenting a modal controller branches off from your primary navigation path, introducing a new interface that takes charge until your user explicitly dismisses it. You present a modal controller like this:

```
[self presentViewController:someControllerInstance animated:YES];
```

The controller that is presented can be any kind of view controller subclass, as well. In the case of a navigation controller, the modal presentation can have its own navigation hierarchy built as a chain of interactions.

Always provide a Done button to allow users to dismiss the controller. The easiest way to accomplish this is to present a navigation controller, adding a bar button to its navigation items. Figure 5-4 shows a modal presentation built around a `UINavigationController` instance using a page-curl presentation. You can see the built-in Done button at the top-right of the presentation.

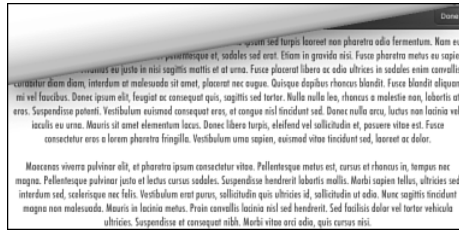


Figure 5-4 This modal view is built using  
UIViewController with a UINavigationController.

In iOS 5.x, modal presentations can use four transition styles:

- **Slide**—This transition style slides a new view over the old.
- **Fade**—This transition style dissolves the new view into visibility.
- **Flip**—This transition style turns a view over to the “back” of the presentation.
- **Curl**—This transition style makes the primary view curl up out of the way to reveal the new view beneath it, as shown in Figure 5–4.

In addition to these transition styles, the iPad offers three presentation styles:

- **Full Screen**—A full-screen presentation is the default on the iPhone, where the new modal view completely covers both the screen and any existing content. This is the only presentation style that is legal for curls—any other presentation style raises a runtime exception, crashing the application.
- **Page Sheet**—In the page sheet, coverage defaults to a portrait aspect ratio, so the modal view controller completely covers the screen in portrait mode and partially covers the screen in landscape mode, as if a portrait-aligned piece of paper were added to the display.
- **Form Sheet**—The form sheet display covers a small center portion of the screen, allowing you to shift focus to the modal element while retaining the maximum visibility of the primary application view.

Your modal view controllers must autorotate. This skeleton demonstrates the simplest possible modal controller you should use. Notice the Interface Builder–accessible `done:` method.

```
@interface ModalViewController : UIViewController
- (IBAction)done:(id)sender;
@end
```

```
@implementation ModalViewController
- (IBAction)done:(id)sender
```

```

{
    [self dismissModalViewControllerAnimated:YES];
}

- (BOOL) shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)toInterfaceOrientation
{
    return YES;
}
@end

```

Storyboards simplify the creation of modal controller elements. Drag in a navigation controller instance, along with its paired view controller, adding a Done button to the provided navigation bar. Set the view controller's class to your custom modal type and connect the Done button to the `done:` method. Make sure you name your navigation controller in the attributes inspector, so you can use that identifier to load it.

You can either add the modal components to your primary storyboard or create them in a separate file. Recipe 5-4 loads a custom file (*Modal~DeviceType.storyboard*) but you can just as easily add the elements in your *MainStoryboard\_DeviceType* file.

Recipe 5-4 offers the key pieces for creating modal elements. The presentation is performed in the application's main view controller hierarchy. Here, users select the transition and presentation styles from segmented controls, but these are normally chosen in advance by the developer and set in code or in IB. This recipe offers a toolbox that you can test out on each platform, using each orientation, to explore how each option looks.

---

#### Recipe 5-4 Presenting and Dismissing a Modal Controller

```

// Presenting the controller
- (void) action: (id) sender
{
    // Load info controller from storyboard
    UIStoryboard *sb = [UINavigationController
        storyboardWithName: (IS_IPAD ? @"Modal~iPad" : @"Modal~iPhone")
        bundle: [NSBundle mainBundle]];
    UINavigationController *navController =
        [sb instantiateViewControllerWithIdentifier:
            @"infoNavigationController"];

    // Select the transition style
    int styleSegment =
        [(UISegmentedControl *)self.navigationItem.titleView
            selectedSegmentIndex];
    int transitionStyles[4] = {
        UIModalTransitionStyleCoverVertical,
        UIModalTransitionStyleCrossDissolve,
        UIModalTransitionStyleFlipHorizontal,

```

```

        UIModalTransitionStylePartialCurl});
navController.modalTransitionStyle = transitionStyles[styleSegment];

// Select the presentation style for iPad only
if (IS_IPAD)
{
    int presentationSegment =
        [(UISegmentedControl *)][self.view subviews]
        lastObject] selectedIndex];
    int presentationStyles[3] = {
        UIModalPresentationFullScreen,
        UIModalPresentationPageSheet,
        UIModalPresentationFormSheet};

    if (navController.modalTransitionStyle ==
        UIModalTransitionStylePartialCurl)
    {
        // Partial curl with any non-full screen presentation
        // raises an exception
        navigationController.modalPresentationStyle =
            UIModalPresentationFullScreen;
        [(UISegmentedControl *)][self.view subviews]
        lastObject] setSelectedSegmentIndex:0];
    }
    else
        navigationController.modalPresentationStyle =
            presentationStyles[presentationSegment];
}

[self.navigationController presentModalViewController:
    navigationController animated:YES];
}

- (void) loadView
{
    [super loadView];
    self.view.backgroundColor = [UIColor whiteColor];
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Action", );

    UISegmentedControl *segmentedControl =
        [(UISegmentedControl alloc) initWithItems:
            [@"Slide Fade Flip Curl" componentsSeparatedByString:@" "]];
    segmentedControl.segmentedControlStyle = UISegmentedControlStyleBar;
    self.navigationItem.titleView = segmentedControl;

```

```

if (IS_IPAD)
{
    NSArray *presentationChoices =
        [NSArray arrayWithObjects:
            @"Full Screen", @"Page Sheet", @"Form Sheet", nil];
    UISegmentedControl *iPadStyleControl =
        [[UISegmentedControl alloc] initWithItems:presentationChoices];
    iPadStyleControl.segmentedControlStyle =
        UISegmentedControlStyleBar;
    iPadStyleControl.autoresizingMask =
        UIViewAutoresizingFlexibleWidth;
    iPadStyleControl.center =
        CGPointMake(CGRectGetMidX(self.view.bounds), 22.0f);
    [self.view addSubview:iPadStyleControl];
}
}

```

---

### Get This Recipe's Code

To get the code used for this recipe, go to <https://github.com/erica/iOS-5-Cookbook>, or if you've downloaded the disk image containing all the sample code from the book, go to the folder for Chapter 5 and open the project for this recipe.

## Recipe: Page View Controllers

This `UIPageViewController` class builds a book-like interface that uses individual view controllers as its pages. Users swipe from one page to the next or tap the edges to move to the next or previous page. All a controller's pages can be laid out in a similar fashion, such as in Figure 5-5, or each page can provide a unique user interaction experience. Apple precooked all the animation and gesture handling into the class for you. You provide the content, implementing delegate and data source callbacks.

### Book Properties

Your code customizes a page view controller's look and behavior. Its key properties specify how many pages are seen at once, the content used for the reverse side of each page, and more. Here's a rundown of those properties:

- The controller's `doubleSided` property determines whether content appears on both sides of a page, as shown in Figure 5-5, or just one side. Reserve the double-sided presentation for side-by-side layout when showing two pages at once. If you don't, you'll end up making half your pages inaccessible. The controllers on the "back" of the pages will never move into the primary viewing space. The book layout is controlled by the book's spine.

- The `spineLocation` property can be set at the left or right, top or bottom, or center of the page. The three spine constants are `UIPageViewControllerSpineLocationMin`, corresponding to top or left, `UIPageViewControllerSpineLocationMax` for the right or bottom, and `UIPageViewControllerSpineLocationMid` for the center. The first two of these produce single-page presentations; the last with its middle spine is used for two-page layouts. Return one of these choices from the `pageViewController:spineLocationForInterfaceOrientation:` delegate method, which is called whenever the device reorients, to let the controller update its views to match the current device orientation.
- Set the `navigationOrientation` property to specify whether the spine goes left/right or top/bottom. Use either `UIPageViewControllerNavigationOrientationHorizontal` (left/right) or `UIPageViewControllerNavigationOrientationVertical` (top/bottom). For a vertical book, the pages flip up and down, rather than employing the left and right flips normally used.
- The `transitionStyle` property controls how one view controller transitions to the next. At the time of writing, the only transition style supported by the page view controller is the page curl, `UIPageViewControllerTransitionStylePageCurl`.

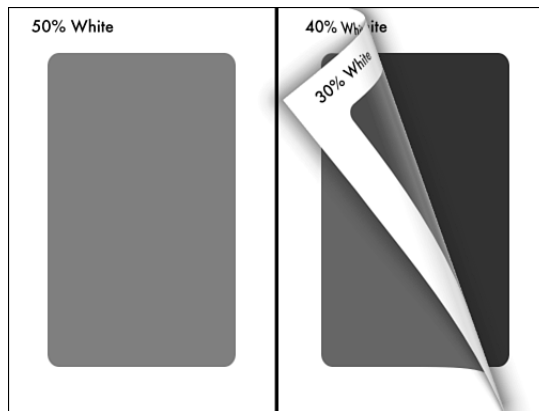


Figure 5-5 The `UIPageViewController` class creates virtual “books” from individual view controllers.

## Wrapping the Implementation

Like table views, page view controllers use a delegate and data source to set the behavior and contents of its presentation. Unlike with table views, I have found that it’s simplest to wrap these items into a custom class to hide their details from my applications. I find the

code needed to support a page view implementation rather quirky—but highly reusable. A wrapper lets you turn your attention away from fussy coding details to specific content-handling concerns.

In the standard implementation, the data source is responsible for providing page controllers on demand. It returns the next and previous view controller in relationship to a given one. The delegate handles reorientation events and animation callbacks, setting the page view controller's controller array, which always consists of either one or two controllers, depending on the view layout. As Recipe 5-5 demonstrates, it's a bit of a mess to implement.

Recipe 5-5 creates a `BookController` class. This class numbers each page, hiding the next/previous implementation details and handling all reorientation events. A custom delegate protocol (`BookDelegate`) becomes responsible for returning a controller for a given page number when sent the `viewControllerForPage:` message. This simplifies implementation so the calling app only has to handle a single method, which it can do by building controllers by hand or by pulling them from a storyboard.

To use the class defined in Recipe 5-5, you must establish the controller, add it as a subview, and declare it as a child view controller, ensuring it receives orientation and memory events. Here's what that code might look like. Notice how the new controller is added as a child to the parent, and its initial page number set:

```
// Establish the page view controller
bookController = [BookController bookWithDelegate:self];
bookController.view.frame = (CGRect){.size = appRect.size};

// Add the child controller, and set it to the first page
[self.view addSubview:bookController.view];
[self addChildViewController:bookController];
[bookController didMoveToParentViewController:self];
[bookController moveToPage:0];
```

## Exploring the Recipe

Recipe 5-5 handles its delegate and data source duties by tagging each view controller's view with a number. It uses this number to know exactly which page is presented at any time and to delegate another class, the `BookDelegate`, to produce a view controller by index.

The page controller itself always stores zero, one, or two pages in its view controller array. Zero pages means the controller has not yet been properly set up. One page is used for spine locations on the edge of the screen; two pages for a central spine. If the page count does not exactly match the spine setup, you will encounter a rather nasty runtime crash.

The controllers stored in those pages are produced by the two data source methods, which implement the before and after callbacks. In the page controller's native implementation, controllers are defined strictly by their relationship to each other, not by an index.

This recipe replaces those relationships with a simple number, asking its delegate for the page at a given index.

Here, the `useSideBySide:` method decides where to place the spine, and thus how many controllers show at once. This implementation sets landscape as side-by-side and portrait as one-page. You may want to change this for your applications. For example, you might use only one page on the iPhone, regardless of orientation, to enhance text readability.

Recipe 5-5 allows both user- and application-based page control. Users can swipe and tap to new pages or the application can send a `moveToPage:` request. This allows you to add external controls in addition to the page view controller's gesture recognizers.

The direction that the page turns is set by comparing the new page number against the old. This recipe uses a Western-style page turn, where higher numbers are to the right and pages flip to the left. You may want to adjust this as needed for countries in the Middle and Far East.

This recipe, as shown here, continually stores the current page to system defaults, so it can be recovered when the application is relaunched. It will also notify its delegate when the user has turned to a given page, which is useful if you add a page slider, as is demonstrated in Recipe 5-6.

#### Recipe 5-5 Creating a Page View Controller Wrapper

---

```
// Define a custom delegate protocol for this wrapper class
@protocol BookControllerDelegate <NSObject>
- (id) viewControllerForPage: (int) pageNumber;
@optional
- (void) bookControllerDidTurnToPage: (NSNumber *) pageNumber;
@end

// A Book Controller wraps the Page View Controller
@interface BookController : UIPageViewController
    <UIPageViewControllerDelegate, UIPageViewControllerDataSource>
+ (id) bookWithDelegate: (id) theDelegate;
+ (id) rotatableViewController;
- (void) moveToPage: (uint) requestedPage;
- (int) currentPage;

@property (nonatomic, weak) id <BookControllerDelegate> bookDelegate;
@property (nonatomic, assign) uint pageNumber;
@end

#pragma Book Controller
@implementation BookController
@synthesize bookDelegate, pageNumber;

#pragma mark Utility
// Page controllers are numbered using tags
```



```

- (int) currentPage
{
    int pageCheck = ((UIViewController *)[self.viewControllers
        objectAtIndex:0]).view.tag;
    return pageCheck;
}

#pragma mark Page Handling
// Update if you'd rather use some other decision style
- (BOOL) useSideBySide: (UIInterfaceOrientation) orientation
{
    BOOL isLandscape = UIInterfaceOrientationIsLandscape(orientation);
    return isLandscape;
}

// Update the current page, set defaults, call the delegate
- (void) updatePageTo: (uint) newPageNumber
{
    pageNumber = newPageNumber;

    [[NSUserDefaults standardUserDefaults]
        setInteger:pageNumber forKey:DEFAULTS_BOOKPAGE];
    [[NSUserDefaults standardUserDefaults] synchronize];

    SAFE_PERFORM_WITH_ARG(bookDelegate,
        ),
        [NSNumber numberWithInt:pageNumber]);
}

// Request controller from delegate
- (UIViewController *) controllerAtPage: (int) aPageNumber
{
    if (bookDelegate && [bookDelegate respondsToSelector:
        ])
    {
        UIViewController *controller =
            [bookDelegate viewControllerForPage:aPageNumber];
        controller.view.tag = aPageNumber;
        return controller;
    }
    return nil;
}

// Update interface to the given page
- (void) fetchControllersForPage: (uint) requestedPage
    orientation: (UIInterfaceOrientation) orientation

```

```

{
    BOOL sideBySide = [self useSideBySide:orientation];
    int numberOfPagesNeeded = sideBySide ? 2 : 1;
    int currentCount = self.viewControllers.count;

    uint leftPage = requestedPage;
    if (sideBySide && (leftPage % 2)) leftPage--;

    // Only check against current page when count is appropriate
    if (currentCount && (currentCount == numberOfPagesNeeded))
    {
        if (pageNumber == requestedPage) return;
        if (pageNumber == leftPage) return;
    }

    // Decide the prevailing direction, check new page against the old
    UIPageViewControllerNavigationDirection direction =
        (requestedPage > pageNumber) ?
        UIPageViewControllerNavigationDirectionForward :
        UIPageViewControllerNavigationDirectionReverse;
    [self updatePageTo:requestedPage];

    // Update the controllers, never adding a nil result
    NSMutableArray *pageControllers = [NSMutableArray array];
    SAFE_ADD(pageControllers, [self controllerAtPage:leftPage]);
    if (sideBySide)
        SAFE_ADD(pageControllers, [self controllerAtPage:leftPage + 1]);
    [self setViewControllers:pageControllers
        direction: direction animated:YES completion:nil];
}

// Entry point for external move request
- (void) moveToPage: (uint) requestedPage
{
    [self fetchControllersForPage:requestedPage
        orientation: (UIInterfaceOrientation)[UIDevice
            currentDevice].orientation];
}

#pragma mark Data Source
- (UIViewController *)pageViewController:
    (UIPageViewController *)pageViewController
viewControllerAfterViewController:
    (UIViewController *)viewController

```

```

{
    [self updatePageTo:pageNumber + 1];
    return [self controllerAtPage:(viewController.view.tag + 1)];
}

- (UIViewController *)pageViewController:
    (UIPageViewController *)pageViewController
    viewControllerBeforeViewController:
        (UIViewController *)viewController
    {
        [self updatePageTo:pageNumber - 1];
        return [self controllerAtPage:(viewController.view.tag - 1)];
    }

#pragma mark Delegate Method
- (UIPageViewControllerSpineLocation)pageViewController:
    (UIPageViewController *) pageViewController
    spineLocationForInterfaceOrientation:
        (UIInterfaceOrientation) orientation
    {
        // Always start with left or single page
        NSUInteger indexOfCurrentViewController = 0;
        if (self.viewControllers.count)
            indexOfCurrentViewController =
                ((UIViewController *) [self.viewControllers
                    objectAtIndex:0]).view.tag;
        [self fetchControllersForPage:indexOfCurrentViewController
            orientation:orientation];

        // Decide whether to present side-by-side
        BOOL sideBySide = [self useSideBySide:orientation];
        self.doubleSided = sideBySide;

        UIPageViewControllerSpineLocation spineLocation = sideBySide ?
            UIPageViewControllerSpineLocationMid :
            UIPageViewControllerSpineLocationMin;
        return spineLocation;
    }

// Return a new book
+ (id) bookWithDelegate: (id) theDelegate
{
    BookController *bc = [[BookController alloc]
        initWithTransitionStyle:
            UIPageViewControllerTransitionStylePageCurl
        navigationOrientation:
            UIPageViewControllerNavigationOrientationHorizontal
    ];
    bc.delegate = theDelegate;
    return bc;
}

```

```

        options:nil];

    bc.dataSource = bc;
    bc.delegate = bc;
    bc.bookDelegate = theDelegate;

    return bc;
}
@end

```

---

### Get This Recipe's Code

To get the code used for this recipe, go to <https://github.com/erica/iOS-5-Cookbook>, or if you've downloaded the disk image containing all the sample code from the book, go to the folder for Chapter 5 and open the project for this recipe.

