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

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

Visit us on the Web: informit.com/aw

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 978-0-321-83207-8 (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: 978-0-321-83207-8
ISBN-10: 0-321-83207-8

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

Second printing: January 2012

**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

**3 Building Your First Project 127**
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 UITableView Controller 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 Watmough, Youssef Francis, Bryan Henry, William DeMuro, Jeremy Sinclair, Arshad Tayyeb, Daniel Peebles, ChronicProductions, Greg Hartstein, Emanuele Vulcano, Sean Heber, Josh Bleecher 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 and irc.freenode.net, among many others too numerous to name individually. Their techniques, suggestions, and feedback helped make this book possible. If I have overlooked anyone who helped contribute, please accept my apologies for the oversight.

Special thanks go out to my family and friends, who supported me through month after month of new beta releases and who patiently put up with my unexplained absences and frequent howls of despair. I appreciate you all hanging in there with me. And thanks to my children for their steadfastness, even as they learned that a hunched back and the sound of clicking keys is a pale substitute for a proper mother. My kids provided invaluable assistance over the 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 TUAW.com, and has blogged in the past at O'Reilly's Mac DevCenter, Lifehacker, and Ars Technica. In addition to being the author of dozens of iOS-native applications, Erica holds a Ph.D. in Computer Science from Georgia Tech's Graphics, Visualization and Usability Center. A geek, a programmer, and an author, she's never met a gadget she didn't love. When not writing, she and her geek husband parent three geeks-in-training, who regard their parents with restrained bemusement, when they're not busy rewiring the house or plotting global dominance.

# 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). The *Expanded Electronic Edition* will include the equivalent of what would amount to several hundred pages of printed material. For customers who have already purchased the ebook form of the print book and only want the additional chapters, we have created *The iOS 5 Developer's Cookbook: The Additional Recipes*. As with the *Extended Electronic Edition*, *The Additional Recipes* will be available in ePub, Kindle, and PDF.

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 anything else you'd like to contribute to a future edition, please contact me at erica@ericasadun.com. Let me thank you all in advance. I appreciate all feedback that helps make this a better, stronger book.

—Erica Sadun, November 2011 (updated December 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). 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,[1] or if you prefer books, you can learn from the best. Aaron Hillegass, founder of the Big Nerd Ranch in Atlanta,[2] is the coauthor of *iOS Programming: The Big Nerd Ranch Guide, Second Edition* and author of *Cocoa Programming for Mac OS X, 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.

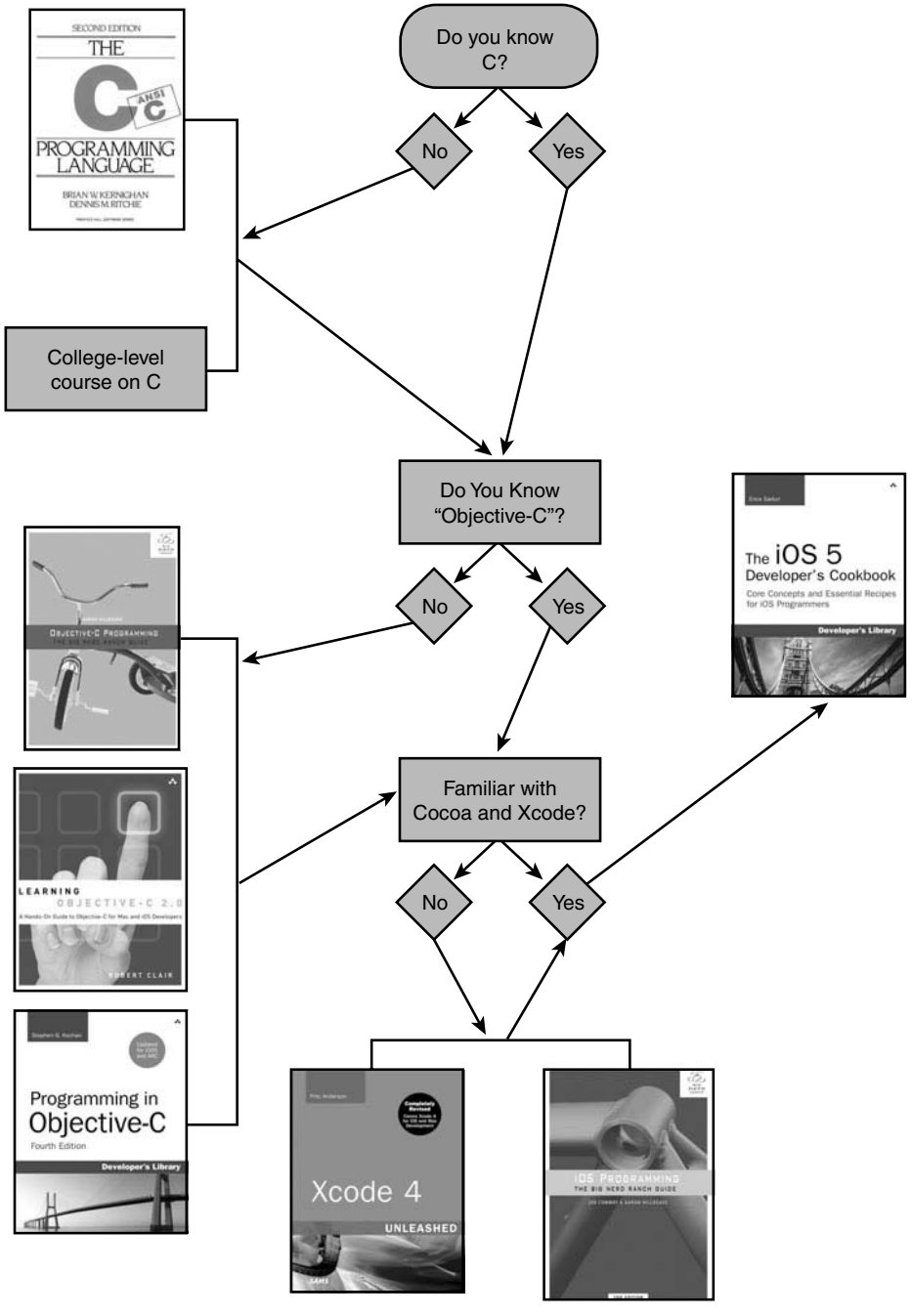


Figure P-1 What it takes to be an iOS programmer.

> **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, or stop by 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

1 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).

2 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: trina.macdonald@pearson.com
Mail: Trina MacDonald
Senior Acquisitions Editor
Addison-Wesley/Pearson Education, Inc.
1249 Eighth Street
Berkeley, CA 94710

# 3

# Building Your First Project

Xcode helps you craft iOS SDK applications with an exciting suite of code editors and testing tools. This chapter introduces you to the basics of using Xcode to build your projects. You see how to build a simple Hello World project, compile and test it in the simulator, and then learn how to compile for and deploy to the device. You also discover some basic debugging tools and walk through their use as well as pick up some tips about handy compiler directives. This chapter also looks at how to submit to the App Store and perform ad hoc distribution for testing. By the time you finish this chapter, you'll have followed the application-creation process from start to finish and been shown valuable tricks along the way.

## Creating New Projects

If diving into SDK programming without a lifeline seems daunting, be reassured. Xcode simplifies the process of getting started. It provides preconfigured projects that you can easily adapt while exploring the SDK. These projects provide fully working skeletons. All you need to do is add a little custom functionality to make that app your own.

To get started, launch Xcode 4 and choose File > New Project (Command-Shift-N). The New Project template window (see Figure 3-1) opens, allowing you to select one of these application styles to get started. These project styles are chosen to match the most common development patterns for iOS. Your choices are as follows:

- **Document-Based Application**—Intended for use with iCloud, the document template creates a starting point for building applications around ubiquitous elements.
- **Master-Detail Application**—Usually based around lists and tables, master-detail applications let users drill their way through a hierarchical interface. These apps offer a tree-structured collection of interface choices, each choice sliding to a new screen or presenting, in the case of the iPad, in a separate detail view. The bars at the top of the navigation screens include an integrated Back button, letting users return to previous screens with a single tap. On the iPad, split views automatically

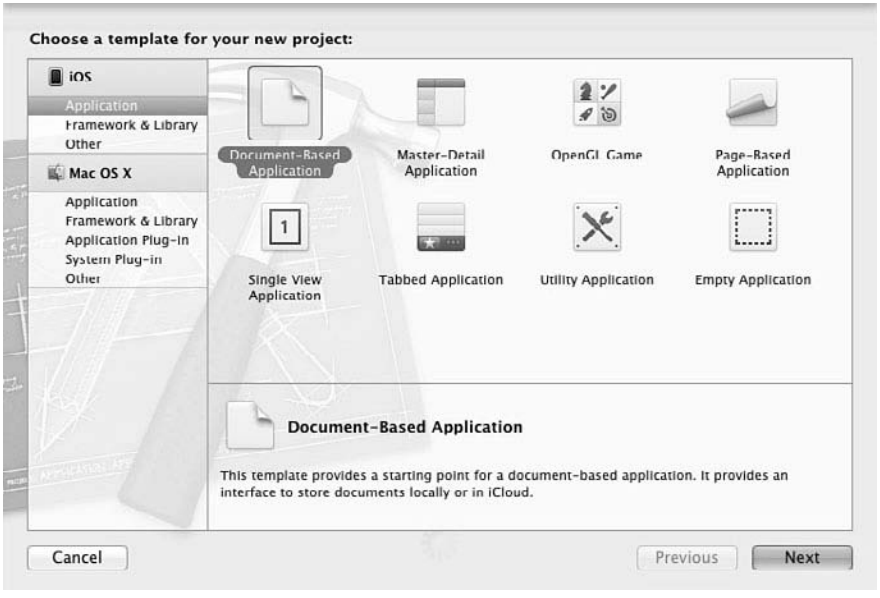


Figure 3-1 The Xcode New Project template selection window. The Xcode interface will change as Apple continues evolving its design tools.

accommodate themselves to match device orientations. In landscape mode, both views are shown at once; in portrait, the detail view is shown and the selection choices appear via a navigation-bar-based popover.

- **OpenGL Game**—When programming with OpenGL ES, all you need is a view to draw into and a timer that offers an animation heartbeat. The OpenGL Game template provides these elements, letting you build your OpenGL ES graphics on top.
- **Page-Based Application**—Create a book-style application by choosing this page view controller-based template. New to iOS 5, the page view controller allows users to navigate through an electronic "book" using familiar touch-based gestures. The data source client feeds view controllers to the application, and a delegate allows the app to respond to gesture-based navigation and orientation updates.
- **Single View Application**—This simple template provides a basic starting point for a primary view controller, offering storyboards for both iPhone and iPad distribution. Choose this style when you're looking for an app that centers on a primary view rather than one that needs a specialized container style, such as a navigation controller, tab bar controller, split view controller, page view controller, and so on.
- **Tabbed Application**—Apple's iPod and YouTube applications offer typical examples of tab bar applications. In these applications, users can choose from a series of

parallel screens by tapping buttons in a bar at the bottom of the application. For example, the YouTube application lets you choose from Featured Videos, Most Viewed, Bookmarks, and the search pane, each of which is accessed through a tab button. The Tabbed Application template provides a skeleton that you can grow to add panes and their contents. Although you can select either the iPhone or iPad as your target product in Xcode, Apple encourages you to avoid creating tab-bar-style applications on the iPad. The iPad's generous screen space provides enough room that you do not need to fold your main interfaces using the conventions of tab bar and navigation applications.

- **Utility Application**—Meant to be the simplest style of application, the Utility Application template creates a two-sided single-view presentation like the ones you see in the Stocks and Weather application. The template provides a main view and a flip view, which you can easily customize. This application template uses a simple flip-to-the-other-side presentation on the iPhone. On the iPad, it offers a popover linked to an information bar button item. Utility applications work best when they are highly focused. The flip-view or popover generally serves for in-app settings rather than an extension of the primary interface feature set.
- **Empty Application**—The window-based application essentially offers the same template as the Single View one but without the storyboards. You get an application delegate and a customizable window, and that's about it. One advantage of choosing this template is that it's relatively easy to customize if you prefer to build your iPhone applications completely from scratch.

> **Note**
>
> Apple offers sample code and tutorials at the iOS Reference Library. Visit the library online at http://developer.apple.com/library/ios/navigation/index.html; you must use your developer credentials to access its contents. In addition to sample code, you'll find release notes, technical notes, Getting Started guides, Coding How-To's, and more. Many of these resources are also available directly in Xcode through its built-in documentation browser.

# Building Hello World the Template Way

Xcode's preconfigured templates offer the easiest path to creating a Hello World–style sample application. In the following steps, you create a new project, edit it to say "Hello World," and run it on the iOS simulator. As you build your first Xcode project, you'll discover some of the key development pathways.

## Create a New Project

With the iOS SDK installed, launch Xcode. Close the Xcode Welcome page; it's the window that says "Welcome to Xcode" and offers options such as Create a New Xcode Project. This window continues to appear until you uncheck "Show this window when

Xcode launches" before closing it. Thereafter, you can access the page via Window > Welcome to Xcode (Command-Shift-1).

To create a new project, choose File > New > New Project (Command-Shift-N). This opens the template selection window shown in Figure 3-1. By default, the template selection window is embedded in a large new window, which is called a "workspace." Workspaces embed all of Xcode's editing and inspector features into a single window.

The left column in Figure 3-1 includes three iPhone project categories. These are Application (that is, the screen you see in Figure 3-1) Framework & Library (for creating static Cocoa Touch library modules), and Other (which initially contains a single, empty project style).

Choose Application > Single View Application and then click Next. Xcode opens a "Choose options for your new project:" screen, as shown in Figure 3-2. Enter **Hello World** for the project name and set the company identifier (for example, com.sadun). Company identifiers typically use reverse domain naming. The new application identifier appears below this in light grey text. In my case, this is com.sadun.Hello-World. Enter an optional class prefix. Xcode uses this to prepend template classes. If you enter *ES*, for example, Xcode creates ESAppDelegate.h and .m files. Choose Universal from the Device Family pop-up. Leave Use Storyboard checked. Leave Include Unit Tests unchecked. Click Next.

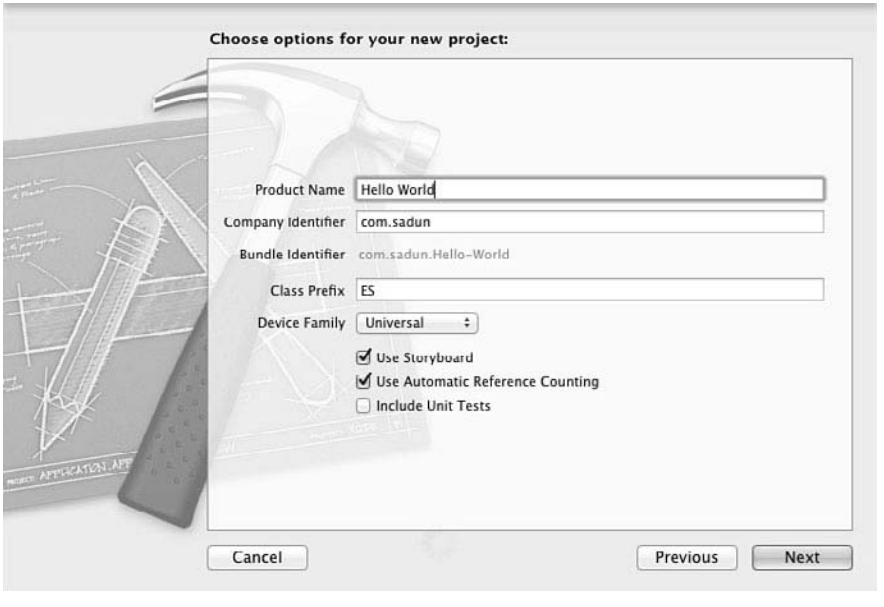


Figure 3-2 Once you set your company identifier, Xcode applies your settings between successive runs. This helps you avoid what was one of the most irritating features of previous Xcode releases—having to remember to edit your settings from "com.yourcompany" for each project.

Choose where to save the new project (such as the desktop) and click Save. A new Hello World Xcode workspace opens (see Figure 3-3). This project contains all the files needed to design a new, universal application centered on a single primary window. The files are grouped together into folders and listed in the left-hand pane, which is called the Navigator. Your new project is selected by default. Its project and target settings appear in the right-hand editor pane. This project settings editor is functionally similar to the target info pane found in earlier Xcode releases but with many improved features. The first editor pane you see allows you to choose orientation support and set application images.

Figure 3-3 This brand-new Hello World project was created by choosing one of the available templates. On older projects, you may see a "modernize" button on this screen as well.

Feel free to explore the files in your new project. Open groups (they look like folders but are just organizing aids for your project) to expose their contents, and then select any file to view it in the editor. To expose all the folders at once, Option-click the arrow next to the main project listing (that is, "Hello World" in this case) when no subfiles are showing. If files are showing, Option-click twice. The first click hides all the subfiles; the second click reveals them.

Xcode 4's editor supports code, property lists, images (display only), and Interface Builder (IB) files. If you click, for example, on MainWindow_iPad.xib, you will open an IB editor pane. That's a big change from Xcode 3, where Interface Builder was a separate program.

As Figure 3-3 shows, Xcode's GUI has undergone massive changes from its 3.*x* incarnations. Whether you are new to iOS development or transitioning from an earlier SDK,

it's well worth exploring the Xcode workspace and its components. From a more integrated single-window workspace to the migration of Interface Builder into the main Xcode application, a lot of new features are waiting for you in Xcode 4.

## Introducing the Xcode Workspace

Although it is not clear from Figure 3-3, the standard Xcode Project window is composed of three primary sections. You can best see this by clicking the Utility button at the top-right of the window. This is the second button from the right, and looks like a white rectangle with a smaller darker rectangle to its right. It is part of a set of seven buttons grouped as three buttons (labeled "Editor"), then three buttons (labeled "View"), then one button (labeled "Organizer"). Go ahead and click it to reveal the Utility section to the right of the editor. You may want to resize the workspace after exposing the Utility section.

Figure 3-4 shows the expanded workspace. It consists of three areas and a toolbar. From left to right, those areas include the Navigator pane, which allows you to browse project components, the Editor pane in the center, which provides editors and viewers for your files, and the Utility area to the right, which offers inspectors and library access.