## Recipe: Scrubbing Pages in a Page View Controller

Manually flipping from page to page quickly becomes tedious, especially when you're working with a presentation of dozens or hundreds of virtual pages. To address this, you can add a slider to your books. Recipe 5-6 creates a slider that appears when the background is tapped and that fades away after a few seconds if not used.

A custom tap gesture recognizer starts the timer, which is reset whenever the user interacts with the slider. Once the timer fires, the slider overview animates away and the user is left with the full-screen page presentation. This approach, using a tap-based overlay, is common to many of Apple's own applications such as the Photos app.

### Recipe 5-6 Adding an Auto-hiding Slider to a Page View Controller

---

```

// Slider callback resets the timer, moves to the new page
- (void) moveToPage: (UISlider *) theSlider
{
    [hiderTimer invalidate];
    hiderTimer = [NSTimer scheduledTimerWithTimeInterval:3.0f
        target:self selector:)
        userInfo:nil repeats:NO];
    [bookController moveToPage:(int) theSlider.value];
}

// BookController Delegate method allows slider value update
- (void) bookControllerDidTurnToPage: (NSNumber *) pageNumber
{
    pageSlider.value = pageNumber.intValue;
}

```

```

// Hide the slider after the timer fires
- (void) hideSlider: (NSTimer *) aTimer
{
    [UIView animateWithDuration:0.3f animations:^(void){
        pageSlider.alpha = 0.0f;});
    [hiderTimer invalidate];
    hiderTimer = nil;
}

// Present the slider when tapped
- (void) handleTap: (UITapGestureRecognizer *) recognizer
{
    [UIView animateWithDuration:0.3f animations:^(void){
        pageSlider.alpha = 1.0f;});
    [hiderTimer invalidate];
    hiderTimer = [NSTimer scheduledTimerWithTimeInterval:3.0f
        target:self selector:)
        userInfo:nil repeats:NO];
}

- (void) viewDidLoad
{
    [super viewDidLoad];

    // Add page view controller as a child view, and do housekeeping
    [self addChildViewController:bookController];
    [self.view addSubview:bookController.view];
    [bookController didMoveToParentViewController:self];
    [self.view addSubview:pageSlider];
}

- (void) loadView
{
    [super loadView];
    CGRect appRect = [[UIScreen mainScreen] applicationFrame];
    self.view = [[UIView alloc] initWithFrame: appRect];
    self.view.backgroundColor = [UIColor whiteColor];
    self.view.autoresizingMask =
        UIViewAutoresizingFlexibleHeight |
        UIViewAutoresizingFlexibleWidth;

    // Establish the page view controller
    bookController = [BookController bookWithDelegate:self];
    bookController.view.frame = (CGRect){.size = appRect.size};
}

```

```
// Set the tap to reveal the hidden slider
UITapGestureRecognizer *tap = [[UITapGestureRecognizer alloc]
    initWithTarget:self action:]);
[self.view addGestureRecognizer:tap];
}
```

---

## Recipe: Tab Bars

On the iPhone and iPod touch, the `UITabBarController` class allows users to move between multiple view controllers and to customize the bar at the bottom of the screen. This is best seen in the YouTube and iPod applications. Both offer one-tap access to different views, and both offer a More button leading to user selection and editing of the bottom bar. Tab bars are not recommended for use as a primary design pattern on the iPad, although Apple supports their use in both split views and popovers when needed.

With tab bars, you don't push views the way you do with navigation bars. Instead, you assemble a collection of controllers (they can individually be `UIViewController`s, `UINavigationController`s, or any other kind of view controllers) and add them into a tab bar by setting the bar's `viewControllers` property. It really is that simple. Cocoa Touch does all the rest of the work for you. Set `allowsCustomizing` to `YES` to enable user reordering of the bar.

Recipe 5-7 creates 11 simple view controllers of the `BrightnessController` class. This class sets its background to a specified gray level—in this case, from 0% to 100% in steps of 10%. Figure 5-5 (left) shows the interface in its default mode, with the first four items and a More button displayed.

Users may reorder tabs by selecting the More option and then tapping Edit. This opens the configuration panel shown in Figure 5-6 (right). These 11 view controllers offer the options a user can navigate through and select from. Readers of earlier editions of this book might note that the Configure title bar's tint finally matches the rest of the interface. Apple introduced the `UIAppearance` protocol, which allows you to customize all instances of a given class. Recipe 5-7 uses this functionality to tint its navigation bars black.

```
[[UINavigationController appearance] setTintColor:[UIColor blackColor]];
```

This recipe adds its 11 controllers twice. The first time it assigns them to the list of view controllers available to the user:

```
tbarController.viewControllers = controllers;
```

The second time it specifies that the user can select from the entire list when interactively customizing the bottom tab bar:

```
tbarController.customizableViewControllers = controllers;
```



Figure 5-6 Tab bar controllers allow users to pick view controllers from a bar at the bottom of the screen (left side of the figure) and to customize the bar from a list of available view controllers (right side of the figure).

The second line is optional; the first is mandatory. After setting up the view controllers, you can add all or some to the customizable list. If you don't, you still can see the extra view controllers using the More button, but users won't be able to include them in the main tab bar on demand.

Tab art appears inverted in color on the More screen. According to Apple, this is the expected and proper behavior. They have no plans to change this. It does provide an interesting view contrast when your 100% white swatch appears as pure black on that screen.

#### Recipe 5-7 Creating a Tab View Controller

```
@interface BrightnessController : UIViewController
{
    int brightness;
}
@end
```

```

@implementation BrightnessController
// Create a swatch for the tab icon using standard Quartz
// and UIKit image calls
- (UIImage*) buildSwatch: (int) aBrightness
{
    CGRect rect = CGRectMake(0.0f, 0.0f, 30.0f, 30.0f);
    UIGraphicsBeginImageContext(rect.size);
    UIBezierPath *path = [UIBezierPath
        bezierPathWithRoundedRect:rect cornerRadius:4.0f];
    [[UIColor blackColor]
        colorWithAlphaComponent:(float) aBrightness / 10.0f] set];
    [path fill];

    UIImage *image = UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();

    return image;
}

// The view controller consists of a background color
// and a tab bar item icon
- (BrightnessController *) initWithBrightness: (int) aBrightness
{
    self = [super init];
    brightness = aBrightness;
    self.title = [NSString stringWithFormat:@"%d%%", brightness * 10];
    self.tabBarItem = [[UITabBarItem alloc] initWithTitle:self.title
        image:[self buildSwatch:brightness] tag:0];
    return self;
}

// Tint the background
- (void) viewDidLoad
{
    [super viewDidLoad];
    self.view.backgroundColor =
        [UIColor colorWithWhite:(brightness / 10.0f) alpha:1.0f];
}

+ (id) controllerWithBrightness: (int) brightness
{
    BrightnessController *controller = [[BrightnessController alloc]
        initWithBrightness:brightness];
    return controller;
}
@end

```



```

#pragma mark Application Setup
@interface TestBedAppDelegate : NSObject
    <UIApplicationDelegate, UITabBarControllerDelegate>
{
    UIWindow *window;
    UITabBarController *tabBarController;
}
@end

@implementation TestBedAppDelegate
- (void)applicationDidFinishLaunching:(UIApplication *)application
{
    [application setStatusBarHidden:YES];
    window = [[UIWindow alloc]
        initWithFrame:[UIScreen mainScreen] bounds]];

    // Globally use a black tint for nav bars
    [[UINavigationBar appearance]
        setTintColor:[UIColor blackColor]];

    // Build an array of controllers
    NSMutableArray *controllers = [NSMutableArray array];
    for (int i = 0; i <= 10; i++)
    {
        BrightnessController *controller =
            [BrightnessController controllerWithBrightness:i];
        UINavigationController *nav =
            [[UINavigationController alloc]
                initWithRootViewController:controller];
        nav.navigationBar.barStyle = UIBarStyleBlackTranslucent;
        [controllers addObject:nav];
    }

    tabBarController = [[RotatingTabBarController alloc] init];
    tabBarController.viewControllers = controllers;
    tabBarController.customizableViewControllers = controllers;
    tabBarController.delegate = self;

    window.rootViewController = tabBarController;
    [window makeKeyAndVisible];
    return YES;
}
@end

```

---

### Get This Recipe's Code

To get the code used for this recipe, go to <https://github.com/erica/iOS-5-Cookbook>, or if you've downloaded the disk image containing all the sample code from the book, go to the folder for Chapter 5 and open the project for this recipe.

## Recipe: Remembering Tab State

On iOS, persistence is golden. When starting or resuming your application from termination or interruption, always return users to a state that closely matches where they left off. This lets your users pick up with whatever tasks they were involved with and provides a user interface that matches the previous session. Recipe 5-8 introduces an example of doing exactly that.

This recipe stores both the current tab order and the currently selected tab, and does so whenever those items are updated. When a user launches the application, the code searches for previous settings and applies them when they are found.

The approach used here depends on two delegate methods. The first, `tabBarController:didEndCustomizingViewControllers:`, provides the current array of view controllers after the user has customized them with the More > Edit screen. This code captures their titles (10%, 20%, and so on) and uses that information to relate a name to each view controller.

The second delegate method is `tabBarController:didSelectViewController:`. The tab bar controller sends this method each time a user selects a new tab. By capturing the `selectedIndex`, this code stores the controller number relative to the current array.

Setting these values depends on using iOS's built-in user defaults system, `NSUserDefaults`. This preferences system works very much as a large mutable dictionary. You can set values for keys using `setObject:forKey:`, as shown here:

```
[[NSUserDefaults standardUserDefaults] setObject:titles
 forKey:@"tabOrder"];
```

Then you can retrieve them with `objectForKey:`, like so:

```
NSArray *titles = [[NSUserDefaults standardUserDefaults]
 objectForKey:@"tabOrder"];
```

Always make sure to synchronize your settings as shown in this code to ensure that the defaults dictionary matches your changes. If you do not synchronize, the defaults may not get set until the application terminates. If you do synchronize, your changes are updated immediately. Any other parts of your application that rely on checking these settings will then be guaranteed to access the latest values.

When the application launches, it checks for previous settings for the last selected tab order and selected tab. If it finds them, it uses these to set up the tabs and select a tab to make active. Because the titles contain the information about what brightness value to show, this code converts the stored title from text to a number and divides that number by ten to send to the initialization function.

Most applications aren't based on such a simple numeric system. Should you use titles to store your tab bar order, make sure you name your view controllers meaningfully and in a way that lets you match a view controller with the tab ordering.

### Note

You could also store an array of the view tags as `NSNumber`s or, better yet, use the `NSKeyedArchiver` class that is introduced in Chapter 8, "Gestures and Touches." Keyed archiving lets you rebuild views using state information that you store on termination.

### Recipe 5-8 Storing Tab State to User Defaults

---

@implementation TestBedAppDelegate

```
- (void)tabBarController:(UITabBarController *)tabBarController
    didEndCustomizingViewControllers:(NSArray *)viewControllers
    changed:(BOOL)changed
{
    // Collect the view controller order
    NSMutableArray *titles = [NSMutableArray array];
    for (UIViewController *vc in viewControllers)
        [titles addObject:vc.title];

    [[NSUserDefaults standardUserDefaults]
     setObject:titles forKey:@"tabOrder"];
    [[NSUserDefaults standardUserDefaults] synchronize];
}

- (void)tabBarController:(UITabBarController *)controller
    didSelectViewController:(UIViewController *)viewController
{
    // Store the selected tab
    NSNumber *tabNumber = [NSNumber numberWithInt:
        [controller selectedIndex]];
    [[NSUserDefaults standardUserDefaults]
     setObject:tabNumber forKey:@"selectedTab"];
    [[NSUserDefaults standardUserDefaults] synchronize];
}

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    [application setStatusBarHidden:YES];
    window = [[UIWindow alloc]
        initWithFrame:[UIScreen mainScreen] bounds]];

    // Globally use a black tint for nav bars
    [[UINavigationController appearance] setTintColor:[UIColor blackColor]];
}
```

```

NSMutableArray *controllers = [NSMutableArray array];
NSArray *titles = [[NSUserDefaults standardUserDefaults]
    objectForKey:@"tabOrder"];

if (titles)
{
    // titles retrieved from user defaults
    for (NSString *theTitle in titles)
    {
        BrightnessController *controller =
            [BrightnessController controllerWithBrightness:
                ([theTitle intValue] / 10)];
        UINavigationController *nav =
            [[UINavigationController alloc]
                initWithRootViewController:controller];
        nav.navigationBar.barStyle = UIBarStyleBlackTranslucent;
        [controllers addObject:nav];
    }
}
else
{
    // generate all new controllers
    for (int i = 0; i <= 10; i++)
    {
        BrightnessController *controller =
            [BrightnessController controllerWithBrightness:i];
        UINavigationController *nav =
            [[UINavigationController alloc]
                initWithRootViewController:controller];
        nav.navigationBar.barStyle = UIBarStyleBlackTranslucent;
        [controllers addObject:nav];
    }
}

tabBarController = [[RotatingTabController alloc] init];
tabBarController.viewControllers = controllers;
tabBarController.customizableViewControllers = controllers;
tabBarController.delegate = self;

// Restore any previously selected tab
NSNumber *tabNumber = [[NSUserDefaults standardUserDefaults]
    objectForKey:@"selectedTab"];
if (tabNumber)
    tabBarController.selectedIndex = [tabNumber intValue];

```

```
    window.rootViewController = tabBarController;  
    [window makeKeyAndVisible];  
    return YES;  
}  
@end
```

### Get This Recipe's Code

To get the code used for this recipe, go to <https://github.com/erica/iOS-5-Cookbook>, or if you've downloaded the disk image containing all the sample code from the book, go to the folder for Chapter 5 and open the project for this recipe.

## Recipe: Building Split View Controllers

Split view controllers provide the preferred way to present hierarchically driven navigation on the iPad. They generally consist of a table of contents on the left and a detail view on the right, although the class (and Apple's guidelines) is not limited to this presentation style. The heart of the class consists of the notion of an organizing section and a presentation section, both of which can appear onscreen at once in landscape orientation, and whose organizing section converts to a bar-button-launched popover in portrait orientation.

Figure 5-7 shows the very basic split view controller built by Recipe 5-8 in landscape and portrait orientations. This controller adjusts the brightness of the detail view by selecting an item from the list in the root view. In landscape, both views are shown at once. In portrait orientation, the user must tap the upper-left button on the detail view to access the root view in a popover. When programming for this orientation, be aware that the popover can interfere with detail view, as it is presented over that view; design accordingly.

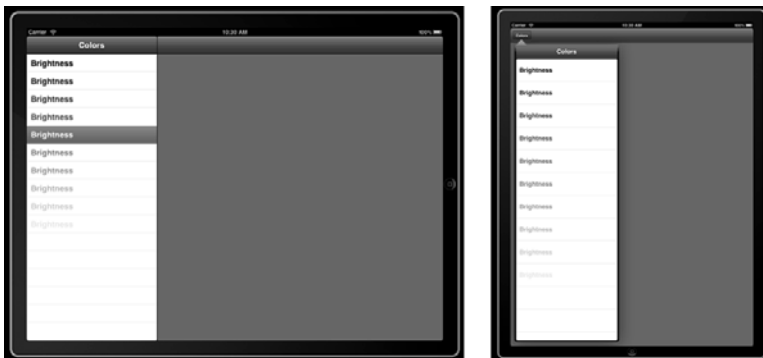


Figure 5-7 At their simplest, split view controllers consist of an organizing pane and a detail view pane. The organizing pane, which is hidden in portrait orientation, can be viewed from a popover accessed from the navigation bar.

Accomplishing this requires three separate objects: the root and detail view controllers, and building the split view controller. What's more, you'll generally want to add the root and detail controllers to navigation controller shells, to provide a consistent interface. In the case of the detail controller, this provides a home for the bar button in portrait orientation. The following method builds the two child views, embeds them into navigation controllers, adds them to a view controller array, and returns a new split view controller that hosts those views:

```
- (UISplitViewController *) splitViewController
{
    // Create the navigation-run root view
    ColorViewController *rootVC = [ColorViewController controller];
    UINavigationController *rootNav = [[UINavigationController alloc]
        initWithRootViewController:rootVC];

    // Create the navigation-run detail view
    DetailViewController *detailVC = [DetailViewController controller];
    UINavigationController *detailNav = [[UINavigationController alloc]
        initWithRootViewController:detailVC];

    // Add both to the split view controller
    UISplitViewController *svc =
        [[UISplitViewController alloc] init];
    svc.viewControllers = [NSArray arrayWithObjects:
        rootNav, detailNav, nil];
    svc.delegate = detailVC;

    return svc;
}
```

The root view controller is typically some kind of table view controller, as is the one in Recipe 5-8. Tables view controllers are discussed in great detail in Chapter 11, “Creating and Managing Table Views,” but what you see here is pretty much as bare bones as they get. It is a list of ten items, each one with a cell title that is tinted proportionally between 0% and 90% of pure white.