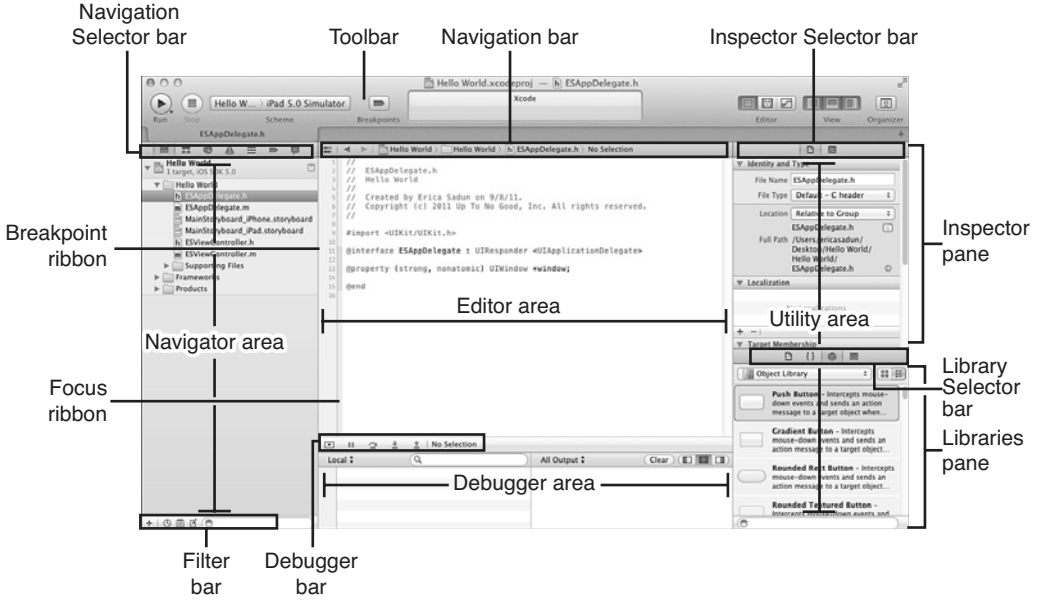


Figure 3-4 The Xcode workspace consists of several areas, as shown here.

An optional debugging pane appears at the bottom of the editor when a program runs. You can show and hide it by clicking the debugger disclosure button at the bottom-left of the editor or by clicking the center of the three view buttons at the top-right corner of the Xcode window. The disclosure button is a small rectangle with an upward ("show") or downward ("hide") triangle. The view button is a rectangle with a darker embedded rectangle at its bottom.

A small central activity view appears in the center of the toolbar at the top of the window. It has a light blue color and can be found just under the window's title. This view shows you the current state of your project and any ongoing activities, such as building code or an ongoing search. It looks a lot like the activity view you see in iTunes. Application run, stop, and breakpoint controls can be found just to the left of the activity view.

### Controlling the Workspace

The seven buttons at the right of the workspace toolbar allow you to select the ways you want to view your workspace. Starting from the left, these buttons are as follows.

- **Editor buttons**—These three editor buttons control how code is displayed in the central editor window. Your choices are standard, assistant, and version:
  - The standard editor (Command-Return) displays a single source code pane.
  - The assistant (Command-Option-Return) allows you to split your viewer in two, providing context-enhancing files in the secondary pane. The assistant can automatically find files and display file counterparts (for example, showing MyClass.h when viewing MyClass.m), superclass or subclass files, and so forth. The assistant is an absolutely brilliant feature of Xcode 4. Customize your assistant layout using the View > Assistant Layout submenu.
  - The version editor (Command-Shift-Option-Return) allows you to compare two versions of a single file against each other, so you can spot differences and see what has changed over time between separate commits.

- **View buttons**—These three buttons allow you to hide and show the Navigator (left), Debug (center), and Utility (right) panes that appear in Figure 3-4. When all three are hidden, only the central editor pane appears. Doing so provides what appears to be a simple editor window but with quick access back to your project controls. You can also double-click files in the Navigator to open them up in new windows, with the other panes hidden.
- **Organizer button**—Looking like a small window with smaller embedded rectangles, the Organizer button provides one-click access to the Xcode Organizer window. The Organizer, which is discussed later in this chapter, provides a single source for your mobile device controls, documentation, project organization, and more. You can also access the organizer by choosing Window > Organizer, or by pressing Command-Shift-2 on the keyboard.

> **Note**
>
> Double-click any file listed in the Navigator to open it in a separate editor window. By default, this hides the top toolbar. Restore it by selecting View > Show Toolbar. There is no shortcut for this menu item by default, although you can add one using a third-party utility such as Quickeys or via the preferences' key bindings pane.

### Xcode Navigators

The left pane of the Xcode window doesn't just list projects and folders. It handles a lot more as well. It's called the "navigator area" and it lets you navigate information about projects in your workspace. Xcode 4 offers seven specialized navigators. Each navigator organizes information so you can browse through it. These navigators are accessed through the tab buttons at the top of the Navigator pane. Xcode 4 navigators include the following items:

- The Project Navigator (Command-1) lists the groups and files that comprise your project. Selecting a file opens that file in an editor in the central pane. The kind of editor presented depends on the selected file. A code file (.h, .m, and so on) opens a code editor. Property list files, such as your Info.plist, open a property list editor. Interface files (.storyboard and .xib files) open in an Interface Builder (IB) editor, directly in the Xcode project window. This differs from earlier versions of Xcode, where IB was a standalone program, outside of Xcode.
- The Symbol Navigator (Command-2) enumerates the classes, functions, and other symbols used within your project. Selecting a symbol opens the declaring header file, so you can instantly look up how the element is defined. The Utility > Symbols (Quick Help) inspector pane, found in the right-hand area on the other side of the editor, works synchronously with the Symbol Navigator by providing instant contextual documentation for any selected symbol, whether you select one from the Symbol Navigator or from the source editor.

> **Note**
>
> When the Utility > Symbols (Quick Help) inspector is hidden, you can Option-click any symbol in the navigator or any text within a code editor to pop up an Xcode 4 Quick Help window with the same contextual documentation found in the Utility > Symbols pane. Use Option-double-click to open the item in the Organizer's documentation window. Two buttons appear at the top-right of the Quick Help pop-up. The small book button leads to any existing documentation. The file icon with the "h" on it links to the declaring header file.

- The Search Navigator (Command-3) provides an easy way to find text within your project. You can search both in your workspace (including any projects that have been added to the workspace) and in associated frameworks for instances of that text. Choose the search method you want to use from the text field pop-down as

you type into the text field. A simple pop-up to the left of the search field lets you use find-and-replace features as well. You are not limited to searching via this Search Navigator. You can also search via Command-F (Edit > Find > ...) in any text-based editor.

> **Note**
>
> The small magnifying glass icon at the left of the search field in the Search Navigator plays an important role in controlling what documentation you search through. Click the magnifying glass and select Find Options to reveal options for matching your search phrase against source code.

- The Issue Navigator (Command-4) provides a list of warnings and errors found during your build requests. Use this pane to select each issue and highlight its problem in the central code editor.
- The Debug Navigator (Command-5) offers a way to examine threads and stacks during execution. When you select an item, the editor will instantly jump to either the source code or disassembly window, where you can see the current point of execution.
- The Breakpoint Navigator (Command-6) provides a global list of debugging breakpoints within your project. Click any breakpoint to view it in-file in the editor.
- The Log Navigator (Command-7) allows you to view your build results as a history. Click any past build to see the build log in the central editor pane.

### Xcode Utility Panes

Context-specific helper panes appear in the right-hand utility pane. Like the left-hand navigator, the utility area can be shown or hidden as needed. This area consists of two sections: at the top is the Inspectors pane; at the bottom is the Libraries pane. Inspectors provide information about and options to customize a current selection. For example, when you select a label (a `UILabel` instance) in the embedded Interface Builder, you can set its alignment, background color, or dimensions using inspectors. Libraries offer pre-built components that you can incorporate into your project, and can include media and code snippets.

Both panes provide a set of tabbed subpanes in a similar fashion to the Navigator panes. Buttons appear at the top of each pane. Inspector buttons update to match the context of the current selection. For example, the object attributes and connections inspectors that appear for a `UILabel` object do not appear when working with text in a source code editor. That's because those Interface Builder–style inspectors have no meaning when working with source code. The Utility pane updates to match the currently selected item in the central editor.

Inspector shortcuts are Command-Option combined with a number, starting with 1 and 2 for the File (Command-Option-1) and Quick Help (Command-Option-2) inspectors, respectively. They increase numerically from there, depending on the available inspector utility panes. Typically IB-style inspectors include Identity (Command-Option-3), Attributes (Command-Option-4), Size (Command-Option-5), and Connections (Command-Option-6). Xcode 3.*x* users should note that the pane orders have been switched around a bit from what was used in 3.*x* Interface Builder, plus the old keyboard shortcuts no longer apply.

Libraries use Command-Control-Option shortcuts combined with a number. The library pane provides access to file templates (Command-Control-Option-1), code snippets (Command-Control-Option-2), objects (Command-Control-Option-3), and media (Command-Control-Option-4). You can add your own custom elements into the code snippets libraries to provide easy access to repeated code patterns.

Hide or reveal the Utilities section by typing Command-Control-Option-0.

### The Editor Window

The editor window includes a primary space, where content is displayed. Above this space, you'll find a "jump bar." This bar simplifies finding other files using any level of grouping. Each of the path-like elements provides a pop-up menu, offering quick access to those elements. Jump bars appear in many Xcode 4 roles. They all work similarly and introduce a hierarchical way to move around a structured system.