When an item is selected, the controller uses its built-in `splitViewController` property to affect its detail view. This property returns the split view controller that owns the root view. From there, the controller can retrieve the split view's `delegate`, which has been assigned to the detail view. By casting that delegate to the detail view controller's class, the root view can affect the detail view more meaningfully. In this extremely simple example, the selected cell's text tint is applied to the detail view's background color.

### Note

Make sure you set the root view controller's `title` property. It is used to set the text for the button that appears in the detail view during portrait mode.

Recipe 5-9's `DetailViewController` class is about as skeletal an implementation as you can get. It provides the most basic functionality you need in order to provide a detail view implementation with split view controllers. This consists of the will-hide/will-show method pair that adds and hides that all-important bar button for the detail view.

When the split view controller converts the root view controller into a popover controller in portrait orientation, it passes that new controller to the detail view controller. It is the detail controller's job to retain and handle that popover until the interface returns to landscape orientation. In this skeletal class definition, a retained property holds onto the popover for the duration of portrait interaction.

#### Recipe 5-9 Building Detail and Root Views for a Split View Controller

---

```
@interface DetailViewController : UIViewController
    <UIPopoverControllerDelegate, UISplitViewControllerDelegate>
{
    UIPopoverController *popoverController;
}
@property (nonatomic, retain) UIPopoverController *popoverController;
@end

@implementation DetailViewController
@synthesize popoverController;

+ (id) controller
{
    DetailViewController *controller =
        [[DetailViewController alloc] init];
    controller.view.backgroundColor = [UIColor blackColor];
    return controller;
}

// Called upon going into portrait mode, hiding the normal table view
- (void)splitViewController: (UISplitViewController*)svc
    willHideViewController: (UIViewController *)aViewController
    withBarButtonItem: (UIBarButtonItem*)barButtonItem
    forPopoverController: (UIPopoverController*)aPopoverController
{
    barButtonItem.title = aViewController.title;
    self.navigationItem.leftBarButtonItem = barButtonItem;
    self.popoverController = aPopoverController;
}

// Called upon going into landscape mode.
- (void)splitViewController: (UISplitViewController*)svc
    willShowViewController: (UIViewController *)aViewController
    invalidatingBarButtonItem: (UIBarButtonItem *)barButtonItem
```

```

{
    self.navigationItem.leftBarButtonItem = nil;
    self.popoverController = nil;
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation
{
    return YES;
}
@end

@interface ColorViewController : UITableViewController
@end
@implementation ColorViewController
+ (id) controller
{
    ColorViewController *controller =
        [[ColorViewController alloc] init];
    controller.title = @"Colors";
    return controller;
}

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
{
    return 10;
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:@"generic"];
    if (!cell) cell = [[UITableViewCell alloc]
        initWithStyle:UITableViewCellStyleDefault
        reuseIdentifier:@"generic"];

    cell.textLabel.text = @"Brightness";
    cell.textLabel.textColor =
        [UIColor colorWithWhite:(indexPath.row / 10.0f) alpha:1.0f];

```



```

        return cell;
    }

- (void)tableView:(UITableView *)tableView
  didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    // On selection, update the main view background color
    UIViewController *controller =
        (UIViewController *)self.splitViewController.delegate;
    UITableViewCell *cell = [tableView cellForRowAtIndexPath:indexPath];
    controller.view.backgroundColor = cell.textLabel.textColor;
}
@end

```

---

### Get This Recipe's Code

To get the code used for this recipe, go to <https://github.com/erica/iOS-5-Cookbook>, or if you've downloaded the disk image containing all the sample code from the book, go to the folder for Chapter 5 and open the project for this recipe.

## Recipe: Creating Universal Split View/Navigation Apps

Recipe 5-10 modifies Recipe 5-9's split view controller to provide a functionally equivalent application that runs properly on both iPhone and iPad platforms. Accomplishing this takes several steps that add to Recipe 5-9's code base. You do not have to remove functionality from the split view controller approach but you must provide alternatives in several places.

Recipe 5-10 depends on a macro that is used throughout which determines whether the code is being run on an iPad- or iPhone-style device:

```
#define IS_IPAD (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad)
```

This macro returns YES when the device characteristics are iPad-like, rather than being iPhone-like (such as on the iPhone or iPod touch.) First introduced in iOS 3.2, idioms allow you to perform runtime checks in your code to provide interface choices that match the deployed platform.

In an iPhone deployment, the detail view controller remains code identical to Recipe 5-9, but to be displayed it must be pushed onto the navigation stack rather than shown side-by-side in a split view. The navigation controller is set up as the primary view for the application window rather than the split view. A simple check at application launch lets your code choose which approach to use:

```

- (UINavigationController *) navWithColorViewController
{
    ColorViewController *colorViewController =

```

```

        [ColorViewController controller];
    UINavigationController *nav = [[UINavigationController alloc]
        initWithRootViewController:colorViewController];
    return nav;
}

- (void)applicationDidFinishLaunching:(UIApplication *)application
{
    window = [[UIWindow alloc] initWithFrame:
        [[UIScreen mainScreen] bounds]];

    if (IS_IPAD)
        window.rootViewController = [self splitviewController];
    else
        window.rootViewController = [self navWithColorViewController];

    [window addSubview:mainController.view];
    [window makeKeyAndVisible];
}

```

The rest of the story lies in the two methods of Recipe 5-10, within the color-picking table view controller. Two key checks decide whether to show disclosure accessories and how to respond to table taps:

- On the iPad, disclosure indicators should never be used at the last level of detail presentation. On the iPhone, they indicate that a new view will be pushed on selection. Checking for deployment platform lets your code choose whether or not to include these accessories in cells.
- When you're working with the iPhone, there's no option for using split views, so your code must push a new detail view onto the navigation controller stack. Compare this to the iPad code, which only needs to reach out to an existing detail view and update its background color.

In real-world deployment, these two checks would likely expand in complexity beyond the details shown in this simple recipe. You'd want to add a check to your model to determine if you are, indeed, at the lowest level of the tree hierarchy before suppressing disclosure accessories. Similarly, you may need to update or replace presentations in your detail view controller.

#### Recipe 5-10 Adding Universal Support for Split View Alternatives

```

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:@"generic"];
    if (!cell) cell = [[UITableViewCell alloc]

```

```

        initWithStyle: UITableViewCellStyleDefault
        reuseIdentifier:@"generic"];

    cell.textLabel.text = @"Brightness";
    cell.textLabel.textColor =
        [UIColor colorWithWhite:(indexPath.row / 10.0f) alpha:1.0f];

    cell.accessoryType = IS_IPAD ?
        UITableViewCellAccessoryNone :
        UITableViewCellAccessoryDisclosureIndicator;

    return cell;
}

- (void)tableView:(UITableView *)tableView
  didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView cellForRowAtIndexPath:indexPath];

    if (IS_IPAD)
    {
        UIViewController *controller =
            (UIViewController *)self.splitViewController.delegate;
        controller.view.backgroundColor = cell.textLabel.textColor;
    }
    else
    {
        DetailViewController *controller = [
            DetailViewController controller];
        controller.view.backgroundColor = cell.textLabel.textColor;
        [self.navigationController
            pushViewController:controller animated:YES];
    }
}

```

---

### Get This Recipe's Code

To get the code used for this recipe, go to <https://github.com/erica/iOS-5-Cookbook>, or if you've downloaded the disk image containing all the sample code from the book, go to the folder for Chapter 5 and open the project for this recipe.

## Recipe: Custom Containers and Segues

Apple's split view controller was groundbreaking in that it introduced the notion that more than one controller could live onscreen at a time. Until the split view, the rule was one controller with many views at a time. With split view, several controllers co-existed onscreen, all of them independently responding to orientation and memory events.

Apple exposed this multiple-controller paradigm to developers in the iOS 5 SDK. You can now create a parent controller and add child controllers to it. Events are passed from parent to child as needed. This allows you to build custom containers, outside of the Apple-standard set of containers such as tab bar and navigation controllers. Here is how you might load children from a storyboard and add them to a custom array of child view controllers:

```

UIStoryboard *aStoryboard = [UIStoryboard storyboardWithName:@"child"
    bundle:[NSBundle mainBundle]];
childControllers = [NSArray arrayWithObjects:
    [aStoryboard instantiateViewControllerWithIdentifier:@"0"],
    [aStoryboard instantiateViewControllerWithIdentifier:@"1"],
    [aStoryboard instantiateViewControllerWithIdentifier:@"2"],
    [aStoryboard instantiateViewControllerWithIdentifier:@"3"],
    nil];

// Set each child as a child view controller, setting its frame
for (UIViewController *controller in childControllers)
{
    controller.view.frame = backsplash.bounds;
    [self addChildViewController:controller];
}

```

With custom containers comes their little brother, custom segues. Just as tab and navigation controllers provide a distinct way of transitioning between child controllers, you can build custom segues that define animations unique to your class. There's not a lot of support in Interface Builder for custom containers with custom segues, so it's best to develop your presentations in code at this time. Here's how you might implement the code that moves the controller to a new view:

```

// Informal delegate method
- (void) segueDidComplete
{
    pageControl.currentPage = vcIndex;
}

// Transition to new view using custom segue
- (void) switchToView: (int) newIndex
    goingForward: (BOOL) goesForward
{
    if (vcIndex == newIndex) return;

    // Segue to the new controller
    UIViewController *source =
        [childControllers objectAtIndex:vcIndex];
    UIViewController *destination =
        [childControllers objectAtIndex:newIndex];
}

```

```

RotatingSegue *segue = [[RotatingSegue alloc]
    initWithIdentifier:@"segue"
    source:source destination:destination];
segue.goesForward = goesForward;
segue.delegate = self;
[segue perform];

vcIndex = newIndex;
}

```

Here, the code identifies the source and destination child controllers, builds a segue, sets its parameters, and tells it to perform. An informal delegate method is called back by that custom segue on its completion. Recipe 5-11 shows how that segue is built. In this example, it creates a rotating cube effect that moves from one view to the next. Figure 5-8 shows the segue in action.



Figure 5-8 Custom segues allow you to create visual metaphors for your custom containers. Recipe 5-11 builds a “cube” of view controllers that can be rotated from one to the next.

The segue’s `goesForward` property determines whether the rotation moves to the right or left around the virtual cube. Although this example uses four view controllers, as you saw in the code that laid out the child view controllers, that’s a limitation of the

metaphor, not of the code itself, which will work with any number of child controllers. You can just as easily build three- or seven-sided presentations with this, although you are breaking an implicit “reality” contract with your user if you do so. To add more (or fewer) sides, you should adjust the animation geometry in the segue away from a cube to fit your virtual  $n$ -hedron.

### Recipe 5-11 Creating a Custom View Controller Segue

---

```
@implementation RotatingSegue

@synthesize goesForward;
@synthesize delegate;

// Return a shot of the given view
- (UIImage *)screenShot: (UIView *) aView
{
    // Arbitrarily dims to 40%. Adjust as desired.
    UIGraphicsBeginImageContext(hostView.frame.size);
    [aView.layer renderInContext:UIGraphicsGetCurrentContext()];
    UIImage *image = UIGraphicsGetImageFromCurrentImageContext();
    CGContextSetRGBFillColor(UIGraphicsGetCurrentContext(),
        0, 0, 0, 0.4f);
    CGContextFillRect (UIGraphicsGetCurrentContext(), hostView.frame);
    UIGraphicsEndImageContext();
    return image;
}

// Return a layer with the view contents
- (CALayer *) createLayerFromView: (UIView *) aView
    transform: (CATransform3D) transform
{
    CALayer *imageLayer = [CALayer layer];
    imageLayer.anchorPoint = CGPointMake(1.0f, 1.0f);
    imageLayer.frame = (CGRect){.size = hostView.frame.size};
    imageLayer.transform = transform;
    UIImage *shot = [self screenShot:aView];
    imageLayer.contents = (__bridge id) shot.CGImage;

    return imageLayer;
}

// On starting the animation, remove the source view
- (void)animationDidStart:(CAAnimation *)animation
{
    UIViewController *source =
        (UIViewController *) super.sourceViewController;
    [source.view removeFromSuperview];
}
```

```

// On completing the animation, add the destination view,
// remove the animation, and ping the delegate
- (void)animationDidStop:(CAAnimation *)animation finished:(BOOL)finished
{
    UIViewController *dest =
        (UIViewController *) super.destinationViewController;
    [hostView addSubview:dest.view];
    [transformationLayer removeFromSuperlayer];
    if (delegate)
        SAFE_PERFORM_WITH_ARG(delegate,
                               @selector(seguedDidComplete), nil);
}

// Perform the animation
- (void)animateWithDuration: (CGFloat) aDuration
{
    CAAAnimationGroup *group = [CAAnimationGroup animation];
    group.delegate = self;
    group.duration = aDuration;

    CGFloat halfWidth = hostView.frame.size.width / 2.0f;
    float multiplier = goesForward ? -1.0f : 1.0f;

    // Set the x, y, and z animations
    CABasicAnimation *translationX = [CABasicAnimation
        animationWithKeyPath:@"sublayerTransform.translation.x"];
    translationX.toValue =
        [NSNumber numberWithFloat:multiplier * halfWidth];

    CABasicAnimation *translationZ = [CABasicAnimation
        animationWithKeyPath:@"sublayerTransform.translation.z"];
    translationZ.toValue = [NSNumber numberWithFloat:-halfWidth];

    CABasicAnimation *rotationY = [CABasicAnimation
        animationWithKeyPath:@"sublayerTransform.rotation.y"];
    rotationY.toValue = [NSNumber numberWithFloat: multiplier * M_PI_2];

    // Set the animation group
    group.animations = [NSArray arrayWithObjects:
        rotationY, translationX, translationZ, nil];
    group.fillMode = kCAFillModeForwards;
    group.removedOnCompletion = NO;

    // Perform the animation
    [CATransaction flush];
    [transformationLayer addAnimation:group forKey:kAnimationKey];
}

```

```
- (void) constructRotationLayer
{
    UIViewController *source =
        (UIViewController *) super.sourceViewController;
    UIViewController *dest =
        (UIViewController *) super.destinationViewController;
    hostView = source.view.superview;

    // Build a new layer for the transformation
    transformationLayer = [CALayer layer];
    transformationLayer.frame = hostView.bounds;
    transformationLayer.anchorPoint = CGPointMake(0.5f, 0.5f);
    CATransform3D sublayerTransform = CATransform3DIdentity;
    sublayerTransform.m34 = 1.0 / -1000;
    [transformationLayer setSublayerTransform:sublayerTransform];
    [hostView.layer addSublayer:transformationLayer];

    // Add the source view, which is in front
    CATransform3D transform = CATransform3DMakeIdentity;
    [transformationLayer addSublayer:
        [self createLayerFromView:source.view
            transform:transform]];

    // Prepare the destination view either to the right or left
    // at a 90/270 degree angle off the main
    transform = CATransform3DRotate(transform, M_PI_2, 0, 1, 0);
    transform = CATransform3DTranslate(transform,
        hostView.frame.size.width, 0, 0);
    if (!goesForward)
    {
        transform = CATransform3DRotate(transform, M_PI_2, 0, 1, 0);
        transform = CATransform3DTranslate(transform,
            hostView.frame.size.width, 0, 0);
        transform = CATransform3DRotate(transform, M_PI_2, 0, 1, 0);
        transform = CATransform3DTranslate(transform,
            hostView.frame.size.width, 0, 0);
    }
    [transformationLayer addSublayer:
        [self createLayerFromView:dest.view transform:transform]];
}

// Standard UIStoryboardSegue perform
- (void)perform
{
    [self constructRotationLayer];
    [self animateWithDuration:0.5f];
}

@end
```

---



### Get This Recipe's Code

To get the code used for this recipe, go to <https://github.com/erica/iOS-5-Cookbook>, or if you've downloaded the disk image containing all the sample code from the book, go to the folder for Chapter 5 and open the project for this recipe.

## Transitioning Between View Controllers

UIKit offers a simple way to animate view features when you move from one child view controller to another. You provide a source view controller, a destination, and a duration for the animated transition. You can specify the kind of transition in the options. Supported transitions include page curls, dissolves, and flips. This method creates a simple curl from one view controller to the next:

```
- (void) action: (id) sender
{
    [self transitionFromViewController:redController
      toViewController:blueController
      duration:1.0f
      options:UIViewAnimationOptionLayoutSubviews |
              UIViewAnimationOptionTransitionCurlUp
      animations:^(void){}
      completion:^(BOOL finished){
        [redController.view removeFromSuperview];
        [self.view addSubview:blueController.view];}
    ];
}
```

You can use the same approach to animate UIView properties without the built-in transitions. For example, this method re-centers and fades out the red controller while fading in the blue. These are all animatable UIView features and are changed in the `animations: block`.

```
- (void) action: (id) sender
{
    blueController.view.alpha = 0.0f;
    [self transitionFromViewController:redController
      toViewController:blueController
      duration:2.0f
      options:UIViewAnimationOptionLayoutSubviews
      animations:^(void){
        redController.view.center = CGPointMake(0.0f, 0.0f);
        redController.view.alpha = 0.0f;
        blueController.view.alpha = 1.0f;}
      completion:^(BOOL finished){
        [redController.view removeFromSuperview];
      }
    ];
}
```

```

        [self.view addSubview:blueController.view];
    }
}

```

Using transitions and view animations is an either/or scenario. Either set a transition option *or* change view features in the animations block. Otherwise, they conflict, as you can easily confirm for yourself.

Use the completion block to remove the old view and move the new view into place. You should not have to explicitly call `didMoveToParentViewController:` or any of the related, contained view controller methods.