A special menu item, the small button with eight rectangles at the very left of the jump bar helps you find files related to the currently displayed item. Options include code counterparts (.m/.h file pairs), superclasses, subclasses, siblings, categories, included files, and more. This is a particularly powerful menu that you shouldn't overlook.

The central space of the editor window offers Quick Look technology to provide previews of nearly any kind of content you will add to your Xcode projects or will need to use in support of that project. This means you can use the content viewer not only for direct coding material, but also to review PDFs and keynote presentations that relate to developing the project, for example.

### Working with Multiple Editor Windows

You may want to work with several editor windows at once. To accomplish this, you can create a new workspace window, work in one window using tabs, or detach a floating editor. Both windows and tabs allow you to switch quickly between various editors and files.

To open new editor windows without taking an entire workspace with you, double-click any filename. A floating editor window appears above your workspace with the contents of that file.

To create a new workspace window, choose File > New > New Window (Command-Shift-T). The new window contains the same project or projects as the original workspace. Changes to a file in one window or tab are mirrored to the same file in other windows or tabs.

> **Note**
>
> To add line numbers to your source code editing windows, open Preferences (Xcode > Preferences, Command-,). Select the Text Editing pane and check Line Numbers.

To use tabs, choose View > Show Tab Bar. This reveals a workspace tab bar between the main toolbar and the panes below it. Create new tabs using File > New Tab (Command-T) or right-click (Control-click) in the tab bar and choose New Tab from the contextual pop-up. Alternatively, press Shift-Option while clicking a filename in the navigator and click the + button at the top-right to open the file in a new tab.

Xcode tabs follow Apple's standards, so if you're used to using Safari tabs, they'll work similarly in Xcode. To navigate between tabs from the keyboard use Command-Shift-[ to move left and Command-Shift-] to move right. Click + to add a new tab to your window. You can pull tabs out into their own windows and can drop tabs into existing windows by adding them to tab bars.

The Cocoa Samurai blog (cocoasamurai.blogspot.com) created a number of Xcode 4 keyboard shortcut reference guides. These infographics, which are hosted at github (github.com/Machx/Xcode-Keyboard-Shortcuts), provide an exhaustive guide to the key combinations you can use to navigate through Xcode.

> **Note**
>
> Xcode provides full Undo support for a single session. You can even undo past a previous save so long as you do so within the same session. That is, you cannot close a project, reopen it, and then revert changes made before the project was closed.

## Review the Project

When Xcode creates your new project, it populates it with all the basic elements and frameworks you need to build your first iOS application. Items you see in this project include the following:

- **Frameworks > Foundation and Core Graphics frameworks**—These essential frameworks enable you to build your iPhone applications and are similar to the ones found on OS X.
- **Frameworks > UIKit framework**—This framework provides iOS-specific user interface APIs and is key to developing applications that can be seen and interacted with on the iPhone screen.
- **Products > HelloWorld.app**—This placeholder is used to store your finished application. Like on the Macintosh, iPhone applications are bundles and consist of many items stored in a central folder.
- **Supporting Files > HelloWorld-Info.plist**—This file describes your application to the iPhone's system and enables you to specify its executable, its application identifier, and other key features. It works in the same way Info.plist files work on the Mac. Localizable strings for the property list can be found in the Supporting Files > InfoPlist.strings file(s).

- **MainStoryboard_iPhone.storyboard and MainStoryboard_iPad.storyboard**—These Interface Builder files create a minimally populated GUI for each platform. You will modify these in the upcoming walkthrough.
- **[*Prefix*]AppDelegate.h, [*Prefix*]AppDelegate.m, [*Prefix*]ViewController.h, [*Prefix*]ViewController.m, main.m**—These files contain a rough Objective-C skeleton that you can customize and expand to create your application. The prefix used by these files is set in the new project options screen. Feel free to browse through the code, but you will not edit these files in the upcoming walkthrough. Instead, you use the way that Xcode set them up and limit your modifications to the Interface Builder storyboards.

> **Note**
>
> To add frameworks to your project in Xcode 4, select your blue project file in the Project Navigator. In the central editor pane, click TARGETS > Target Name in the very left column. Click Build Phases and open the Link Binary With Libraries disclosure triangle. Click the + button, navigate to the framework you wish to add (you'll find standard frameworks under Device folders), select it, and click Add. An options pane opens. Uncheck Copy items into destination group's folder (if needed), make sure that your target remains checked, and click Finish. To remove a framework, simply select it and click -.

## Open the iPhone Storyboard

Locate the MainStoryboard_iPhone.storyboard file in the Project Navigator. Storyboards store Interface Builder layouts and can include all the screens (called "scenes") for a single application. Select the storyboard file to open it in the central editor so you can begin to edit the file. You will see a grid pattern in the background of the editor and a scene list on the left side of the window. This list initially consists of the single Hello World View Controller Scene. The scene appears in the gridded area and consists of an empty view on top and an associated object dock beneath it. Figure 3-5 shows how this looks.

The two icons in the dock represent elements of the interface you're editing. On the right is the view's owner—in this case, its view controller. It represents the view controllers attached to the view.

View controllers don't have a visual presentation. They manage views, but they don't display anything of their own. Each view controller has a property called "view" that is set to some `UIView` responsible for providing the actual onscreen presentation. Here, that view is displayed above the dock. You can see more about the controller by selecting it and opening View > Utilities > Show Identity Inspector. Observe the class listed in the inspector (`ESViewController` or similar).

The view controller element in the dock is called a "proxy." A proxy plays a role in IB but the object that it represents (the view controller) is not itself embedded in the .storyboard archive. This proxy represents the object that loads and owns the view.

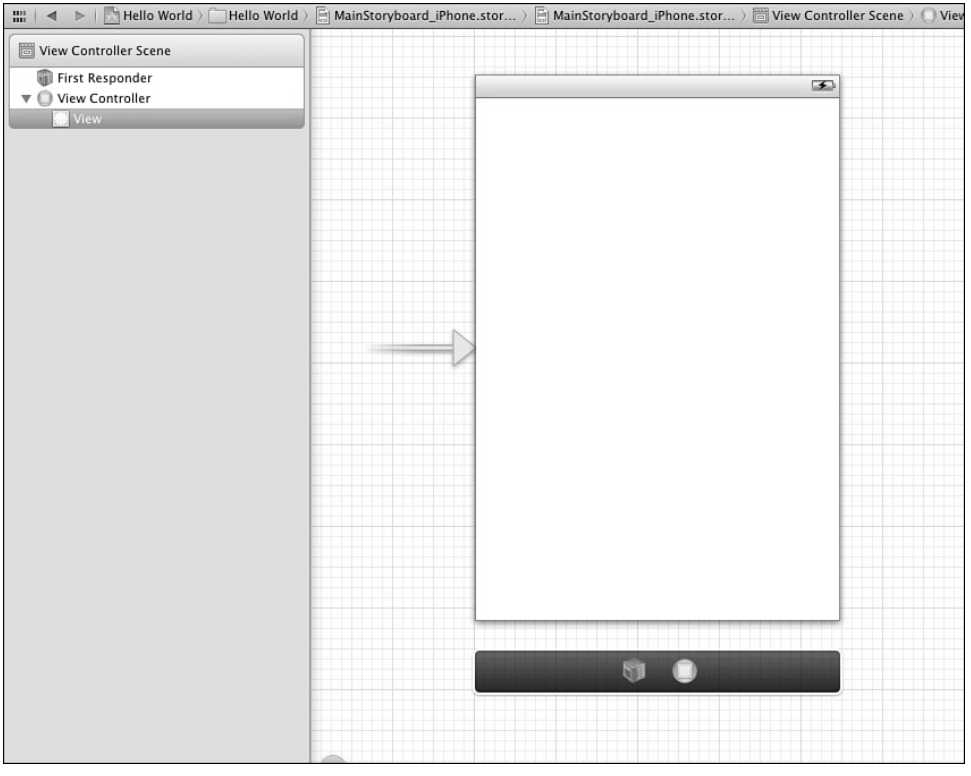


Figure 3-5 The Interface Builder window for an iPhone storyboard. A small dock floats below each storyboard scene, offering access to objects associated with its view.

To see how items are connected to each other, choose View > Utilities > Show Connections Inspector. You see an outlet listed called "view." Hover your mouse over this outlet and the view darkens. A small tooltip (saying "View") appears as well. That's because the `view` outlet for your view controller is already connected to that view. Outlet is IB-talk for "instance variable."

The other icon, the one that appears in the left position of the dock, is called First Responder. It looks like a dark orange cube with the number 1 on it. Like the view controller, it's a proxy object. It represents the onscreen object that is currently responding to user touches. During the lifetime of an application, the first responder changes as users interact with the screen. For example, imagine a form. As the user touches each text field in that form, that field becomes active and assumes the first responder role. At times you want to allow interface elements (such as buttons and switches) to control whatever item is the first responder. Connecting to this proxy in Interface Builder allows you to do so.

## Edit the View

To start customizing, click the big white view. By default, this view is empty although it shows a status bar at the top. It's up to you to customize this and add any content. To do so, you rely on two tools from the Utility area: the Interface Builder object library and the inspector.

Choose View > Utilities > Show Attributes Inspector (Command-Option-4). The Attributes Inspector lets you adjust the properties of the currently selected object—in this case, the view that you are editing. In the inspector, locate the View > Background Listing with its colored swatch. Click the swatch and choose a new color from the Colors palette. The view you are editing automatically updates its background color.