Although simple to implement, this kind of transition is not meant for use with Core Animation. If you wish to add Core Animation effects to your view-controller-to-view-controller transitions, look at using a custom segue instead.

## One More Thing: Interface Builder and Tab Bar Controllers

Xcode offers easy-to-customize tab bar instances that get you started building tab-bar-based GUIs in Interface Builder. By default, this object creates two child view controllers in the storyboard. You can expand this basic presentation by adding new view controllers to the tab bar controller and/or setting classes using the identity inspector (see Figure 5-9).

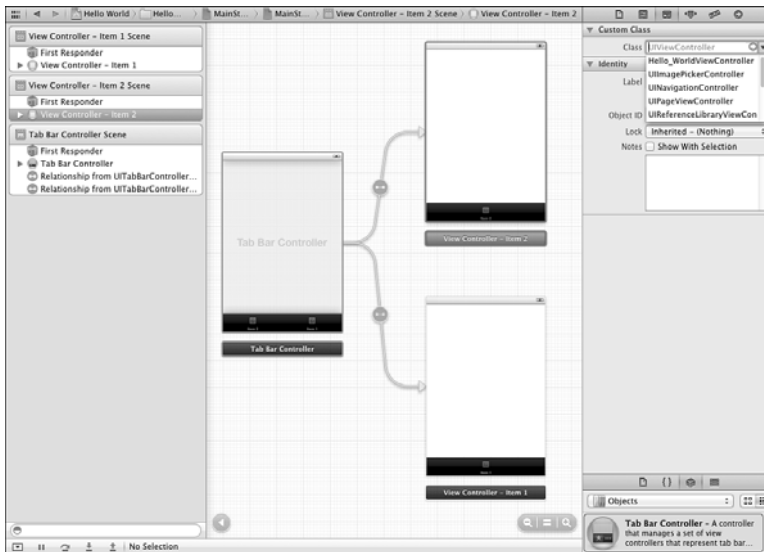


Figure 5-9 Interface Builder storyboards provide tools for laying out tab bar controllers, simplifying laying out what is essentially a logical and not a visual class, compared to what previous versions of Xcode allowed.

You'll likely want to create a new view controller class for each tab, to allow each tab to offer a separate and meaningful function. To add art to the tabs in IB, drag 20×20 PNG images from the Library > Media pane onto each tab button, as shown mid-drag in Figure 5-10, or set the art using the tab bar item's attribute inspector. The Media pane lists the images you have added to your Xcode project. Design your images using a transparent background and a white foreground.



Figure 5-10 Drag art from the media library directly onto the tab bar item shown below each child view controller.

Interface Builder's new storyboards offer a friendly way to both lay out individual view controllers and connect them to their parents. This is a vast change from previous versions of Xcode, where many developers found themselves forgoing IB to design and deploy tab bars and navigation bars in code.

## Summary

This chapter showed many view controller classes in action. You learned how to use them to handle view presentation and user navigation for various device deployment choices. With these classes, you discovered how to expand virtual interaction space and create multipage interfaces as demanded by applications, while respecting the human interface guidelines on the platform in question. Before moving on to the next chapter, here are a few points to consider about view controllers:

- Use navigation trees to build hierarchical interfaces. They work well for looking at file structures or building a settings tree. When you think “disclosure view” or “preferences,” consider pushing a new controller onto a navigation stack or using a split view to present them directly.
- Don't be afraid to use conventional UI elements in unconventional ways so long as you respect the overall Apple Human Interface Guidelines. Parts of this chapter covered innovative uses for the `UINavigationController` that didn't involve any navigation. The tools are there for the using.
- Be persistent. Let your users return to the same GUI state that they last left from. `NSUserDefaults` provides a built-in system for storing information between application runs. Use these defaults to re-create the prior interface state.

- Go universal. Let your code adapt itself for various device deployments rather than forcing your app into an only-iPhone or only-iPad design. This chapter touched on some simple runtime device detection and interface updates that you can easily expand for more challenging circumstances. Universal deployment isn't just about stretching views and using alternate art and .xib files. It's also about detecting when a device influences the *way* you interact, not just the look of the interface.
- When working with custom containers, don't be afraid of using storyboards directly. You do not have to build and retain an array of all your controllers at once. Storyboards offer direct access to all your elements, letting you move past the controller setting you use in tab bars and mimicked in Recipe 5-11. Like the new page view controller class, just load the controllers you need, when you need them.
- Interface Builder's new storyboards provide a welcome new way to set up navigation controllers, tab bars, and more. They are a great innovation on Apple's part and are sure to simplify many design tasks for you.

*This page intentionally left blank*

# Index

## Symbols

---

@ (at) symbol, 53, 65

^ (caret), 85

+ (plus sign), 137

## A

---

### **acceleration**

- detecting shakes with, 683–686

- locating “up,” 668–672

  - basic orientation, 671–672

  - calculating relative angle, 671

  - catching acceleration events, 669

  - retrieving current accelerometer  
angle synchronously, 670

- moving onscreen objects, 672–676

**accelerometer key, 663**

**accelerometer:didAccelerate: method, 668**

**AccelerometerHelper, 683–686**

**accessing AVFoundation camera, 359–368**

- building camera helper, 367–368

- camera previews, 364

- establishing camera session, 361–363

- EXIF, 365

- image geometry, 365–367

- laying out camera previews, 364–365

- querying and retrieving camera, 360–361
- requiring cameras, 360
- switching cameras, 363

#### **accounts, GitHub**

#### **action sheets. *See also* alerts**

- creating, 646–648
- displaying text in, 648–649
- scrolling, 648

#### **actions**

- action names for undo and redo, 422
- adding to iOS-based temperature converter, 223
- connecting buttons to, 451–452
- distribution, 179–181

#### **ActivityAlert, 639–642**

#### **ad hoc distribution, 182–183**

#### **additional device information, recovering, 664–665**

#### **addObjects method, 616**

#### **addressFromString: method, 702**

#### **ad-hoc packages, building, 183**

#### **adjusting**

- retain counts, 100
- views around keyboards, 495–498

#### **affine transforms, 319–320**

#### **alerts, 633. *See also* action sheets; progress indicators**

- alert delegates, 634–636
- alert indicators, 654
- audio alerts, 654–655
  - alert sound, 656
  - delays, 656–658
  - system sounds, 655–656
  - vibration, 656

#### **badging applications, 654**

#### **building, 633–634**

#### **custom overlays, 649–650**

#### **displaying, 636**

#### **local notifications, 652–653**

#### **modal alerts with run loops, 642–645**

#### **no-button alerts, 639–642**

#### **popovers, 650–652**

#### **tappable overlays, 650**

#### **types of alerts, 636–637**

#### **variadic arguments, 645–646**

#### **volume alert, 658**

#### **alertView.clickedButtonAtIndex: method, 635**

#### **alertViewStyle property, 636**

#### **algorithmically sorting tables, 580–581**

#### **AllEditingEvents event, 447**

#### **AllEvents event, 447**

#### **alloc method, 55**

#### **allocating memory, 54–55**

#### **allowsEditing property, 344**

#### **AllTouchEvent event, 447**

#### **alpha property, 321**

#### **alternating cell colors, 565–566**

#### **analyzing code, 165**

#### **Anderson, Fritz, 126**

#### **animations**

- in buttons, 456–458
- custom containers and segues, 284–290
- transitioning between view controllers, 290–291
- view animations, 321–324
  - blocks approach, 323–324
  - bouncing views, 329–331

- building transactions, 322–323
- conditional animation, 324
- Core Animation Transitions, 328–329
- fading in/out, 324–326
- flipping views, 327
- image view animations, 331–332
- swapping views, 326–327
- App Store, submitting to, 186–187**
- appearance proxies, 460–465**
- application badges, 654**
- application delegates, 28–30**
- application identifiers**
  - inspecting, 172
  - registering, 21–22
- applicationDidBecomeActive: method, 28**
- applicationDidFinishLaunching: method, 423**
- application:didFinishLaunchingWithOptions: method, 28, 653**
- applicationDidReceiveMemoryWarning method, 30**
- applicationIconBadgeNumber property, 654**
- ApplicationReserved event, 447**
- applications**
  - application bundle
    - executables, 32–33
    - icon images, 34–36
    - image storage in, 337
    - Info.plist file, 33–34
    - Interface Builder files, 37
  - application skeleton, 25–26, 34–36
  - autorelease pools, 27
  - main.m file, 26–27
  - UIApplicationMain function, 27–28
  - compiled applications, signing, 175
  - folder hierarchy, 32
  - IPA archives, 38
  - limits, 17–18
  - opening images in, 338
  - Organizer, 169
  - running
    - Hello World, 141
    - for storyboard interfaces, 216
  - sandboxes, 38–39
- applicationSupportsShakeToEdit property, 423**
- applicationWillEnterBackground: method, 28**
- applicationWillEnterForeground: method, 28**
- applicationWillResignActive: method, 28**
- ARC (automatic reference counting), 55**
  - autorelease pools, 94–95
  - bypassing, 97–98
  - casting between Objective-C and Core Foundation, 99
    - adjusting retain counts, 100
    - basic casting, 99–100
    - choosing bridging approach, 101
    - conversion issues, 102–103
    - retains, 101
    - runtime workarounds, 102
    - transfers, 100–101
  - deallocation, 84
  - disabling
    - across a target, 96
    - on file-by-file basis, 97
  - memory management, 70–71
  - migrating to, 95–96
  - qualifiers, 77, 89



- autoreleased qualifiers, 91-92
  - strong and weak properties, 89-90
  - variable qualifiers, 90-91
- reference cycles, 92-94
- rules, 98-99
- tips and tricks, 103
- archiving, 416-418**
- armv6 key, 663**
- armv7 key, 663**
- arrays, 58-59**
  - building, 76-118
  - checking, 118
  - converting into strings, 118
  - converting strings to, 112
- arrayWithContentsOfFile: method, 120**
- arrayWithObjects: method, 63, 72, 117**
- art, adding to buttons, 450-451**
- assigning**
  - block preferences, 85-87
  - data sources to tables, 556-557
  - delegates to tables, 558
- associated objects, 304-305**
- asynchronous downloads**
  - download helper, 715-721
  - NSURLConnectionDownload Delegate protocol, 713-714
- at (@) symbol, 53, 65**
- atomic qualifiers, 77-78**
- attitude property, 676**
- attributed strings**
  - automatically parsing markup text into, 532-535
  - building, 526-532
  - extensions library, 532
- audio alerts, 654-655**
  - alert sound, 656
  - audio platform differences, 10
  - delays, 656-658
  - system sounds, 655-656
  - vibration, 656
  - volume alert, 658
- Audio Queue, 655**
- AudioServicesAddSystemSoundCompletion method, 655**
- AudioServicesCreateSystemSoundID method, 655**
- AudioServicesPlayAlertSound method, 645**
- AudioServicesPlaySystemSound method, 655-656**
- authentication challenge, handling, 721-725**
- autocapitalizationType property, 492-493**
- autocorrectionType property, 493**
- auto-focus-camera key, 663**
- automatic reference counting. See ARC (automatic reference counting)**
- automatically parsing markup text into attributed strings, 532-535**
- automating camera shots, 358**
- autorelease pools, 27, 94-95**
- autoreleased objects**
  - creating, 68-69
  - object lifetime, 69
  - retaining, 69-70
- autoreleased qualifiers, 91-92**
- autosizing, 235-237**
  - evaluating options, 238-239
  - example, 237-239
- available disk space, checking, 692-693**
- AVAudioPlayer, 655**

**AVCaptureVideoPreviewLayer class, 364**

**AVFoundation camera, accessing, 359-368**

- building camera helper, 367-368
- camera previews, 364
- establishing camera session, 361-363
- EXIF, 365
- image geometry, 365-367
- laying out camera previews, 364-365
- querying and retrieving cameras, 360-361
- requiring cameras, 360
- switching cameras, 363

## B

---

**background colors, changing, 561-562**

**backgroundColor property, 321**

**backtraces, 157-158**

**badging applications, 654**

**bar buttons, 249-250**

**bars, 195-196**

**Base SDK targets, 173**

**battery state (iPhone), monitoring, 666-667**

**batteryMonitoringEnabled property, 666**

**batteryState property, 666**

**becomeFirstResponder method, 682**

**beginAnimations:context method, 322**

**beginGeneratingDeviceOrientation  
Notifications method, 672**

**Bezier paths, drawing Core Text onto,  
543-544**

**big phone text, 551-554**

**BigTextView, 551-554**

**bigTextWithString: method, 553**

**bitmap representation**

- manual image processing with,  
377-383
- applying image processing, 380-382
- drawing into bitmap context,  
378-380
- limitations of, 382-383
- testing touches against bitmaps,  
411-413

**\_block variable, 87**

**blocking checks, 705-707**

**blocks, 45-46, 84-85**

- applications for, 88
- assigning block preferences, 85-87
- building animations with, 323-324
- defining, 85
- local variables, 87
- memory management, 88
- typedef, 87-88

**borderStyle property, 494**

**bouncing views, 329-331**

**bounded movement, 408-409**

**bounded views, randomly moving, 318-319**

**Breakpoint Navigator, 135**

**breakpoints, 153-154, 156-157**

**\_\_bridge\_retained cast, 101**

**bridge\_transfer cast, 100-101**

**bridging, 101**

**BrightnessController class, 271**

**browsing**

- parse tree, 736-738
- SDK APIs, 149-151

**build configurations, adding, 181****building**

- alerts, 633–634
- arrays, 76–118
- dictionaries, 118–119
- parse tree, 734–736
- strings, 110
- URLs, 120–121
- web-based servers, 738–741

**builds**

- cleaning, 178–179
- locating, 178–179

**built-in type detectors, 520–522****buttons**

- adding
  - in Interface Builder, 449–452
  - to keyboards, 498–500
  - to storyboard interfaces, 214
- animation, 456–458
- art, 450–451
- bar buttons, 249–250
- building in Xcode, 453–455
- connecting to actions, 451–452
- multiline button text, 455
- types of, 448–449

**bypassing ARC (automatic reference counting), 97–98****bytesRead property, 716**


---

**C**
**cached object allocations, monitoring, 162–163****calculating relative angle, 671****callbacks, 107–108. *See also* specific methods**

- declaring, 107–108
- implementing, 108

**cameraCaptureMode property, 351****camera-flash key, 663****cameraFlashMode property, 351****cameraOverlayView property, 358****cameras. *See also* images**

- automating shots, 358
- AVFoundation camera, accessing, 359–368
  - building camera helper, 367–368
  - camera previews, 364
  - establishing camera session, 361–363
  - EXIF, 365
  - image geometry, 365–367
  - laying out camera previews, 364–365
  - querying and retrieving cameras, 360–361
  - requiring cameras, 360
  - switching cameras, 363
- camera previews, 364–365
- custom overlays, 358–359
- as flashlights, 353
- model differences, 9

- selecting between, 351
  - sessions, establishing, 361–363
  - writing images to photo album, 349–353
- cameraViewTransform property, 359**
- Canny edge detection, 377**
- Car, 61**
- Carbon, 81**
- CAReplicatorLayer class, 332**
- caret (^), 85**
- case of strings, changing, 114**
- casting between Objective-C and Core Foundation, 99**
- adjusting retain counts, 100
  - basic casting, 99–100
  - conversion issues, 102–103
  - runtime workarounds, 102
- catching acceleration events, 669**
- categories, 104–105**
- Catmull-Rom splines, 426–429**
- cells**
- adding, 576–578
  - building custom cells
    - alternating cell colors, 565–566
    - creating grouped tables, 567
    - in Interface Builder, 563–565
    - removing selection highlights, 566
    - selection traits, 565
  - checked table cells, 571–572
  - colors, alternating, 565–566
  - custom cells, remembering control state, 567–570
  - disclosure accessories, 572–574
  - removing selection highlights, 566
  - reordering, 579–580
  - returning, 583
  - reusing, 560, 570–571
  - swiping, 576
  - tables, 557–558
  - types of, 562–563
- centers of views, 313–314**
- certificates, requesting, 20**
- CF (Core Foundation)**
- casting between Objective-C and Core Foundation, 99
    - adjusting retain counts, 100
    - basic casting, 99–100
    - choosing bridging approach, 101
    - conversion issues, 102–103
    - retains, 101
    - runtime workarounds, 102
  - explained, 81–82
- CFBridgingRelease(), 100**
- CFRelease(), 101**
- CFRunLoopRun(), 643**
- CGImageSource, 365**
- CGPoint, 309, 310**
- CGPointApplyAffineTransform method, 411**
- CGRect, 309–310, 313–314**
- CGRectFromString(), 309, 414**
- CGRectGetMidX, 309**
- CGRectGetMidY, 309**
- CGRectInset, 309**
- CGRectIntersectsRect, 309**
- CGRectMake, 309**
- CGRectOffset, 309**

**CGRectZero, 309**

**CGSize structure, 309, 310**

**changes, detecting, 619**

**changing**

- entry points in storyboard  
interfaces, 215

- view controller class, 217

**checked table cells, creating, 571-572**

**checking**

- arrays, 118
- available disk space, 692-693
- network status, 695-697
- for previous state, 415-416
- site availability, 707-709
- spelling, 522-523

**checkUndoAndUpdateNavBar: method, 419, 420**

**child-view undo support, 418-419**

**choosing**

- bridging approach, 101
- between cameras, 351

**CIImage, 363**

**circles**

- detecting, 429-435
- drawing Core Text into, 539-542

**circular hit tests, 409-411**

**class files, generating, 614-615**

**class methods. *See* methods**

**classes. *See also* specific classes**

- class hierarchy, 63-64
- declaring, 52-54
- extending with categories, 104-105
- Foundation framework. *See* Foundation

- generating class files, 614-615

- naming, 53

- singletons, 103-104

**cleaning builds, 178-179**

**closures. *See* blocks**

**Cocoa Programming for Mac OS X, (Hillegass), 126**

**Cocoa Touch, 5, 82, 196**

**code**

- analyzing, 165
- editing
  - hybrid interfaces, 232-233
  - popover, 218-220

**code signing identities, 172-173**

**collapsing methods, 178**

**collections**

- arrays
  - building, 76-118
  - checking, 118
  - converting into strings, 118
- dictionaries, 119
  - building, 118-119
  - creating, 119
  - listing keys, 119
  - removing objects from, 119
  - replacing objects in, 119
- memory management, 120
- sets, 120
- writing to file, 120

**color control, 469-470**

**color sampling, 384-386**

**commaFormattedStringWithLongLong: method, 692**

- commitAnimations** method, 322
- compiled applications**, signing, 175
- compilers**. *See* ARC (automatic reference counting); MRR (Manual Retain Release)
- compile-time checks**, 175
- compiling Hello World**, 174-175
- componentsJoinedByString**: method, 118
- composition controllers**, presenting, 356
- conditional animation**, 324
- configuring iOS development teams**, 19
- conforming to protocols**, 108-109
- connection:didFailWithError**: method, 715-716
- connection:didFinishLoading**: method, 716
- connection:didReceiveData**: method, 715-716
- connection:didReceiveResponse**: method, 716
- connections**
  - connectivity changes, scanning for, 700-702
  - iPad interfaces, 226-227
  - popovers, 218
- connectionShouldUseCredentialStorage**: method, 723
- consoles**
  - debuggers, 158
  - Organizer, 169-170
- contact add buttons**, 448
- contentViewController**, 199
- contexts**, creating, 615-616
- continuous gestures**, 433
- control state**, remembering, 567-570
- controllerDidChangeContent**: method, 619, 625
- controllers**. *See* view controllers
- controls**, 445
  - buttons
    - adding in Interface Builder, 449-452
    - animation, 456-458
    - art, 450-451
    - building in Xcode, 453-455
    - connecting to actions, 451-452
    - multiline button text, 455
    - types of, 448-449
  - color control, 469-470
  - control events, 446-448
  - customizable paged scroller, 481-486
  - page indicator controls, 478-481
  - sliders, 458-465
    - appearance proxies, 460-465
    - customizing, 459-460
    - efficiency, 460
    - star slider example, 472-475
  - steppers, 471-472
  - subclassing, 467-471
    - creating UIControls, 468
    - custom color control, 469-470
    - dispatching events, 468-469
    - tracking touches, 468
  - switches, 471-472
  - toolbars, 486-489
    - accepting keyboard entry into, 508-511
    - building in code, 487-488

- building in Interface Builder, 486-487
- iOS 5 toolbar tips, 489
- touch wheel, 476-478
- twice-tappable segmented controls, 465-467
- types of, 445-446
- UIView, 193-194

**conversion method, adding to iOS-based temperature converter, 225**

**converter interfaces, building, 227-230**

**converting**

- arrays into strings, 118
- Empty Application template to pre-ARC development standards, 97-98
- interface builder files to objective-C equivalents, 151-153
- RGB to HSB colors, 386
- strings
  - to arrays, 112
  - to/from C strings, 111
- XML into trees, 733
  - browsing parse tree, 736-738
  - building parse tree, 734-736
  - tree nodes, 733

**coordinate systems, 310-311**

**Core Animation emitters, 675-676**

**Core Animation Transitions, 328-329**

**Core Data, 611-612**

- changes, detecting, 619
- class files, generating, 614-615
- contexts, creating, 615-616
- databases, querying, 618-619
- model files, creating and editing, 612-613

- objects
  - adding, 616-618
  - removing, 619-620
- search tables, 623-625
- table data sources, 620-623
- table editing, 625-628
- undo/redo support, 628-632

**Core Foundation (CF)**

- casting between Objective-C and Core Foundation, 99
  - adjusting retain counts, 100
  - basic casting, 99-100
  - choosing bridging approach, 101
  - conversion issues, 102-103
  - retains, 101
  - runtime workarounds, 102
  - transfers, 100-101
- explained, 81-82

**Core Image, 368-370-376**

**Core Location, 10-11**

**Core Motion, 676-680**

- device attitude, 676, 680-681
- drawing onto paths, 542-551
- gravity, 676
- handler blocks, 677-680
- magnetic field, 733
- model differences, 10-11
- rotation rate, 676
- splitting into pages, 536-537
- testing for sensors, 677
- user acceleration, 676

**Core Text**

- building attributed strings, 526-532
- drawing into circles, 539-542

- drawing into PDF, 537-539
- drawing onto paths
  - glyphs, 545-546
  - proportional drawing, 544-545
  - sample code, 546-551
- CoreImage framework, 360**
- CoreMedia framework, 360**
- CoreVideo framework, 360**
- counting sections and rows, 583**
- Cox, Brad J., 51**
- credentials, storing, 722-728**
- cStringUsingEncoding: method, 111**
- C-style object allocations, 80-73**
- CTFramesetterSuggestFrameSizeWithConstraints method, 536**
- currentPage method, 479**
- custom accessory views, dismissing text with, 498-500**
- custom alert overlays, 649-650**
- custom camera overlays, 358-359**
- custom containers and segues, 284-290**
- custom fonts, 525-526**
- custom gesture recognizers, 433-435**
- custom getters and setters, 74-76**
- custom headers/footers, 591-592**
- custom images, 344**
- custom input views**
  - adding to non-text views, 511-513
  - creating, 503-508
  - input clicks, 511-513
  - replacing UITextField keyboards with, 503-508
- custom popover view, 217-218**
- custom sliders, 459-460**
- customizable paged scroller, 481-486**

---

## D

- data, displaying, 192-193**
- data access limits, 13**
- data detectors, 520**
- data sources, 46-47, 106**
  - assigning to tables, 556-557
  - methods, implementing, 559
  - table data sources and Core Data, 620-623
- data uploads, 728**
- databases, querying, 618-619**
- dataDetectorTypes property, 520**
- dates, 115-116**
  - date pickers, creating, 603-605
  - formatting, 606-608
- dealloc methods, 82-84**
- deallocating objects, 82-84**
  - ARC (automatic reference counting), 84
  - with MRR (Manual Retain Release), 82-84
- Debug Navigator, 135**
- debuggers, 153**
  - adding simple debug tracing, 158
  - backtraces, 157-158
  - breakpoints, setting, 153-157
  - consoles, 158
  - debug tracing, 158
  - inspecting labels, 155-156
  - opening, 154-155
- declaring. *See also* defining, 85, 106-107**
  - classes, 52-54
  - methods, 59
  - optional callbacks, 107-108
  - properties, 73-74



**defaultCredentialForProtectionSpace:**  
**method, 725**

**defining. *See also* declaring**

blocks, 85

protocols, 106–107

**delays with audio alerts, 656–658**

**delegate methods, 589**

**delegates, 106**

alert delegates, 634–636

assigning, 558

**delegation, 42–43, 106, 585**

**delete requests, 576**

**deleteBackward method, 509**

**Deployment targets, 173**

**designing rotation, 233**

**detail disclosure buttons, 448**

**detail view controllers, 279**

**detecting**

changes, 619

circles, 429–435

external screens, 687

leaks, 159–162

misspellings, 522–523

shakes

with acceleration, 683–686

with motion events, 681–683

simulator builds, 175

text patterns, 518–522

built-in type detectors, 520–522

creating expressions, 518–519

data detectors, 520

enumerating regular

expressions, 519

**Developer Enterprise Program, 3**

**Developer Profile organizers, 171**

**Developer Program, 2–3**

**developer programs, 1–2**

Developer Program, 2–3

Developer University Program, 3

Enterprise Program, 3

Online Developer Program, 2

provisioning portal, 19

application identifier registration,  
 21–22

certificate requests, 20

device registration, 20–21

provisioning profiles, 22–23

team setup, 19

registering for, 3

**Developer University Program, 3**

**development devices, 5–6**

**device attitude, 680–681**

**device capabilities, 661**

acceleration

locating “up,” 668–672

moving onscreen objects, 672–676

available disk space, checking, 692–693

Core Motion, 676–680

device attitude, 676, 680–681

gravity, 676

handler blocks, 677–680

magnetic field, 733

rotation rate, 676

testing for sensors, 677

user acceleration, 676

device information

accessing basic device information,  
 661–662

- recovering additional device information, 664–665
- external screens, 686–687
  - detecting, 687
  - display links, 688
  - overscanning compensation, 688
  - retrieving screen resolutions, 687
  - Video Out setup, 688
  - VIDEOkit, 688–692
- iPhone battery state, monitoring, 666–667
- proximity sensor, enabling/disabling, 667–668
- required device capabilities, 663
- restrictions, 662–664
- shake detection
  - with AccelerometerHelper, 683–686
  - with motion events, 681–683
- device differences, 8–9**
  - audio, 10
  - camera, 9
  - Core Location, 10–11
  - Core Motion, 10–11
  - OpenGL ES, 11–12
  - processor speeds, 11
  - screen size, 9
  - telephony, 10
  - vibration support and proximity, 11
- device limitations, 12**
  - application limits, 17–18
  - data access limits, 13
  - energy limits, 16–17
  - interaction limits, 16
  - memory limits, 13
  - storage limits, 12
  - user behavior limits, 18
- device logs, 168–169**
- device registration, 20–21**
- device signing identities, 172–173**
- devices, building, 170**
  - compiling and running Hello World, 174–175
  - development provisions, 170–171
  - enabling devices, 171
  - inspecting application identifiers, 172
  - setting Base and Deployment SDK targets, 173
  - setting device and code signing identities, 172–173
  - signing compiled applications, 175
- dictionaries**
  - building, 118–119
  - creating, 119
  - listing keys, 119
  - removing objects from, 119
  - replacing objects in, 119
  - searching, 119
- dictionaryWithContentsOfFile: method, 120**
- dictionaryWithKeysAndValues: method, 63**
- didAddSubview: method, 301**
- didMoveToSuperview: method, 301**
- didMoveToWindow: method, 301**
- direct manipulation interface example, 401–402. *See also* touches**
- disabling**
  - ARC (automatic reference counting)
    - across a target, 96
    - on file-by-file basis, 97

- idle timer, 358
- proximity sensor, 667–668
- disclosure accessories, 572–574**
- discrete gestures, 433**
- discrete valued star slider, 472–475**
- disk space, checking available disk space, 692–693**
- dismiss code, adding to storyboard interfaces, 215**
- dismissing**
  - remove controls, 575–576
  - text with custom accessory views, 498–500
  - UITextField keyboards, 491–495
- dispatching events, 468**
- display links, 688**
- display properties of views, 320–321**
- displaying**
  - alerts, 636
  - data, 192–193
  - remove controls, 575
  - volume alert, 658
- distribution, 178**
  - ad hoc distribution, 182–183
  - adding build configurations, 181
  - ad-hoc packages, building, 183
  - locating and cleaning builds, 178–179
  - over-the-air ad hoc distribution, 184–186
  - schemes and actions, 179–181
- document interaction controller, 200**
- Document-Based applications, 127**
- Documents folder, saving images to, 353–354**
- Done key, dismissing UITextField keyboards with, 494–495**
- dot notation, 71–72**
- doubleSided property, 262**
- DoubleTapSegmentedControl, 465–467**
- DownloadHelper, 715–721**
- DownloadHelperDelegate protocol, 716**
- downloading**
  - asynchronous downloads
    - download helper, 715–721
    - NSURLConnectionDownload Delegate protocol, 713–714
    - SDK (software development kit), 4–5
  - synchronous downloads, 709–713
- download:withTargetPath:withDelegate: method, 717**
- dragging items from scroll view, 440–443**
- DragView, 401–402. *See also* touches**
- drawings. *See also* images**
  - Core Text
    - drawing into circles, 539–542
    - drawing into PDF, 537–539
    - drawing onto paths, 542–551
  - drawing touches onscreen, 424–426
  - smoothing, 426–429
- drawInRect: method, 390**
- drawRect: method, 424–426**
- Dromick, Oliver, 532**
- dumping fonts, 524**
- dumpToPDFFile: method, 537–539**
- dynamic slider thumbs, 460–465**

## E

---

**editing.** *See also* undo support

code

hybrid interfaces, 232–233

popovers, 218–220

model files, 612–613

shake-to-edit support, 423

tables in Core Data, 625–628

view attributes, 211

views, 140–141

**EditingChanged** event, 447

**EditingDidEnd** event, 447

**EditingDidEndOnExist** event, 447

**editor window, Xcode workspace,** 136

**efficiency, adding to sliders,** 460

**e-mail, sending images via,** 354–358

creating message contents, 354–355

presenting composition controller, 356

**emitters,** 675–676

**Empty Application** template

converting to pre-ARC development standards, 97–98

creating projects, 129

**enableInputClicksWhenVisible** method, 511

**enablesReturnKeyAutomatically** property, 526

**enabling**

multitouch, 435–438

proximity sensor, 667–668

**encapsulation,** 71

**encodeWithCoder:** method, 416

**endGeneratingDeviceOrientationNotification** method, 672

**energy limits,** 16–17

**Enterprise Program,** 3

**entry points, changing in storyboard** interfaces, 215

**enumerateKeysAndObjectsUsingBlock:** method, 85

**enumerateKeysAndObjectsWithOptions:usingBlock:** method, 85

**enumerateObjectsAtIndexes:options:usingBlock:** method, 85

**enumeration**

fast enumeration, 63

regular expressions, 519

**establishMotionManager** method, 677

**events**

acceleration events, catching, 669

control events, 446–448

dispatching, 468–471

motion events, detecting shakes with, 681–683

**Exchangeable Image File Format (EXIF),** 365

**executables,** 32–33

**EXIF (Exchangeable Image File Format),** 365

**expectedLength** property, 716

**expressions**

creating, 518–519

enumerating, 519

**extending**

classes with categories, 104–105

UIDevice class for reachability, 697–700

**external screens,** 686–687

detecting, 687

display links, 688

overscanning compensation, 688

retrieving screen resolutions, 687

Video Out setup, 688

VIDEOkit, 688-692

### extracting

extracting view hierarchy trees recipe,  
297-298

face information, 376-377

numbers from strings, 114

## F

---

**face detection, 370-376**

**face information, extracting, 376-377**

**fading views in/out, 324-326**

**fast enumeration, 63**

**fetch requests**

with predicates, 624-625

querying database, 618-619

**fetchObjects property, 619**

**fetchResultsController variable, 618**

**fetchPeople method, 618-619**

**file types, supported image file types, 339**

**files**

class files, generating, 614-615

file management, 121-123

header files, 26

Info.plist file, 33-34

Interface Builder files, 37

IPA archives, 38

.m file extension, 26

main.m file, 26-27

model files, creating and editing,  
612-613

storyboard files, 26

sysctl.h file, 664

writing collections to, 120

XIB files, 26

**filtering text entries, 516-518**

**filters (Core Image), 368-370**

**finding UDIDs (unique device identifiers), 21**

**findPageSplitsForString: method, 537**

**first responders, 423-424**

**flashlights, 353**

**flipping views, 327**

**floating progress monitors, 642**

**folder hierarchy, 32**

**fonts**

custom fonts, 525-526

dumping, 524

**footers, customizing, 591-592**

**format specifiers (strings), 65**

**formatting dates, 606-608**

**forwarding messages, 123-126**

**forwardingTargetForSelector: method, 126**

**forwardInvocation: method, 124**

**Foundation, 72, 109-110**

arrays

building, 76-118

checking, 118

converting into strings, 118

collections, 117

dates, 115-116

dictionaries

building, 118-119

creating, 119

listing keys, 119

removing objects from, 119

replacing objects in, 119

searching, 119

file management, 121–123  
 NSData, 121  
 numbers, 115  
 sets, 120  
 strings, 110
 

- building, 110
- changing case of, 114
- converting to arrays, 112
- converting to/from C strings, 111
- extracting numbers from, 114
- length and indexed characters, 110–111
- mutable strings, 114
- reading/writing, 111
- searching/replacing, 113
- substrings, 112–113
- testing, 114

 timers, 116–117  
 URLs, building, 120–121  
**frame property, 308**  
**frames, 309–318**

- centers of, 313–314
- moving, 311–312
- resizing, 312–313
- utility methods, 314–318

**frameworks for AVFoundation camera usage, 359–360**  
**free(), 55**  
**freeing memory, 55–56**  
**FTPHostDelegate protocol, 43**  
**functions. See specific functions**

---

## G

---

**gamekit key, 663**  
**GameKit peer picker, 201**  
**garbage collection, 18**  
**geometry**

- image geometry, 365–367
- of views, 308–311
  - coordinate systems, 310–311
  - frames, 309–318
  - transforms, 310

**gesture conflicts, resolving, 407**  
**gesture recognizers, 397, 400–401. See also touches**

- custom gesture recognizers, 433–435
- dragging from scroll view, 440–443
- long presses, 401
- movement constraints, 408–409
- multiple gesture recognizers, 404–407
- pans, 401–404
- pinches, 400
- resolving gesture conflicts, 407
- rotations, 400
- simple direct manipulation interface, 401–402
- swipes, 400
- taps, 400
- testing
  - against bitmap, 411–413
  - circular hit tests, 409–411

**gestureRecognizer:shouldRecognizeSimultaneouslyWithGestureRecognizer: method, 404**  
**gestureWasHandled method, 440**