Next, open the object library by choosing the third icon at the top of the bottom pane. Alternatively, select View > Utilities > Show Object Library (Command-Control-Option-3). This library (see Figure 3-6) presents a list of prebuilt Cocoa Touch elements you can use in your IB files. These include both abstract elements such as view controllers as well as visual components such as buttons and sliders. Enter **label** in the search field at the bottom of the library. Drag the label from the middle pane and drop it onto your window. (Alternatively, double-click the label in the library. This automatically adds that item to your window.)

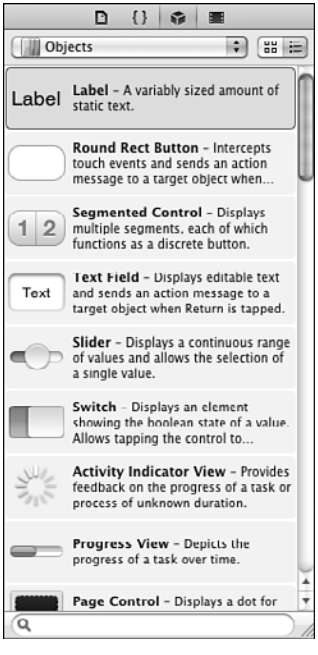


Figure 3-6 The Interface Builder object library.

Once it is dragged to the view, double-click the label and change the words from "Label" to "Hello World." You can also move the label around in the window to appeal to your aesthetic sensibilities or set its location in the Size Inspector. The Attributes Inspector allows you to set the font face and size, as well as the text color. You may need to resize your label to accommodate the new size. Unselect "Autoshrink" and note that there are two places to set the font size. The field in the main inspector sets the *minimum* font size for the label. The pop-up for the font sets the *desired* font size.

Save your project with File > Save (Command-S). You have now customized your iPhone window with this content.

Next, customize the iPad interface. Return to the Project Navigator and select MainStoryboard_iPad.storyboard. You'll notice that the iPad presentation is far larger than the iPhone one. You may not be able to see the entire iPad interface at once, even on relatively large screens.

Interface Builder allows you to double-click in the grid background to shrink the presentation to a more manageable size, but you cannot perform edits in this mode. You can also use the new zoom/shrink buttons at the bottom-right of the editor window. Many developers find it worth investing in a large vertical monitor rather than a horizontal one in order to better work with iPad edits in IB.

As before, change the view's background color, add a label ("Hello World on iPad," perhaps) and mess with its font and placement. Again, save your project (File > Save, Command-S).

## Run Your Application

Locate the pop-up in the workspace toolbar just to the right of the Start/Stop buttons at the top-left of the window. From this pop-up choose Hello World > iPhone Simulator. This tells Xcode to compile your project for the Macintosh-based iPhone simulator. If more than one simulator choice presents, select the most recent SDK (that is, 5.3 rather than 5.0).

Click the run button (by default it looks like a "Play" button) or type Command-R and then wait as Xcode gets to work. It takes a few seconds to finish compiling, and then Xcode automatically launches the simulator, installs your project, and runs it. Figure 3-7 shows the result, the Hello World application running on the simulator.

After testing the application on the iPhone simulator, click the stop button (to the right of the run button) and test the application using the iPad simulator. Select Hello World > iPad Simulator from the pop-up and again click the run button.

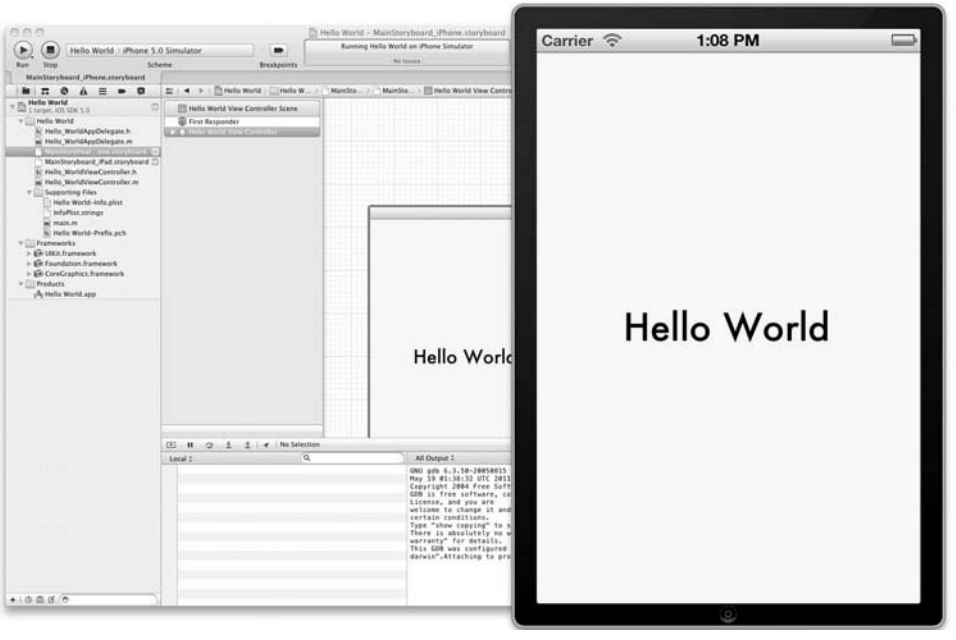


Figure 3-7 The customized Hello World application runs on the simulator.

# Using the Simulator

The iOS SDK simulator makes it possible to test applications on the Macintosh using many of the same actions a user would perform on an actual device. Because the Macintosh is not a handheld touch-based mobile system, you must use menus, keyboard shortcuts, and the mouse or trackpad to approximate iPhone-style interactions. Table 3-1 shows how to perform these tasks via the simulator.

Table 3-1 **Simulator Equivalents for iPhone Actions**

| Action | Simulator Equivalent |
|---|---|
| Selecting the device | Use Hardware > Device to simulate an original iPhone, Retina iPhone, or iPad-style device. Firmware versions are selectable via Hardware > Version. |
| Rotating the device | Hardware > Rotate Left (Command-left arrow) and Hardware > Rotate Right (Command-right arrow). The simulator supports all four major interface orientations: portrait, landscape left, landscape right, and portrait upside down. You cannot simulate face-up or face-down orientations. |

Table 3-1 **Simulator Equivalents for iPhone Actions**

| Action | Simulator Equivalent |
|---|---|
| Shaking the device | Hardware > Shake Gesture (Command-Control-Z). This simulates a shake using a motion event but does not simulate other accelerometer actions. I encourage you to avoid building applications that depend on users shaking devices, no matter how cool the feature appears. |
| Pressing the Home key | Click the Home button on the simulator screen or choose Hardware > Home (Command-Shift-H). |
| Locking the device | Hardware > Lock (Command-L). |
| Tapping and double-tapping | Click with the mouse, either a single- or double-click. |
| Tapping on the keyboard | Click the virtual keyboard or type on the Mac keyboard. You can use many Mac-style shortcuts for ease of testing, including Command-A, Command-C, and so on. |
| Dragging, swiping, and flicking | Click, drag, and release with the mouse. The speed of the drag determines the action. For flicks, drag very quickly. |
| Pinching in or out | Press and hold the Option key on your keyboard. When the two dots appear, drag them toward each other or away from each other. Hold down the Shift key to move the dot's origin point. |
| Running out of memory | Hardware > Simulate Memory Warning. This allows you to simulate a condition of low available memory, letting you test how your application responds. |
| In-progress phone call (visual display only) | Hardware > Toggle In-Call Status Bar. On the iPhone, you can run an application while on a phone call. The in-call bar appears at the top of the screen for the duration of the call. |
| Attaching a keyboard | Simulate the detection of a Bluetooth or docked hardware keyboard by selecting Hardware > Simulate Hardware Keyboard. |
| Attaching TV Out cables | Choose Hardware > TV Out to simulate the attachment of a VGA or HDMI cable to the dock connector. Use this to test your external screen code, and specifically to catch screen-attached and -detached notifications. A floating window shows the simulated output. |
| Changing zoom | Change the magnification of the simulator by selecting Window > Scale. Choose from 100%, 75%, and 50%. |
| Simulating a printer | Choose File > Open Printer Simulator to test your software with AirPrint. You can also use this simulator to test printing from a device. The printed output opens in Preview. |
| Capturing screenshots | Choose File > Save Screen Shot (Command-S) or copy the screen with Edit > Copy Screen (Command-Control-C). |

Table 3-1 **Simulator Equivalents for iPhone Actions**

| Action | Simulator Equivalent |
|---|---|
| Setting a simulated location | Use the Debug > Location menu to simulate where the iPhone is being used. Choose from a (stationary) custom location, Apple's HQ, Apple Stores, a city bike ride/run, or a drive down the freeway. |
| Slowing down animations | Choose Debug > Toggle Slow Animations to allow you to better view animations over a longer period of time. Use this feature to spot inconsistencies and flaws in your animations. |
| Highlighting potential rendering trouble spots | Use the four Debug > Color options to locate potential presentation issues. The items, which are toggled on and off via menu selection, include blended layers, copied images, misaligned images, and elements rendered off-screen. |
| Resetting the simulator | Choose iOS Simulator > Reset Contents and Settings to restore your simulator to its "factory fresh" original condition, deleting all current applications, settings, and user data. |

## Simulator: Behind the Scenes

Because the simulator runs on a Macintosh, Xcode compiles simulated applications for the Intel chip. Your application basically runs natively on the Macintosh within the simulator using a set of Intel-based frameworks that mirror the frameworks installed with iOS onto actual units. The simulator versions of these frameworks are typically located in the Xcode developer directory, in /Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator5.0.sdk/System/Library or some similar location. The actual location will vary by the version of the SDK you are using and where you have installed the SDK. The /Developer folder is the default location, but you can easily override this.

You can find your applications in your home's Library/Application Support folder. They are stored in iPhone Simulator/ in one of many firmware-specific folders, such as 3.1.2/, 4.2/, and 6.1/ under User/Applications/. It's helpful to visit these folders to peek under the hood and see how applications get deployed to the iPhone; these User/Applications/ folders mimic device installations. Other interesting development folders include the following:

- **/Developer/Platforms/iPhoneSimulator.platform/Developer/Applications**—Default location of the actual iPhone simulator application.
- **/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulatorX.X.sdk/Applications**—Location of the simulator's built-in applications, including Mobile Safari, the Address Book, and so forth. Replace X.X with the firmware version.
- **/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator5.0.sdk/System/Library/Frameworks**—Location of the Cocoa Touch frameworks you can link to from your application.

- **/Developer/Platforms/iPhoneOS.platform/Developer/Library/Xcode/Templates/Project Templates/Application**—Location of the individual project templates shown in Figure 3-1.
- **~/Library/MobileDevice/Provisioning Profiles**—Folder that stores your iPhone Portal provisioning profiles.
- **~/Library/Developer/Xcode/Archives**—Folder that stores archives built with the Build > Build and Archive option in Xcode. Archived applications appear in your Xcode Organizer, where they can be validated, shared, and submitted to iTunes Connect.
- **~/Library/Developer/Xcode/DerivedData**—Build and log folder.
- **~/Library/Developer/Xcode/Snapshots**—Project version control snapshots.
- **~/Library/Developer/Shared/Xcode/Screenshots/**—Folder for screenshots taken with the Organizer.
- **~/Library/Developer/Shared/Project Templates/**—Add custom templates to this folder to have them appear in the New Project screen.
- **~/Library/MobileDevice/Software Images**—Folder that stores iOS firmware files (.ipsw files) that can be installed onto your devices.
- **~/Library/Application Support/MobileSync/Backup**—Folder where iTunes stores iPhone, iPod touch, and iPad backup files.

> **Note**
>
> Application archives let you share applications as .ipa files. They also allow you to run the same validation tests that iTunes Connect uses to confirm that an application is properly signed with a development certificate before or as you submit your apps.

Each application is stored in an individual sandbox. The name of the sandbox is random, using a unique code (generated by `CFUUIDCreateString()`). You can zip up a sandbox folder and be able to share it between Macintoshes. The other Macintosh will need Xcode to be installed in able to access the simulator and its frameworks.

Each sandbox name hides the application it's hosting, so you must peek inside to see what's there. Inside you find the application bundle (HelloWorld.app, for example), a Documents folder, a Library folder, and a temporary (/tmp) folder. While running, each application is limited to accessing these local folders. They cannot use the main user library as applications might on a Macintosh.

With the exception of the Library/Caches folder, all the materials in an application's Documents and Library folders are backed up by iTunes when deployed to a device. The tmp folder is not backed up by iTunes. Use the Caches folder to store large, changing application-support data files that need to persist. Use the tmp folder for materials that iOS can dispose of between application launches.

If you want to clean out your applications' sandbox folders, you can delete files directly while the simulator is not running. You can also delete all the simulator data by choosing

iPhone Simulator > Reset Contents and Settings from the simulator itself. This erases applications, their sandboxes, and any current settings, such as nondefault language choices, that affect how your simulator runs.

Alternatively, use the press-and-hold-until-it-jiggles interface on the simulator that you're used to on the iPhone device itself. After you press and hold any icon for a few seconds, the application icons start to jiggle. Once in this edit mode, you can move icons around or press the corner X icon to delete applications along with their data. Press the Home button to exit edit mode.

Although applications cannot access the user library folder, you can. If you want to edit the simulator's library, the files are stored in the iPhone Simulator/User/Library folder in your home Application Support folder. Editing your library lets you test applications that depend on the address book, for example. You can load different address book sqlitedb files into Library/AddressBook to test your source with just a few or many contacts.

> **Note**
>
> The iPhone simulator and Mac OS X use separate clipboards. The simulator stores its own clipboard data, which it gathers from iOS copy/paste calls. When you use Edit > Paste (Command-V), the simulator pastes text from the Macintosh clipboard into the simulator's clipboard. You can then use the simulator's Edit menu (double-tap in a text box) to paste from the iOS clipboard into simulator applications.

### Sharing Simulator Applications

Simulator-compiled applications provide an important way to share test builds when developers are denied access to new hardware or when beta firmware is not widely distributed. They can also support interface design testing in advance of actual device deployment. Although unsuitable for full debugging and end-user usability tests (see Chapter 1, "Introducing the iOS SDK"), simulator builds do have a role and a purpose in iOS application life cycle.

To share an app, zip up its entire sandbox folder from one Macintosh and then extract it to another Mac's simulator application folder.

## The Minimalist Hello World

While exploring the iOS SDK, and in the spirit of Hello World, it helps to know how to build parsimonious applications. That is, you should know how to build an application completely from scratch, without five source files and two interface files. Here is a walk-through showing you exactly that—a very basic Hello World that mirrors the approach shown with the previous Hello World example but that manages to do so with one file and no .storyboard or xib files.

Start by creating a new project (File > New Project, Command-Shift-N) in Xcode. Choose Empty Application, click Next, enter **Hello World** as the product name, and set

your company identifier as needed (mine is com.sadun). Set the device family to Universal, uncheck Use Core Data, uncheck Include Unit Tests, and click Next. Save it to your desktop.

When the project window opens, select the two App Delegate files (.h and .m) from the project navigator and click delete or backspace to delete them. Choose Delete (formerly Also Move to Trash) when prompted.

Open Hello World > Supporting Files > main.m and replace its contents with Listing 3-1. The source is included in the sample code for this book (see the Preface for details), so you don't have to type it in by hand.

Listing 3-1 **Reductionist main.m**

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

// Simple macro distinguishes iPhone from iPad
#define IS_IPHONE (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPhone)

@interface TestBedAppDelegate : NSObject <UIApplicationDelegate>
{
    UIWindow *window;
}
@end

@implementation TestBedAppDelegate
- (UIViewController *) helloController
{
    UIViewController *vc = [[UIViewController alloc] init];
    vc.view.backgroundColor = [UIColor greenColor];

    // Add a basic label that says "Hello World"
    UILabel *label = [[UILabel alloc] initWithFrame:
        CGRectMake(0.0f, 0.0f, window.bounds.size.width, 80.0f)];
    label.text = @"Hello World";
    label.center = CGPointMake(CGRectGetMidX(window.bounds),
        CGRectGetMidY(window.bounds));
    label.textAlignment = UITextAlignmentCenter;
    label.font = [UIFont boldSystemFontOfSize: IS_IPHONE ? 32.0f : 64.0f];
    label.backgroundColor = [UIColor clearColor];
    [vc.view addSubview:label];

    return vc;
}

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
```

```
    window = [[UIWindow alloc] initWithFrame:
        [[UIScreen mainScreen] bounds]];
    window.rootViewController = [self helloController];
    [window makeKeyAndVisible];
    return YES;
}
@end