**getIPAddressForHost: method, 703**

#### getters

custom getters, 74–76

defining, 73–74

**glyphs, drawing, 545–546**

**goesForward property, 287**

**gps key, 663**

**gravity, 676**

#### grouped tables

coding, 595

creating, 567

grouped preferences tables, creating,  
595–596

**gyroscope key, 663**

---

## H

**handleWebRequest: method, 742**

**handling authentication challenge, 721–725**

**hardware keyboards, resizing views with,  
500–503**

**hasText method, 509**

#### headers

customizing, 591–592

header files, 26

header titles, creating, 584

#### Hello World

compiling, 174–175

creating projects, 129–132

editing views, 140–141

iPhone storyboard, 138–139

minimalist Hello World, 146–149

reviewing projects, 137–138

running, 174–175

running applications, 141

Xcode workspace, 132–133

controlling, 133–134

editor window, 136

Xcode navigators, 134–135

Xcode utility panes, 135–136

#### hiding

application badges, 654

volume alert, 658

#### hierarchies

extracting view hierarchy trees recipe,  
297–298

of views, 295–297

**Hillegass, Aaron, 126**

**hit tests, circular, 409–411**

**Hosgrove, Alex, 440**

**host information, recovering, 702–705**

**hostname method, 703**

**HSB colors, converting RGB to, 386**

#### hybrid interfaces, 230–231

adding views and populating, 231

creating projects, 231

editing, 232–233

populating, 231

views, tagging, 231–232

---

## I

**IB. See Interface Builder**

**iCloud, image storage in, 338**

**icon images, 34–36**

**idle timer, disabling, 358**

**image geometry, 365-367**

**image pickers, 200**

**image view animations, 331-332**

**imageFromURLString: method, 709**

**imageName: method, 339**

**imageOrientation property, 365**

**images, 337. *See also* cameras**

automating camera shots, 358

Core Image face detection, 370-376

Core Image filters, adding, 368-370

creating new, 391-392

creating thumbnails from, 387-390

custom camera overlays, 358-359

customizing, 344

displaying in scrollable view, 392-395

drawing into PDF files, 390-391

extracting face information, 376-377

icon images, 34-36

launch images, 34-36

manual processing with bitmap  
representations, 377-383

applying image processing, 380-382

drawing into bitmap context,  
378-380

limitations of, 382-383

reading data, 339-342

from photo album, 341-347

in sandbox, 340

UIImage convenience  
methods, 339

from URLs, 340-341, 347-349

sampling a live feed, 384-386

saving to Documents folder, 353-354

sending via e-mail, 354-358

creating message contents, 354-355

presenting composition  
controller, 356

storing, 337-338

supported file types, 339

uploading to TwitPic, 728

view-based screenshots, 390

writing to photo album, 349-353

**imageWithContentsOfFile: method, 339**

**implementing**

methods, 60-61

optional callbacks, 108

tables, 558

cell types, 562-563

changing background color,  
561-562

data source methods, 559

populating tables, 558

responding to user touches,  
560-561

reusing cells, 560

selection color, 561

**incorporating protocols, 107**

**index paths, recovering information  
from, 117**

**indexed substrings, requesting, 112**

**indexes, search-aware indexes, 589-590**

**indexPathForObject: method, 620**

**indicators, alert, 654**

**info dark buttons, 448**

**info light buttons, 448**

**Info.plist file, 33-34, 662-664**

**inheriting methods, 59**



**initWithCoder: method, 416**

**input clicks, adding to custom input views, 511-513**

**inputAccessoryView property, 498**

**inputView property, 503**

**inserting subviews, 300**

**insertText: method, 509**

**inspecting**

application identifiers, 172

labels, 155-156

**instance methods. *See* methods**

**instruments**

detecting leaks, 159-162

explained, 5

monitoring cached object allocations, 162-163

**interaction limits, platform limitations, 16**

**interaction properties of views, 320-321**

**Interface Builder**

building custom cells

alternating cell colors, 565-566

creating grouped tables, 567

removing selection highlights, 566

selection traits, 565

buttons, adding, 449-452

custom cells, building, 563-565

files

converting to objective-C equivalents, 151-153

explained, 37

iOS-based temperature converters, 220-222

adding conversion method, 225

adding labels and views, 222

adding media, 221

adding outlets and action, 223

connecting the iPad interface, 226-227

creating new projects, 220

reorientation, 223

testing interfaces, 223

updating keyboard type, 225-226

Round Rect Buttons, 194

tab bar controllers and, 291-292

tips for, 243-244

toolbars, building, 486-487

**interfaces**

building, 207-208

hybrid interfaces. *See* hybrid interfaces, 230-233

storyboard interfaces. *See* storyboard interfaces, 208-215

testing, 223

**Internet, image storage in, 338**

**iOS-based temperature converters, 220**

adding

conversion method, 225

labels and views, 222

media, 221

outlets and actions, 223

connecting the iPad interface, 226-227

creating new projects, 220

Interface Builder, 221-222

reorientation, 223

testing interfaces, 223

updating keyboard type, 225-226

- IP information, recovering, 702-705
- IPA archives, 38
- iPad interfaces, connecting, 226-227
- iPad support, adding to image picker, 343
- iPhone battery state, monitoring, 666-667
- iPhone Developer University Program
- iPhone storyboard, 138-139
- iPhone Xcode projects
  - application delegates, 28-30
  - application skeleton, 25-26
    - autorelease pools, 27
    - main.m file, 26-27
    - UIApplicationMain function, 27-28
  - requirements, 23-25
  - sample code, 31-32
  - view controllers, 30-31
- isDownloading property, 716
- isFlashAvailableForCameraDevice:
  - method, 351
- isKindOfClass: method, 125
- Issue Navigator, 135
- Isted, Tim, 632
- isValidJSONObject method, 742
- iTunes Connect, 4

---

## J

---

- JSON serialization, 742
- JSONObjectWithData:options:error:
  - method, 742

## K

---

- keyboardAppearance property, 493
- keyboards
  - hardware keyboards, resizing views with, 500-503
  - keyboard type, updating, 225-226
  - UITextField keyboards
    - adjusting views around, 495-498
    - custom buttons, 498-500
    - dismissing, 491-495
    - replacing with custom input views, 503-508
  - view design geometry, 205
- keyboardType property, 493
- Keychain Access, 20
- keychain credentials, storing and retrieving, 723-728
- keys (dictionaries), 119
- key-value coding, 78
- key-value observing (KVO), 45, 79
- keywords. *See* specific keywords
- Kochan, Stephen, 126
- Kosmaczewski, Adrian, 151
- kSCNetworkFlagsReachable flag, 707
- kSCNetworkReachabilityFlagsConnectionOnTraffic flag, 696
- kSCNetworkReachabilityFlagsIsDirect flag, 696
- kSCNetworkReachabilityFlagsIsWWAN flag, 696
- KVO (key-value observing), 45, 79

---

## L

---

**labels**

- adding to iOS-based temperature converter, 222

- inspecting, 155-156

**launch images, 34-36****laying out camera previews, 364-365****leaks, detecting, 159-162****learnWord: method, 522****length of strings, 110-111****libraries, SDK limitations, 18****lifetime of autoreleased objects, 69****live feeds, sampling, 384-386****loading image data, 339-342**

- from photo album, 341-347

- in sandbox, 340

- UIImage convenience methods, 339

- from URLs, 340-341, 347-349

**loadView method, 30****local notifications, 652-653****local variables, 87****localIPAddress method, 703****locating builds, 178-179****locating “up”, 668-672**

- basic orientation, 671-672

- calculating relative angle, 671

- catching acceleration events, 669

- retrieving current accelerometer angle synchronously, 670

**location-services key, 663****Log Navigator, 135****logging information, 64-66****long presses, 401****low-memory conditions, simulating, 163-165**


---

## M

---

**.m file extension, 26****magnetic field, 733****magneticField property, 733****magnetometer key, 663****mail composition, 200****main.m file, 26-27****malloc(), 55****managing**

- files, 121-123

- memory

- with ARC (automatic reference counting), 70-71

- blocks, 88

- with MRR (Manual Retain Release), 67-70

- properties, 72-73

**manifests, over-the-air ad hoc distribution, 184-186****manual image processing with bitmap representation, 377-383**

- applying image processing, 380-382

- drawing into bitmap context, 378-380

- limitations of, 382-383

**Manual Reference Counting (MRC). See MRR (Manual Retain Release)****Manual Retain Release. See MRR (Manual Retain Release)**

**markup text, parsing into attributed strings, 532-535**

**Master-Detail application, 127**

**media, adding to iOS-based temperature converter, 221**

**Media Player controllers, 201**

**memory**

- allocating, 54-55

- low-memory conditions, simulating, 163-165

- memory management, 158-159

  - with ARC (automatic reference counting), 70-71

  - blocks, 88

  - with collections, 120

  - with MRR (Manual Retain Release), 67-70

  - and properties, 72-73

- platform limitations, 13

- releasing, 55-56

**menus. *See also* alerts**

- creating, 646-648

- displaying text in, 648-649

- scrolling menus, 648

- segmented controls recipe, 253-255

- two-item menu recipe, 252-253

**messages, 57**

- message forwarding, 123-126

- tracking, 48

**methods. *See also* specific methods**

- class methods, 62

- collapsing, 178

- compared to functions, 57

- declaring, 59

- implementing, 60-61

- inheriting, 59

- undeclared methods, 57-58

**MFMailComposeViewController, 200, 354, 524**

**microphone key, 663**

**migrating to ARC (automatic reference counting), 95-96**

**minimalist Hello World, 146-149**

**misspellings, detecting, 522-523**

**MKMapView, 193**

**mobile provisions. *See* provisioning profiles**

**modal alerts with run loops, 642-645**

**modal presentation, 251**

**modal view controllers recipe, 258-262**

**ModalAlertDelegate, 643-645**

**model differences. *See* device differences**

**model limitations. *See* device limitations**

**model property, 661**

**models (MVC), 46, 612-613. *See also* Core Data**

**model-view-controller. *See* MVC (model-view-controller)**

**monitoring**

- cached object allocations, 162-163

- connectivity changes, 700-702

- iPhone battery state, 666-667

**motion. *See* acceleration; Core Motion**

**motionBegan:withEvent: method, 682**

**motionCancelled:withEvent: method, 682**

**motionEnded:withEvent: method, 682**

**movement constraints, 408-409**

**moving**

- bounded views, 318–319
- onscreen objects with acceleration, 672–676
- views, 239–243, 311–312

**MPMediaPickerController, 201****MPMoviePlayerController, 201****MPMusicPlayerController, 201****MPVolumeSettingsAlertHide, 658****MPVolumeSettingsAlertIsVisible, 658**

**MRC (Manual Reference Counting). *See* MRR (Manual Retain Release)**

**MRR (Manual Retain Release), 55**

- autoreleased objects
  - creating, 68–69
  - object lifetime, 69
  - retaining, 69–70
- deallocation, 82–84
- memory management, 67–70
- qualifiers, 77
- retain counts, 56, 79–80
- retained properties, 72–73

**multiline button text, 455****multiple gesture recognizers, 404–407****multipleTouchEnabled property, 436****multitouch, 400, 435–438****multiwheel tables, 597–600****mutable arrays, 58****mutable strings, 114–87**

**MVC (model-view-controller), 40. *See also* Core Data**

- blocks, 45–46
- controllers, 42
- data sources, 46–47
- delegation, 42–43

model, 46

notifications, 44–45

target-actions, 43–44

UIApplication object, 47–48

view classes, 40–41

---

## N

**name dictionary, 305–308****name property, 662****naming**

- classes, 53
- controllers, 213–214
- scenes, 211
- views, 303–308
  - associated objects, 304–305
  - name dictionary, 305–308

**navigation bars, 195, 203–205**

- adding to storyboard interfaces, 213
- tinting, 214
- undo support, 419–420

**navigation buttons, 211–213****navigation controllers, 247–251**

- adding, 216–217
- modal presentation, 251
- modal view controllers recipe, 258–262
- pushing and popping, 249–250, 255–258
- segmented controls recipe, 253–255
- split view controllers
  - building, 278–282
  - custom containers and segues, 284–290
  - universal apps, building, 282–284

- stack-based design, 249
- two-item menu recipe, 252–253
- UINavigationController class, 250–251
- navigationOrientation property, 263**
- networking, 695**
  - asynchronous downloads
    - download helper, 715–721
    - NSURLConnectionDownload
      - Delegate protocol, 713–714
  - authentication challenge, 721–725
  - blocking checks, 705–707
  - connectivity changes, scanning for, 700–702
  - credentials
    - storing, 722–725
    - storing and retrieving keychain credentials, 723–728
  - host information, recovering, 702–705
  - IP information, recovering, 702–705
  - JSON serialization, 742
  - network connections, testing, 696–697
  - network status, checking, 695–697
  - site availability, checking, 707–709
  - synchronous downloads, 709–713
  - Twitter, 732–733
  - UIDevice, extending for reachability, 697–700
  - uploading data, 728
  - web-based servers, building, 738–741
  - XML, converting into trees, 733
    - browsing parse tree, 736–738
    - building parse tree, 734–736
    - tree nodes, 733
- no-button alerts, 639–642**
- nodes (tree), 733**
- notifications. *See also* alerts**
  - explained, 44–45
  - local notifications, 652–653
- NSArray, 58–59**
- NSAttributedString, 526–532**
- NSAutoreleasePool, 26**
- NSBundle, 32**
- NSComparisonResult, 114**
- NSCompoundPredicate, 623–625**
- NSData, 121**
- NSDataDetector, 520**
- NSDate, 115–116**
- NSDateFormatter, 116**
- NSDecimalNumber, 115**
- NSFileManager, 121–123, 692–693**
- NSInteger, 115**
- NSJSONSerialization, 742**
- NSKeyedArchiver, 416–418**
- NSKeyedUnarchiver, 416–418**
- NSLog, 64–66**
- NSMutableArray, 58–59, 118**
- NSMutableAttributedString, 526–532**
- NSMutableString, 114**
- NSMutableURLRequest, 709**
- NSNotificationCenter, 44**
- NSNumber classes, 115**
- NSObject, 54**
- NSOperationQueue**
  - for blocking checks, 705–707
  - uploading data with, 728
- NSPredicates, 623–625**
- nsprintf function, 66**
- NSRegularExpression, 519**
- NSString, 65**

**NSStringFrom**, 66  
**NSStringFromCGAffineTransform** method, 66  
**NSStringFromCGRect** function, 309  
**NSStringFromCGRect** method, 66, 414  
**NSTimeInterval**, 116  
**NSUInteger**, 115  
**NSURLConnection**, 709, 721  
**NSURLConnectionDownloadDelegate**  
   protocol, 713-714  
**NSURLCredential**, 722-725  
**NSURLCredentialPersistenceForSession**, 722  
**NSURLCredentialPersistenceNone**, 722  
**NSURLCredentialPersistencePermanent**, 722  
**NSURLProtectionSpace**, 722-725  
**NSURLResponse**, 710  
**numberOfPages** property, 479  
**numberOfSectionsInTableView**, 559  
**numbers**  
   extracting from strings, 114  
   NSNumber, 115

---

## O

---

**objc\_retainedObject()**, 102  
**objc\_unretainedObject()**, 102  
**objc\_unretainedPointer()**, 102  
**objectAtIndexPath:** method, 620  
**objectForKey:** method, 119  
**Objective-C 2.0**, 19, 24, 51-52, 87,  
   ARC (automatic reference counting).  
     *See* ARC (automatic reference  
     counting)  
   arrays, 58-59  
   blocks, 84-85  
   applications for, 88  
   assigning block preferences, 85-87  
   defining, 85  
   local variables, 87  
   typedef, 87-88  
   categories, 104-105  
   classes  
     class hierarchy, 63  
     declaring, 52-54  
     extending with categories, 104-105  
     naming, 53  
     singletons, 103-104  
   converting, 151-153  
   fast enumeration, 63  
   Foundation framework. *See*  
     Foundation  
   logging information, 64-66  
   memory management  
     with ARC (automatic reference  
     counting), 70-71  
     memory allocation, 54-55  
     memory deallocation, 55-56  
     with MRR (Manual Retain  
     Release), 67-70  
   message forwarding, 123-126  
   messages, 57  
   methods  
     class methods, 62  
     compared to functions, 57  
     declaring, 59  
     implementing, 60-61  
     inheriting, 59  
     undeclared methods, 57-58

- MRR (Manual Retain Release). *See* MRR (Manual Retain Release)
- objects
- autoreleased objects, 68-70
  - creating, 54, 67-68, 80-82
  - C-style object allocations, 80-73
  - deallocating, 82-84
  - pointing to, 58-59
- properties, 71
- custom getters and setters, 74-76
  - declaring, 73-74
  - dot notation, 71-72
  - encapsulation, 71
  - key-value coding, 78
  - KVO (key-value observing), 79
  - memory management, 72-73
  - qualifiers, 76-78
  - strong properties, 89-90
  - weak properties, 89-90
- protocols, 106
- callbacks, 107-108
  - conforming to, 108-109
  - defining, 106-107
  - incorporating, 107
- Objective-C Programming: The Big Nerd Ranch Guide (Hillegass), 126**
- Objective-C++ hybrid projects, 24**
- object-oriented programming, 39**
- objects. *See also* specific objects**
- adding with Core Data, 616-618
  - autoreleased objects
    - creating, 68-69
    - object lifetime, 69
    - retaining, 69-70
  - creating, 54, 67-68, 80-82
  - C-style object allocations, 80-73
  - deallocating, 82-84
    - ARC (automatic reference counting), 84
    - with MRR (Manual Retain Release), 82-84
  - onscreen objects, moving with acceleration, 672-676
  - pointing to, 58-59
  - removing, 619-620
- Online Developer Program, 2**
- onscreen objects, moving with acceleration, 672-676**
- OOP (object-oriented programming), 39**
- OpenGL ES, 11-12**
- OpenGL Game, 128**
- opengles-1 key, 663**
- opengles-2 key, 663**
- opening debuggers, 154-155**
- operation queues**
- for blocking checks, 705-707
  - uploading data with, 728
- optional callbacks**
- declaring, 107-108
  - implementing, 108
- @optional keyword, 107**
- Organizer, 165**
- applications, 169
  - consoles, 169-170
  - device logs, 168-169
  - devices, 165-166
  - provisioning profiles, 168-133
  - screenshots, 170
  - summary, 167