int main(int argc, char *argv[]) {
    @autoreleasepool {
        int retVal =
            UIApplicationMain(argc, argv, nil, @"TestBedAppDelegate");
        return retVal;
    }
}
```

So what does this application do? It builds a window, colors the background, and adds a label that says "Hello World." In other words, it does exactly what the first Hello World example did, but it does so by hand, without using Interface Builder.

The application starts in main.m by establishing the autorelease pool and calling `UIApplicationMain()`. From there, control passes to the application delegate, which is specified as the last argument of the call by naming the class. This is a critical point for building a non–Interface Builder project, and one that has snagged many a new iPhone developer.

The delegate, receiving the `application:didFinishLaunchingWithOptions:` message, builds a new window, querying the device for the dimensions (bounds) of its screen. It creates a new view controller, assigns that controller as its `rootViewController` property, and orders it out, telling it to become visible. Using the device's screen bounds ensures that the main window's dimensions match the device. For older iPhones, the window will occupy a 320×480-pixel area, for the iPhone 4 and later, 640×960 pixels, for the first two generations of iPads, 768×1024 pixels.

The `helloController` method initializes its view controller's view by coloring its background and adding a label. It uses `UI_USER_INTERFACE_IDIOM()` to detect whether the device is an iPhone or iPad, and adjusts the label's font size accordingly to either 32 or 64 points. In real-world use, you may want to perform other platform-specific adjustments such as choosing art or setting layout choices.

As you can see, laying out the label takes several steps. It's created and the text added, centered, aligned, and other features modified. Each of these specialization options defines the label's visual appearance, steps that are much more easily and intuitively applied in Interface Builder. Listing 3-1 demonstrates that you can build your interface entirely by code, but it shows how that code can quickly become heavy and dense.

In Xcode, the Interface Builder attributes inspector fills the same function. The inspector shows the label properties, offering interactive controls to choose settings such as left, center, and right alignment. Here, that alignment is set programmatically to the constant `UITextAlignmentCenter`, the background color is set to clear, and the label programmatically moved into place via its `center` property. In the end, both the by-hand and Interface Builder approaches do the same thing, but here the programmer leverages specific knowledge of the SDK APIs to produce a series of equivalent commands.

## Browsing the SDK APIs

iOS SDK APIs are fully documented and accessible from within Xcode. Choose Help > Developer Documentation (Command-Option-Shift-?) to open the Xcode Organizer > Documentation browser. The Documentation tab will be selected at the top bar of the window. Other tabs include iPhone, Repositories, Projects, and Archives, each of which plays a role in organizing Xcode resources.

The documentation you may explore in this window is controlled in Xcode's preferences. Open those preferences by choosing Xcode > Preferences (Command-,) > Documentation. Use the GET buttons to download document sets. Keep your documentation up to date by enabling "Check for and install updates automatically."

In the Developer Documentation organizer, start by locating the three buttons at the top of the left-hand area. From left to right these include an eye, a magnifying glass, and an open book. The eye links to explore mode, letting you view all available documentation sets. The magnifying glass offers interactive search, so you can type in a phrase and find matching items in the current document set. The open book links to bookmarks, where you can store links to your most-used documents.

Select the middle (magnifying glass) search button. In the text field just underneath that button locate another small magnifying glass. This second magnifying glass has a disclosure triangle directly next to it. Click that disclosure and select Show Find Options from the pop-up menu. Doing so reveals three options below the text field: Match Type, Doc Sets, and Languages. Use the Doc Sets pop-up to hide all but the most recent iOS documentation set. This simplifies your search results so you do not find multiple hits from SDK versions you're not actually using.

Enter **UILabel** into the search field to find a list of API results that match `UILabel`, as well as full text and title matches. The first item in the results list should link to the iOS Library version of the documentation. Refer to the jump bar at the top of the main area to locate which library you are viewing. This should read something like iOS Library > User Experience > Windows & Views > UILabel Class Reference. The UILabel Class Reference (see Figure 3-8) displays all the class methods, properties, and instance methods for labels as well as a general class overview.

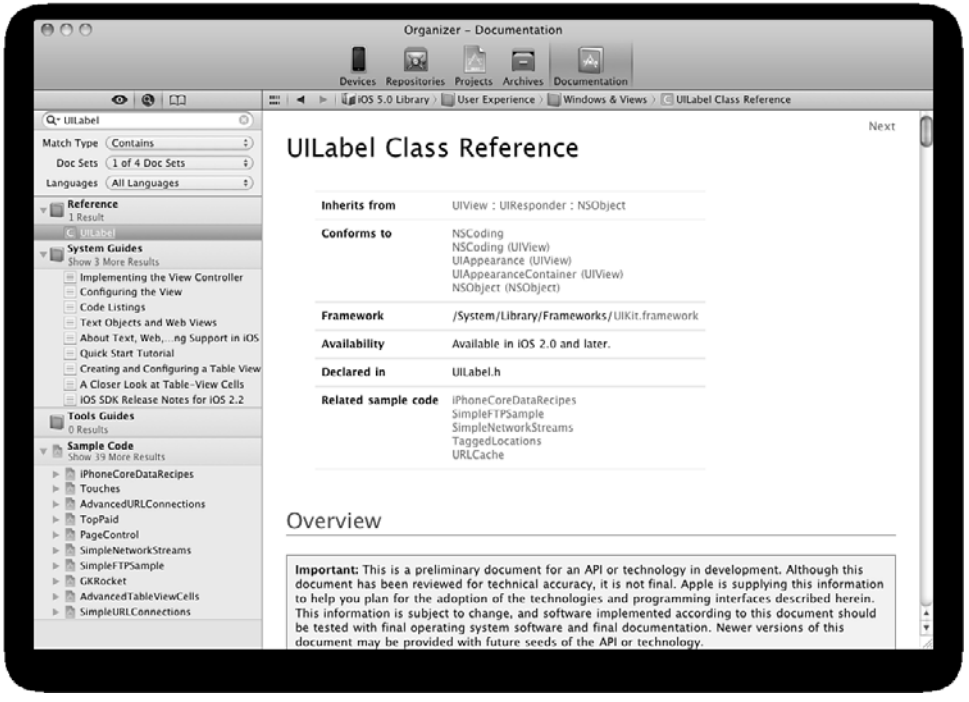


Figure 3-8 Apple offers complete developer documentation from within Xcode itself.

Apple's Xcode-based documentation is thorough and clear. With it you have instant access to an entire SDK reference. You can look up anything you need without having to leave Xcode. When material goes out of date, a document subscription system lets you download updates directly within Xcode.

Xcode 4 lost the handy class overview that appeared to the left of documentation in Xcode 3. The jump bar at the top embeds the same organization features (namely overview, tasks, properties, and so on). If you'd rather view the material with the old-style Developer Library overview, right-click in the class reference area and choose Open Page in Browser. Figure 3-9 shows the browser-based presentation with that helpful at-a-glance class Table of Contents to the left of the core material.

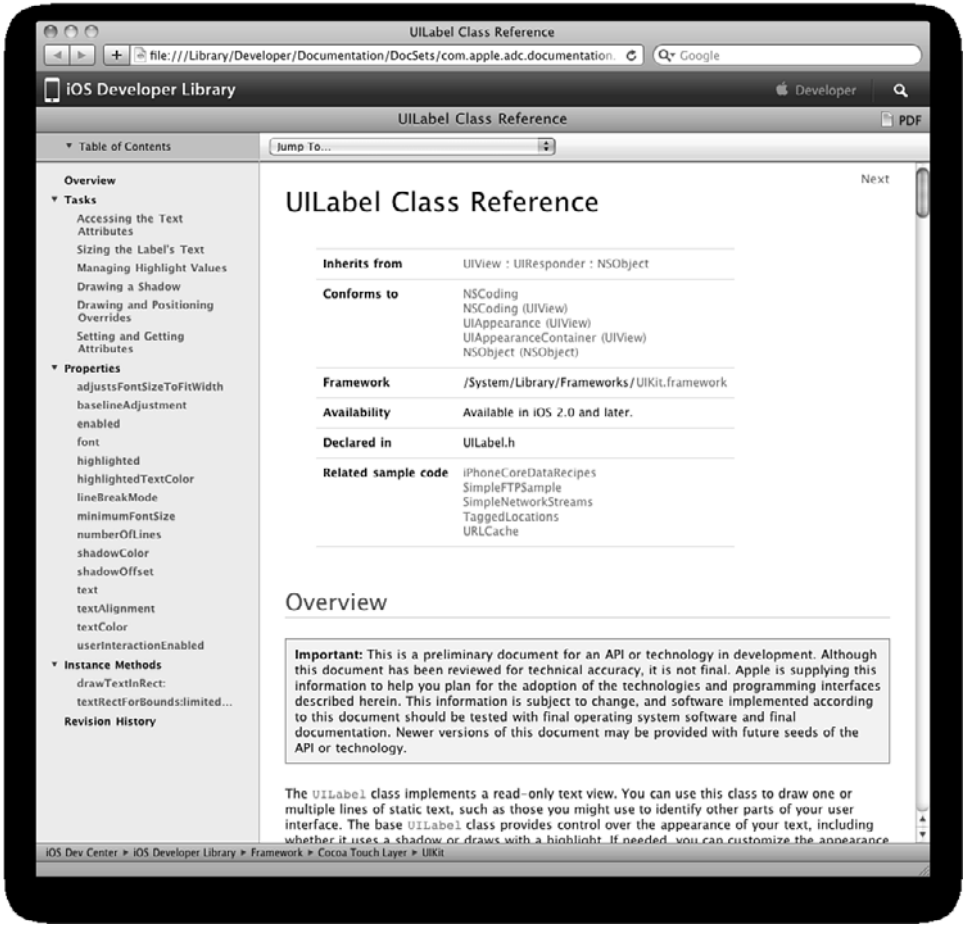


Figure 3-9 The "Table of Contents" view is no longer available from within Xcode itself but can be accessed via Open Page in Browser.

# Converting Interface Builder Files to Their Objective-C Equivalents

A handy open-source utility by Adrian Kosmaczewski allows you to convert Interface Builder files to Objective-C code. With it, you can extract all the layout information and properties of your visual design and see how that would be coded by hand. nib2objc does exactly what its name suggests. With it, you can generate converted code that takes into account the class constructors, method calls, and more. It works on .xib and .storyboard files, although some newer features such as segues are not yet exposed; under the hood both formats are simply XML.

Listing 3-2 shows the result of running nib2objc on the .xib file used in the first walk-through. Compare it to the far simpler (and less thorough) by-hand version in Listing 3-1. It performs more or less the same tasks. It creates a new label and then adds the label to the window. However, this conversion utility exposes all the underlying properties, of which just a few were edited in Listing 3-1.

To peek at the original IB XML, open the storyboard file in Text Edit. Issue `open -e` from the Terminal command line while in the HelloWorld project folder in the en.lproj subfolder:

```
open -e MainStoryboard_iPad.storyboard
```

> **Note**
>
> nib2obj is hosted at http://github.com/akosma/nib2objc and issued under a general "Use this for good not evil" style of license.

Listing 3-2 **HelloWorldViewController.xib after Conversion to Objective-C**