**organizing views, 209-210**

**orientation, image geometry, 365-367**

**orientation property, 671-672**

**outlets**

adding, 223-225

creating, 223-224

**outputStream variable, 715**

**overscanning compensation, 688**

**over-the-air ad hoc distribution, 184-186**

## P

---

**page indicator controls, 478-481**

**page view controllers, 199, 262-269**

properties, 262-263

sliders, adding to, 269-271

wrapping the implementation,  
263-264

**Page-Based Application, 128**

**paged scroller control, 481-486**

**paged scrolling for images, 395**

**pages, splitting Core Text into, 536-537**

**pagingEnabled property, 395**

**pans, 401-404**

**parallel gestures, recognizing, 404-407**

**parse tree**

browsing, 736-738

building, 734-736

**parser:didEndElement: method, 734**

**parser:foundCharacters: method, 734**

**parseXMLFile: method, 734**

**parsing markup text into attributed strings,  
532-535**

**pasteboard, image storage in, 338**

**paths, drawing Core Text onto, 542-551**

Bezier paths, 543-544

glyphs, 545-546

proportional drawing, 544-545

sample code, 546-551

**patterns (text)**

creating, 518-519

detecting, 518-522

built-in type detectors, 520-522

data detectors, 520

enumerating regular  
expressions, 519

**PDF files**

drawing Core Text into, 537-539

drawing images into, 390-391

**peer-peer key, 663**

**performArchive method, 514**

**performFetch method, 623**

**performSelector: method, 124**

**persistence**

adding to direct manipulation  
interfaces, 413

archiving, 416-418

recovering state, 415-416

storing state, 413-415

persistent credentials, 722-728

text editors, 513-516

**phases (touches), 398**

**photo album**

image storage in, 337

reading images from, 341-347

customizing images, 344

iPad support, 343

populating photo collection, 344

- recovering image edit information, 344-347
- writing images to, 349-353
- photos. *See* images**
- pickers, 195**
  - date pickers, creating, 603-605
  - view design geometry, 205
  - view-based pickers, 601-603
- pictures. *See* images**
- pinches, 400**
- placeholder property, 493**
- platform differences, 8-9**
  - audio, 10
  - camera, 9
  - Core Location, 10-11
  - Core Motion, 10-11
  - OpenGL ES, 11-12
  - processor speeds, 11
  - screen size, 9
  - telephony, 10
  - vibration support and proximity, 11
- platform limitations, 12**
  - application limits, 17-18
  - data access limits, 13
  - energy limits, 16-17
  - interaction limits, 16
  - memory limits, 13
  - storage limits, 12
  - user behavior limits, 18
- plus sign (+), 62**
- pointing to objects, 58-59**
- popover, 216-217, 650-652**
  - code, editing, 218-220
  - connections, 218
  - customizing, 217-218
  - navigation controllers, adding, 216-217
  - popover controllers, 199
  - view controller class, changing, 217
- popping view controllers, 249-250, 255-258**
- populating**
  - hybrid interfaces, 231
  - tables, 558
- populating photo collection, in photo album, 344**
- pragma marks, 177-178**
- predicates, 623-625**
- prepareWithInvocationTarget: method, 420**
- presenting composition controllers, 356**
- previous state, checking for, 415-416**
- processor speeds, 11**
- profiles, provisioning, 22-23, 168-133**
- Programming in Objective-C 2.0* (Kochan), 126**
- progress bars, 637, 639-640**
- progress indicators, 637-640**
  - floating progress monitors, 642
  - UIActivityIndicatorView, 637-639
  - UIProgressView, 637, 639-640
- Project Navigator, 134**
- projects**
  - creating new, 127-129
  - Hello World, 129-132
  - editing views, 140-141
  - editor (Xcode workspace), 136
  - iPhone storyboard, 138-139
  - reviewing, 137-138

- Xcode navigators, 134-135
- Xcode utility panes, 135-136
- Xcode workspace, 132-134

**properties, 71**

- custom getters and setters, 74-76
- declaring, 73-74
- dot notation, 71-72
- encapsulation, 71
- key-value coding, 78
- KVO (key-value observing), 79
- memory management, 72-73
- of page view controllers, 262-263
- qualifiers, 76-78
  - ARC (automatic reference counting), 77, 89-92
  - atomic qualifiers, 77-78
  - MRR (Manual Retain Release), 77
- strong properties, 89-90
- weak properties, 89-90

**proportional drawing, 544-545****protocols, 106. See also specific protocols**

- callbacks, 107-108
- conforming to, 108-109
- defining, 106-107
- incorporating, 107

**provisioning portal, 19**

- application identifier registration, 21-22
- certificates, requesting, 20
- team setup, 19

**provisioning profiles, 22-23, 168-133****proximity sensor, enabling/disabling, 667-668****proximityState property, 667****pull-to-refresh, adding to tables, 592-595****pushing view controllers, 249-250, 255-258**


---

## Q

---

**qualifiers, 76-78**

- ARC (automatic reference counting), 77, 89
- atomic qualifiers, 77-78
- autoreleased qualifiers, 91-92
- MRR (Manual Retain Release), 77
- strong and weak properties, 89-90
- variable qualifiers, 90-91

**Quartz Core framework, 328, 360****querying**

- cameras, 360-361
- databases, 618-619
- subviews, 298-299

**queues**

- for blocking checks, 705-707
- uploading data with, 728

---

## R

---

**rangeOfString:options:range: method, 523****ranges, generating substrings from, 113****reachability**

- checking site reachability, 707-709
- extending UIDevice class for, 697-700

**reachabilityChanged method, 700****reading**

- image data, 339-342
- from photo album, 341-347

- in sandbox, 340
  - UIImage convenience methods, 339
  - from URLs, 340–341, 347–349
  - strings, 111
- read-only properties, 73**
- read-write properties, 73**
- recovering**
  - additional device information, 664–665
  - host information, 702–705
  - information from index paths, 117
  - IP information, 702–705
  - state, 415–416
- redo support**
  - action names, 422
  - in Core Data, 628–632
  - text editors, 513–516
- reference cycles with ARC (automatic reference counting), 92–94**
- reflections, adding to views, 332–334**
- registering**
  - application identifiers, 21–22
  - for developer programs, 3
  - devices, 20–21
  - for iTunes Connect, 4
  - undos, 420–422
- registerUndoWithTarget:self method, 420**
- regular expressions**
  - creating, 518–519
  - enumerating, 519
- relative angle, calculating, 671**
- releasing memory, 55–56**
- remembering control state, 567–570**
- remembering tab state, 275–278**
- remove controls**
  - dismissing, 575–576
  - displaying, 575
- removeObjects method, 619**
- removeOverlay: method, 649**
- removing**
  - objects from dictionaries, 119
  - objects with Core Data, 619–620
  - selection highlights from cells, 566
  - subviews, 300
- renderInContext: method, 390**
- reordering**
  - cells, 579–580
  - subviews, 300
- reorientation**
  - enabling, 233–235
  - iOS-based temperature converters, 223
- replacing**
  - keyboards, 503–508
  - objects in dictionaries, 119
  - strings, 113
- requesting**
  - certificates, 20
  - fetch requests
    - with predicates, 624–625
    - querying database, 618–619
  - indexed substrings, 112
  - synchronous requests, 709–713
- @required keyword, 107**
- requireGestureRecognizerToFail: method, 407**
- requiring cameras, 360**

**resizing**

- autosizing, 235-237
  - evaluating options, 238-239
  - example, 237-239

**resizing views, 312-313, 500-503****resolving gesture conflicts, 407****responder methods, 399****respondToSelector: method, 108, 125****retain counts**

- adjusting, 100
- MRR (Manual Retain Release), 56, 79-80

**retained properties (MRR), 72-73****retaining**

- autoreleased objects, 69-70
- touch paths, 438-439

**retains, 101****retrieving**

- cameras, 360-361
- current accelerometer angle, 670
- device attitude, 680-681
- keychain credentials, 723-728
- screen resolutions, 687
- views, 301-303

**returning cells, 583****returnKeyType property, 493****reusing cells, 560****reviewing projects, 137-138****RGB colors, converting to HSB, 386****root view controllers, 279****rotation rate, 676****rotationRate property, 676****rotations, 233, 400****Round Rect Buttons, Interface Builder, 194****rounded rectangle buttons, 449****rows, counting, 583****run loops, modal alerts with, 642-645****running applications**

- Hello World, 141, 174-175
- for storyboard interfaces, 216

**runtime compatibility checks, performing, 175-177**


---

## S

**sample code, 31-32****sampling live feeds, 384-386****sandbox**

- image storage in, 337
- reading images from, 340

**sandboxes, 38-39****saving, images to Documents folder, 353-354****say: method, 645-646****scanning for connectivity changes, 700-702****scenes, naming, 211****scheduling local notifications, 652-653****schemes, distribution, 179-181****SCNetworkReachabilityCreateWithAddress(), 707-709****screens**

- external screens, 686-687
  - detecting, 687
  - display links, 688
  - overscanning compensation, 688
  - retrieving screen resolutions, 687
  - Video Out setup, 688
  - VIDEOkit, 688-692
- model differences, 9

**screenshots**

- Organizer, 170
- view-based screenshots, 390

**scroll view**

- displaying images in, 392-395
- dragging items from, 440-443

**scroll wheel control, 476-478****scroller control, 481-486****scrollRangeToVisible: method, 523****SDK (software development kit), 1. *See also* platform differences; platform limitations**

- developer programs, 1-2
  - Developer Program, 2-3
  - Developer University Program, 3
  - Enterprise Program, 3
  - Online Developer Program, 2
- registering for, 3
- downloading, 4-5,
- limitations, 18-19
- provisioning portal, 19
  - application identifier registration, 21-22
  - certificate requests, 20
  - device registration, 20-21
  - provisioning profiles, 22-23
  - team setup, 19

**SDK APIs, browsing, 149-151****search bars, 195****search display controllers, creating, 586-587****Search Navigator, 135****search tables and Core Data, 623-625****searchable data source methods, building, 587-589****search-aware indexes, 589-590****searchBar:textDidChange: method, 623****searching**

- dictionaries, 119
- strings, 113
- tables, 586
  - customizing headers and footers, 591-592
- delegate methods, 589
- search display controllers, 586-587
- searchable data source methods, 587-589
- search-aware indexes, 589-590
  - for text strings, 523

**section indexes, creating, 584-585****sectionForSectionIndexTitle:atIndex: method, 621****sectionIndexTitleForSectionName: method, 621****sectionIndexTitles property, 621****sectionNameKeyPath property, 620****sections, 581**

- building, 582
- counting, 583
- delegation, 585
- header titles, creating, 584
- returning cells, 583
- section indexes, creating, 584-585

**sections property, 620****secureTextEntry property, 493****segmented controls, 253-255, 465-467****segues, custom containers and, 284-290****selecting between cameras, 351****selection color, 561****selection highlights, removing from cells, 566****selection traits, building custom cells, 565**

**sending**

- images via e-mail, 354-358

- tweets, 732-733

- sensors, testing for, 677

- serialization (JSON), 742

- servers, web-based, 738-741

- setAnimationCurve method, 322

- setAnimationDuration method, 322

- setDelegate: method, 42

- setMessageBody: method, 355

- setPosition:fromPosition: method, 420

- setProgress: method, 639

- sets, 120

- setStyle: method, 639

- setSubject: method, 355

**setters**

- custom setters, 74-76

- defining, 73-74

- setThumbImage:forState: method, 459

**shake detection**

- with AccelerometerHelper, 683-686

- with motion events, 681-683

- shake-controlled undo support, 422

- shake-to-edit support, 423

- shake-controlled undo support, 422

- shake-to-edit support, 423

- sharing simulator applications, 146

- Shark, 5

- shouldAutorotateToInterfaceOrientation:  
method, 30

- show method, 636

- showFromBarButtonItem:animated:  
method, 646

- showFromRect:inView:animated:  
method, 646

- showFromTabBar: method, 646

- showFromToolBar: method, 646

- showInView method, 646

- showsCameraControls property, 358

- shutDownMotionManager method, 677

- signing compiled applications, 175

- simulating low-memory conditions, 163-165

- simulator, 142-144

- explained, 4-5

- how it works, 144-146

- limitations of, 6-7

- sharing, 146

- simulator builds, detecting with  
compile-time checks, 175

- Single View Application, 128

- singletons, 103-104

- site availability, checking, 707-709

- sizing. *See* resizing, 235-239

- sliders, 458-465

- adding to page view controllers,  
269-271

- appearance proxies, 460-465

- customizing, 459-460

- efficiency, 460

- star slider example, 472-475

- Smalltalk, 39

- smoothing drawings, 426-429

- sms key, 663

- software development kit. *See* SDK  
(software development kit)

- sorting tables, 580-581

- spell checking, 522-523

- spellCheckingType property, 493

- spineLocation property, 263

**spinning circles (progress indicators), 637-639**

**split view controllers, 198, 248**

building, 278-282

custom containers and segues, 284-290

universal apps, building, 282-284

**splitting Core Text into pages, 536-537**

**splitViewController property, 279**

**springs, 236**

**sqlite3 utility, 617**

**stack, navigation controllers and, 249**

**standard Developer Program, 2-3**

**star slider example, 472-475**

**startupWithDelegate: method, 689**

**state**

recovering, 415-416

storing, 413-415

**status bars, 202-203**

**steppers, 471-472**

**still-camera key, 663**

**storage limits, 12**

**storeCredentials method, 725**

**storing**

credentials, 722-725

images, 337-338

keychain credentials, 723-728

state, 413-415

**storyboard files, 26**

**storyboard interfaces, 208**

apps, running, 216

building, 208-209

buttons, adding, 214

creating new projects, 208

dismiss code, adding, 215

editing, 211

entry points, changing, 215

naming, 213-214

naming scenes, 211

navigation bars, tinting, 214

navigation buttons, adding, 211-213

navigation controllers, adding, 213

organizing, 209-210

update classes, 210-211

**stringByExpandingTildeInPath: method, 123**

**stringFromAddress: method, 702**

**strings, 110**

attributed strings

automatically parsing markup text into, 532-535

building, 526-532

extensions library, 532

building, 110

changing case of, 114

converting arrays into, 118

converting to arrays, 112

converting to pre-ARC development standards, 111

extracting numbers from, 114

format specifiers, 65

length and indexed characters, 110-111

mutable strings, 114-87

NSString, 65

reading/writing, 111

searching/replacing, 113

substrings, 112-113

testing, 114

text strings, searching for, 523



**stringWithCString:encoding: method, 111**

**strong properties, 89-90**

**struts, 236**

**subclassing controls, 467-471**

creating UIControls, 468

custom color control, 469-470

dispatching events, 468-469

tracking touches, 468

**submitting to the App Store, 186-187**

**substrings, 112-113**

**subviews, 295**

adding, 300

querying, 298-299

reordering and removing, 300

**summary, Organizer, 167**

**swapping views, 326-327**

**swipes, 400**

**swiping cells, 576**

**switches, 471-472**

**switching cameras, 363**

**Symbol Navigator, 134**

**synchronous downloads, 709-713**

**sysctl(), 664**

**sysctlbyname(), 664**

**sysctl.h file, 664**

**System Configuration, networking aware  
function, 696**

**system sounds, 655-656**

**systemName property, 661**

**SystemReserved event, 447**

**systemVersion property, 661**

## T

---

**tab bar controllers, 195**

creating, 271-275

Interface Builder and, 291-292

remembering tab state, 275-278

view design geometry, 203-205

**Tabbed Application, 128-129**

**table controllers, 199**

**table view controllers, 279**

**tables, 195, 574**

background colors, changing, 561-562

building custom cells, 566

alternating cell colors, 565-566

selection traits, 565

cell types, 562-563

cells

adding, 576-578

disclosure accessories, 572-574

reordering, 579-580

reusing, 560

creating, 556

assigning data sources, 556-557

assigning delegates, 558

laying out the view, 556

serving cells, 557-558

custom cells

building in Interface Builder,  
563-565

cell reuse, 570-571

checked table cells, 571

remembering control state,  
567-570

- delete requests, 576
- editing in Core Data, 625-628
- grouped tables
  - coding, 595
  - creating, 567
  - creating grouped preferences tables, 595-596
- implementing, 558
  - cell types, 562-563
  - changing background color, 561-562
  - data source methods, 559
  - populating tables, 558
  - responding to user touches, 560-561
  - reusing cells, 560
  - selection color, 561
- multiwheel tables, 597-600
- populating, 558
- pull-to-refresh, 592-595
- remove controls
  - dismissing, 575-576
  - displaying, 575
- search tables, 623-625
- searching, 586
  - customizing headers and footers, 591-592
  - delegate methods, 589
  - search display controllers, 586-587
  - searchable data source methods, 587-589
  - search-aware indexes, 589-590
- sorting algorithmically, 580-581
- supporting undo, 576
- swiping cells, 576
- table data sources, 620-623
- undo/redo support, 628-632
- tableView:canMoveRowAtIndexPath: method, 626**
- tableView:cellForRowAtIndexPath, 559**
- tableView:numberOfRowsInSection:559**
- tagging views, 231-232, 301-303
- takePicture method, 358**
- tappable alert overlays, 650
- taps, 400
- target-actions, 43-44
- teams, iOS development teams, 19
- telephony, 10
- telephony key, 663
- templates
  - Empty Application template, converting to pre-ARC
  - development standards, 97-98
  - moving, 240-243
- testing
  - interfaces, 223
  - network connections, 696-697
  - for sensors, 677
  - strings, 114
  - touches
    - against bitmap, 411-413
    - circular hit tests, 409-411
  - untethered testing, 7-8
- tethering, 7-8**
- text, 491, 494-495**
  - attributed strings
    - automatically parsing markup text into, 532-535
  - building, 526-532
  - extensions library, 532

big phone text, 551-554

#### Core Text

building attributed strings, 526-532

drawing into circles, 539-542

drawing into PDF, 537-539

drawing onto paths, 542-551

splitting into pages, 536-537

#### custom input views

adding to non-text views, 511-513

input clicks, 511-513

replacing UITextField keyboards with, 503-508

dismissing with custom accessory views, 498-500

displaying in action sheets, 648-649

#### fonts

custom fonts, 525-526

dumping, 524

misspellings, detecting, 522-523

multiline button text, 455

persistence, 513-516

resizing views with hardware keyboards, 500-503

text entry filtering, 516-518

text patterns, detecting, 518-522

built-in type detectors, 520-522

creating expressions, 518-519

data detectors, 520

enumerating regular expressions, 519

text strings, searching for, 523

text trait properties, 492-493

text-input-aware views, creating, 508-511

#### UITextField keyboards

adjusting views around, 495-498

custom buttons, 498-500

dismissing, 491-495

replacing with custom input views, 503-508

undo support, 513-516

view design geometry, 207

**textFieldAtIndex: method, 637**

**textField:shouldChangeCharactersInRange: replacementString: method, 516**

**textFieldShouldReturn: method, 492**

**text-input-aware views, creating, 508-511**

**thumbnail images, creating, 387-390**

**timers, 116-117**

**tinting navigation bars, 214**

**titleLabel property, 455**

**Toll Free Bridging, 82**

**toolbars, 486**

accepting keyboard entry into, 508-511

building in code, 487-488

building in Interface Builder, 486-487

iOS 5 toolbar tips, 489

view design geometry, 203-205

**touch paths, retaining, 438-439**

**touch wheel, 476-478**

**TouchCancel event, 447**

**TouchDown event, 446**

**TouchDragEnter event, 446**

**touches, 397**

circles, detecting, 429-435

dragging from scroll view, 440-443

drawing onscreen, 424-426

- explained, 397-398
- gesture recognizers, 400-401
  - custom gesture recognizers, 433-435
  - long presses, 401
  - multiple gesture recognizers, 404-407
  - pans, 401-404
  - pinches, 400
  - resolving gesture conflicts, 407
  - rotations, 400
  - swipes, 400
  - taps, 400
- movement constraints, 408-409
- multitouch, 400, 435-438
- persistence, 413
  - recovering state, 415-416
  - storing state, 413-415
  - through archiving, 416-418
- phases, 398
- responder methods, 399
- simple direct manipulation interface, 401-402
- smoothing drawings, 426-429
- testing
  - against bitmap, 411-413
  - circular hit tests, 409-411
- touch paths, retaining, 438-439
- tracking, 468
- undo support, 418
  - action names, 422
  - child-view undo support, 418-419
  - force first responder, 423-424
  - navigation bars, 419-420
  - registering undos, 420-422
  - shake-controlled undo support, 422
  - shake-to-edit support, 423
  - undo manager, 418
  - views, 399-400
- touchesBegan:withEvent: method, 399-401**
- touchesCancelled:WithEvent: method, 399**
- touchesEnded:withEvent: method, 399**
- touchesMoved:withEvent: method, 399, 424**
- TouchUpInside event, 446**
- TouchUpOutside event, 446**
- ToughDownRepeat event, 447**
- tracking**
  - messages, 48
  - notifications, 45
  - touches, 468
- trackNotifications: method, 45**
- transactions, building, 322-323**
- transfers, bridge\_transfer cast, 100-101**
- transform property, 309**
- transforms, 310, 319-320**
- transitioning between view controllers, 290-291**
- transitions, Core Animation Transitions, 328-329**
- transitionStyle property, 263**
- TreeNode, 736-738**
- trees, converting XML into, 733**
  - browsing parse tree, 736-738
  - building parse tree, 734-736
  - tree nodes, 733
- tweets, 732-733**
- twice-tappable segmented controls, 465-467**
- TwitPic.com service, uploading images to, 728**
- Twitter, 732-733**

two-item menu recipe, 252-253

TWRequest, 733

TWTweetComposeViewController, 733

typedef, 87-88

## U

UDIDs (unique device identifiers), finding, 21

UIAccelerometerDelegate protocol, 668

UINavigationController, 193, 252, 633

creating alerts, 646-648

displaying text in action sheets,  
648-649

scrolling menus, 648

UIActivityIndicatorView, 196, 637-639

UIAlertView, 193, 302, 633. *See also* alerts

UIAlertViewDelegate protocol, 634

UIAlertViewStyleLoginAndPasswordInput,  
637

UIAlertViewStyleTextInput, 636

UIAlertViewStyleSecureTextInput, 636

UIFont, 525-526

UIApplication, 47-48, 358

UIApplicationLaunchOptionsLocalNotification  
Key key, 653

UIApplicationMain function, 27-28

UIBarButtonItem, 252

UIBezierPath, 542-544

UIButton, 194

adding in Interface Builder, 449-452

animation, 456-458

art, 450-451

building in Xcode, 453-455

multiline button text, 455

types of buttons, 448-449

UIButtonTypeCustom, 453-455

UIControl. *See* controls

UIDatePicker, 195, 603-605

UIDevice

connecting to actions, 451-452

device information

accessing basic device information,  
661-662

recovering additional device  
information, 664-665

extending for reachability, 697-700

iPhone battery state, monitoring,  
666-667

batteryMonitoringEnabled  
property, 666

batteryState property, 666

orientation property, 671-672

proximity sensor, enabling/  
disabling, 667

UIDeviceOrientationFaceDown value, 672

UIDeviceOrientationFaceUp value, 672

UIDeviceOrientationIsLandscape(), 672

UIDeviceOrientationIsPortrait(), 672

UIDeviceOrientationLandscapeLeft  
value, 671

UIDeviceOrientationLandscapeRight  
value, 671

UIDeviceOrientationPortrait value, 671

UIDeviceOrientationPortraitUpsideDown  
value, 671

UIDeviceOrientationUnknown value, 671

UIDeviceProximityStateDidChange  
Notification, 667

UIDocumentInteractionController, 200

UIFont, 525-526

- UIGestureRecognizer.** *See* gesture recognizers
- UIGestureRecognizerDelegate,** 440
- UIGraphicsAddPDFContextDestinationAtPoint()** function, 391
- UIGraphicsSetPDFContextDestinationForRect()** function, 391
- UIGraphicsSetPDFContextURLForRect()** function, 391
- UIImage,** 337, 365. *See also* images
  - convenience methods, 339
  - creating new images, 391–392
- UIImageJPEGRepresentation()** function, 353–355
- UIImagePickerController,** 341–347
  - choosing between cameras, 351
  - customizing images, 344
  - iPad support, 343
  - populating photo collection, 344
  - recovering image edit information, 344–347
- UIImagePNGRepresentation()** function, 353
- UIImageView,** 321, 337
  - animations, 331–332
- UIImageViews,** 192
- UIInputViewAudioFeedback** protocol, 511
- UIKeyboardBoundsUserInfoKey** key, 496
- UIKeyboardDidHideNotification,** 496
- UIKeyboardDidShowNotification,** 496
- UIKeyboardWillChangeFrameNotification,** 496
- UIKeyboardWillHideNotification,** 496
- UIKeyboardWillShowNotification,** 496
- UIKeyInput** protocol, 509
- UIKit** class, 290, 353
- UILabel,** 192
- UILayoutContainerView,** 296
- UIModalPresentationFormSheet,** 251
- UIModalPresentationFullScreen,** 251
- UIModalPresentationPageSheet,** 251
- UIModalTransitionStyleCoverVertical,** 251
- UIModalTransitionStyleCrossDissolve,** 251
- UIModalTransitionStyleFlipHorizontal,** 251
- UIModalTransitionStylePartialCurl,** 251
- UINavigationController,** 195, 259
- UINavigationController,** 41, 197–198, 247–252. *See also* navigation controllers
- UINavigationControllerItem,** 250–251
- UIPageControl,** 478–481
- UIPageViewController,** 199, 262–269
- UIPickerView,** 195, 598–600
- UIProgressView,** 196, 637–640
- UIRequiredDeviceCapabilities** key, 662
- UIResponder** methods, 399
- UIReturnKeyDone,** 491
- UIScreen,** 686. *See also* external screens
  - detecting screens, 687
  - display links, 688
  - overscanning compensation, 688
  - Video Out setup, 688
  - view design geometry, 207
- UIScrollView,** 193–194, 392–395, 481–486
- UISearchBar,** 195, 199
- UISegmentedControl,** 194, 205, 253, 465–467
- UISegmentedControlStyleBar,** 254
- UISegmentedControlStyleBordered,** 254
- UISegmentedControlStylePlain,** 254

**UISlider, 194, 458-465**

- appearance proxies, 460-465
- customizing, 459-460
- efficiency, 460
- star slider example, 472-475

**UISplitViewController, 41, 198, 248****UIStepper, 471-472****UISwitch, 194, 471-472****UITabBarController, 41, 195, 198, 271-275****UITableView, 195, 199, 296, 554-556****UITableViewCell, 195****UITableViewCellStyleDefault, 562****UITableViewCellStyleSubtitle, 562****UITableViewCellStyleValue1, 563****UITableViewCellStyleValue2, 563****UITableViewController, 199, 554-556****UITableViewSeparatorView, 296****UITextChecker, 522-523****UITextField, 194. *See also* text**

- keyboards
  - adjusting views around, 495-498
  - custom buttons, 498-500
  - dismissing, 491-495
- properties, 492-493
- text entry filtering, 516-518

**UITextInputTraits protocol, 492-493****UITextView, 192, 522-523****UIToolbar. *See* toolbars****UITouch. *See* touches****UITouchPhaseCancelled, 398****UITouchPhaseEnded, 398****UITouchPhaseMoved, 398****UITouchPhaseStationary, 398****UIView, 40, 191-192, 290, 295, 302**

- animations, 321-324
  - blocks approach, 323-324
  - bouncing views, 329-331
  - building transactions, 322-323
  - conditional animation, 324
  - Core Animation Transitions, 328-329
  - fading in/out, 324-326
  - flipping views, 327
  - swapping views, 326-327
- centers of views, 313-314
- controls, 193-194
- geometry properties, 308-309
- subview management, 300-301
- transforms, 319-320
- utility methods, 314-318

**UIViewAnimationTransition class, 328****UIViewAutoresizingNone, 237****UIViewController, 40-41, 197, 249, 252, 259****UIWebView, 192, 339, 392-395****UIWindow, 191-192, 296****undeclared methods, 57-58****undo manager, 418****undo support**

- in Core Data, 628-632
- table edits, 576
- text editors, 513-516
- for touches, 418
  - action sheets, 422
  - child-view undo support, 418-419
  - force first responder, 423-424
  - navigation bars, 419-420
  - registering undos, 420-422

- shake-controlled undo support, 422
- shake-to-edit support, 423
- undo manager, 418
- unique device identifiers (UDIDs), 21**
- universal split view apps, building, 282-284**
- University Program, 3,**
- unlearnWord: method, 522**
- unsafe\_unretained qualifier, 91**
- untethered testing, 7-8**
- "up," locating, 668-672**
  - basic orientation, 671-672
  - calculating relative angle, 671
  - catching acceleration events, 669
  - retrieving current accelerometer angle synchronously, 670
- update classes, 210-211**
- updateDefaults method, 414**
- updateExternalView: method, 689**
- updateTransformWithOffset: method, 404**
- updating keyboard type, 225-226**
- uploading data, 728**
- urlconnection property, 716**
- URLs**
  - building, 120-121
  - reading images from, 340-341, 347-349
- user acceleration, 676**
- user behavior limits, platform limitations, 18**
- userAcceleration property, 676**
- userInteractionEnabled property, 321**
- userInterfaceldiom property, 662**
- users, alerting. *See* alerts**
- UTF8String method, 60, 111**
- utilities. *See* specific utilities**

- Utility Application, 129**
- utility methods for views, 314-318**
- UTTypeCopyPreferredTagWithClass() function, 355**

---

## V

- ValueChanged event, 447**
- variables**
  - local variables, 87
  - qualifiers, 89-92
- variadic arguments with alert views, 645-646**
- vibration**
  - alert sound, 656
  - audio alerts, 656
  - model differences, 11
- Video Out setup, 688**
- video-camera key, 663**
- VIDEOkit, 688-692**
- view attributes, editing, 211**
- view classes, 40-41**
- view controllers, 30-31, 42, 196-197, 217, 247**
  - adding to storyboard interfaces, 208-209
  - address book controllers, 200
  - document interaction controller, 200
  - GameKit peer picker, 201
  - image pickers, 200
  - mail composition, 200
  - Media Player controllers, 201
  - modal view controllers recipe, 258-262
  - naming, 213-214



- navigation controllers, 247–251
  - modal presentation, 251
  - pushing and popping, 249–250, 255–258
  - segmented controls recipe, 253–255
  - stack-based design, 249
  - two-item menu recipe, 252–253
  - UINavigationController class, 250–251
- page view controllers, 199, 262–269
  - properties, 262–263
  - sliders, adding to, 269–271
  - wrapping the implementation, 263–264
- popover controllers, 199
- split view controllers, 198, 248
  - building, 278–282
  - custom containers and segues, 284–290
  - universal apps, building, 282–284
- tab bar controllers
  - creating, 271–275
  - Interface Builder and, 291–292
  - remembering tab state, 275–278
- table controllers, 199
- transitioning between, 290–291
- tweet view controller, 732–733
- UINavigationController, 197–198
- UITabBarController, 198
- UIViewController, 197
- view design geometry, 201**
  - keyboards, 205
  - navigation bars, 203–205
  - pickers, 205
  - status bars, 202–203
  - tab bars, 203–205
  - text fields, 207
  - toolbars, 203–205
  - UIScreen, 207
- view-based pickers, 601–603**
- view-based screenshots, 390**
- viewDidAppear: method, 31, 418**
- viewDidLoad: method, 30, 666–667**
- views**
  - adding
    - to hybrid interfaces, 231
    - to iOS-based temperature converter, 222
  - adjusting around keyboards, 495–498
  - alert views. *See* alerts
  - animations, 321–324
    - blocks approach, 323–324
    - bouncing views, 329–331
    - building transactions, 322–323
    - conditional animation, 324
    - Core Animation Transitions, 328–329
    - fading in/out, 324–326
    - flipping views, 327
    - image view animations, 331–332
    - swapping views, 326–327
  - bounded views, randomly moving, 318–319
  - callback methods, 301
  - centers of, 313–314
  - custom accessory views, 498–500
  - custom input views
    - adding to non-text views, 511–513
    - creating, 503–508
    - input clicks, 511–513

- display and interaction properties, 320–321
- displaying data, 192–193
- extracting view hierarchy trees recipe, 297–298
- geometry, 308–311
  - coordinate systems, 310–311
  - frames, 309–318
  - transforms, 310
- Hello World, 140–141
- hierarchies of, 295–297
- for making choices, 193
- moving, 239–243, 311–312
- naming, 303–308
  - associated objects, 304–305
  - name dictionary, 305–308
- organizing, 209–210
- popovers, customizing, 217–218
- reflections, 332–334
- resizing, 312–313
- scroll view
  - displaying images in, 392–395
  - dragging items from, 440–443
- subviews
  - adding, 300
  - querying, 298–299
  - reordering and removing, 300
- tables, laying out, 556
- tagging and retrieving, 301–303
- tagging in hybrid interfaces, 231–232
- text views. *See* text
- text-input-aware views, creating, 508–511

- touching view, 399–400
- transforming, 319–320
- utility methods, 314–318
- viewWillAppear:** method, 31
- viewWithTag:** method, 302, 305
- visualizing cell reuse, 570–571
- vnshomedirectory()**, 111
- vnsmakerange()**, 113
- volume alert, 658

---

## W

- weak properties, 89–90
- web-based servers, building, 738–741
- WebHelper**, 738–741
- whatismyipdotcom** method, 703
- wifi key, 663
- willMoveToSuperview:** method, 301
- willMoveToWindow:** method, 301
- willRemoveSubview:** method, 301
- wrapping page view controller implementations, 263–264
- writeToFile:atomically:** method, 120, 353
- writing
  - collections to file, 120
  - images to photo album, 349–353
  - to strings, 111

---

## X-Y-Z

- .xcdatamodel** files, creating and editing, 612–613
- Xcode**
  - application delegates, 28–30
  - application skeleton, 25–26

- autorelease pools, 27

- main.m file, 26–27

- UIApplicationMain function,  
27–28

- buttons, building, 453–455

- explained, 4

- Hello World, 132–133

- controlling, 133–134

- editor window, 136

- Xcode navigators, 134–135

- Xcode utility panes, 135–136

- project requirements, 23–25

- sample code, 31–32

- utility panes, 135–136

- view controllers, 30–31

#### **Xcode 4 Unleashed (Anderson)**

**XIB files, 26, 231**

**XML, converting into trees, 733**

- browsing parse tree, 736–738

- building parse tree, 734–736

- tree nodes, 733

**XMLParser, 734–736**

*This page intentionally left blank*