```
UIView *view3 = [[UIView alloc] initWithFrame:
    CGRectMake(0.0, 20.0, 320.0, 460.0)];
view3.frame = CGRectMake(0.0, 20.0, 320.0, 460.0);
view3.alpha = 1.000;
view3.autoresizingMask =
    UIViewAutoresizingFlexibleRightMargin |
    UIViewAutoresizingFlexibleBottomMargin;
view3.backgroundColor =
    [UIColor colorWithRed:0.963 green:1.000 blue:0.536 alpha:1.000];
view3.clearsContextBeforeDrawing = YES;
view3.clipsToBounds = NO;
view3.contentMode = UIViewContentModeScaleToFill;
view3.hidden = NO;
view3.multipleTouchEnabled = NO;
view3.opaque = YES;
view3.tag = 0;
view3.userInteractionEnabled = YES;

UILabel *view6 = [[UILabel alloc] initWithFrame:
    CGRectMake(72.0, 150.0, 175.0, 160.0)];
view6.frame = CGRectMake(72.0, 150.0, 175.0, 160.0);
view6.adjustsFontSizeToFitWidth = YES;
view6.alpha = 1.000;
view6.autoresizingMask =
    UIViewAutoresizingFlexibleLeftMargin |
    UIViewAutoresizingFlexibleRightMargin |
    UIViewAutoresizingFlexibleTopMargin |
    UIViewAutoresizingFlexibleBottomMargin;
view6.baselineAdjustment = UIBaselineAdjustmentAlignCenters;
view6.clearsContextBeforeDrawing = YES;
```

```
view6.clipsToBounds = YES;
view6.contentMode = UIViewContentModeLeft;
view6.enabled = YES;
view6.hidden = NO;
view6.lineBreakMode = UILineBreakModeTailTruncation;
view6.minimumFontSize = 10.000;
view6.multipleTouchEnabled = NO;
view6.numberOfLines = 1;
view6.opaque = NO;
view6.shadowOffset = CGSizeMake(0.0, -1.0);
view6.tag = 0;
view6.text = @"Hello World";
view6.textAlignment = UITextAlignmentLeft;
view6.textColor = [UIColor colorWithRed:0.000 green:0.000 blue:0.000 alpha:1.000];
view6.userInteractionEnabled = NO;

[view3 addSubview:view6];
[view2 addSubview:view3];
```

# Using the Debugger

Xcode's integrated debugger provides a valuable tool for iPhone application development. This walkthrough shows you where the debugger is and provides a simple grounding for using it with your program. In these steps, you discover how to set breakpoints and use the debugger console to inspect program details. These steps assume you are working on the second, minimalist Hello World example just described and that the project window is open and the main.m file displayed.

## Set a Breakpoint

Locate the `helloController` method in the main.m file of your Hello World project. Click in the leftmost Xcode window column, just to the left of the `label` assignment line. A blue breakpoint indicator appears (see Figure 3-10). The dark blue color means the breakpoint is active. Tap once to deactivate—the breakpoint turns light blue—and once more to reactivate.

Remove breakpoints by dragging them offscreen or right-clicking them; add them by clicking in the column, next to any line of code. Once added, your breakpoints appear in the workspace's Breakpoint Navigator (Command-6). You can delete breakpoints from the navigator (select, then press the Delete button) and can deactivate and reactivate them from there as well.

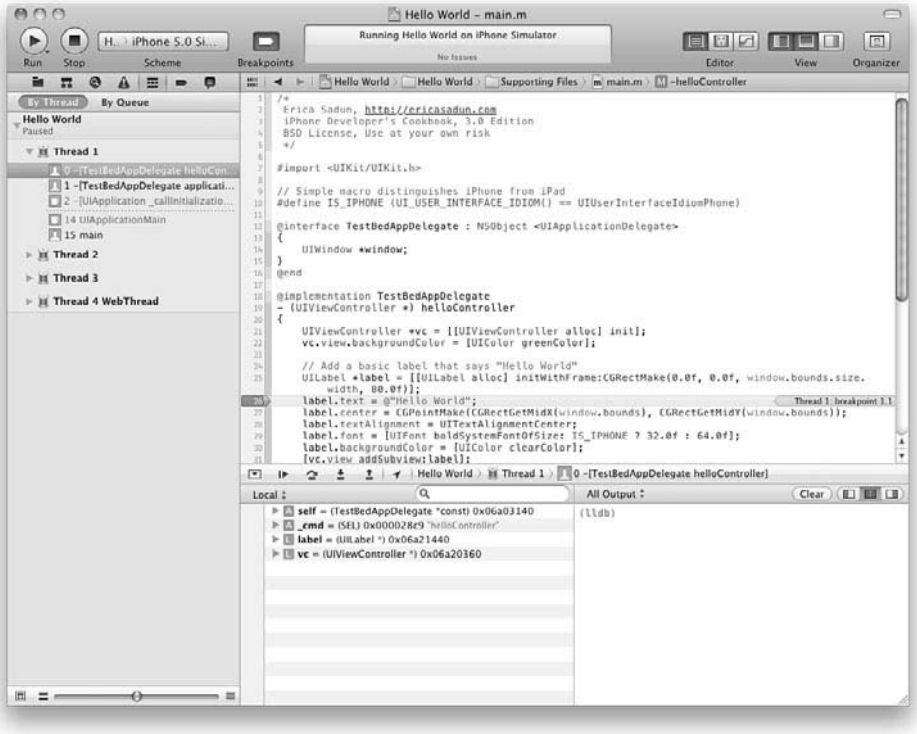


Figure 3-10 Blue breakpoint indicators appear in the gutter to the left of the editor area. You can reveal the debugger at the bottom of the workspace by clicking the Debugger disclosure button while running an application or by clicking the center of the three View buttons at the top-right of the workspace window at any time.

## Open the Debugger

Compile the application (Product > Build, Command-B) and run it (Product > Run, Command-R). The simulator opens, displays a black screen, and then pauses. Execution automatically breaks when it hits the breakpoint. A green bar appears next to your breakpoint, with the text "Thread 1: Stopped at breakpoint 1."

In Xcode, the debugging pane appears automatically as you run the application. You can reveal or hide it manually by clicking the middle of the three View buttons at the top-right of the workspace window; it looks like a rectangle with a dark bottom. When the debugger is shown, you can drag its jump bar (not the one at the top of the editor window, but the one at the top of the debugger) upward to provide more room for your output.

The debugger provides both a graphical front end for inspecting program objects as well as a text-based log area with an interactive debugger console. Xcode offers two command-line debuggers: gdb and lldb. The LLDB project (hosted at http://llvm.org) expands

upon the standard GNU debugger (gdb) with improved memory efficiency and Clang compiler integration.

Select which debugger you wish to use by editing your project scheme, as shown in Figure 3-11. Select Edit Scheme from the pop-up at the left of your toolbar at the top of your workspace window just to the right of the Run and Stop buttons. (Make sure you're selecting the Hello World part of the pop-up, not the iPhone or iPad simulator part.) Use the Info > Debugger pop-up to switch between GDB and LLDB. Click OK.

Figure 3-11 The project scheme editor allows you to select which debugger you prefer to use.

## Inspect the Label

Once stopped at the breakpoint, the interactive debugger and the debugger command line let you inspect objects in your program. Using lldb, you can look at the label by typing `print-object label` or, more simply `po label` at the command line. Use `print` or `p` to print non-objects, such as integer values.

```
(lldb) po label
(UILabel *) $3 = 0x06a3ded0 <UILabel: 0x6a3ded0; frame = (0 0; 320 80);
clipsToBounds =
YES; userInteractionEnabled = NO; layer = <CALayer: 0x6a3df40>>
```

At this time, the label's text has not yet been set and it has not yet been added to the view controller's view. You can confirm that the label is not yet added to the view by

looking at the view controller's view's subviews, currently an empty array, and the label's superview, currently `nil`.

```
(lldb) po [[vc view] subviews]
(id) $4 = 0x06811e20 <__NSArrayI 0x6811e20>(

)

(lldb) po [label superview]
(id) $5 = 0x00000000 <nil>
(lldb)
```

You can also view the label directly using the inspector at the left side of the debugger. Locate `label` and click the disclosure triangle to the left of it to show the properties of the label object. The label's `_text` field is set to `<nil>`.

The Step Into button appears to the left of the debugger jump bar. It looks like an arrow pointing down to a small black line. Click it once. The text assignment executes and the green arrow moves down by one line. The summary of the `label.text` updates. It should now say something like "Hello World (iPhone)." Confirm by inspecting the label from the command line by typing `po label` again. The label has updated its text instance variable to Hello World.

```
(lldb) po label
(UILabel *) $12 = 0x06a3ded0 <UILabel: 0x6a3ded0; frame = (0 0; 320 80); text =
'Hello
World'; clipsToBounds = YES; userInteractionEnabled = NO; layer = <CALayer:
0x6a3df40>>
```

If you want to get really crazy, you can override values directly by using `p` or `print` to make interpreted calls during the execution of your application. This call changes the label text from "Hello World" to "Bye World." It does that by executing the items within the square brackets, casting the void return to an integer, and then printing the (meaningless) results.

```
(lldb) p (int)[label setText:@"Bye World"]
(int) $13 = 117671936
(lldb) po label
(UILabel *) $14 = 0x06a3ded0 <UILabel: 0x6a3ded0; frame = (0 0; 320 80); text = 'Bye
World'; clipsToBounds = YES; userInteractionEnabled = NO; layer = <CALayer:
0x6a3df40>>
```

## Set Another Breakpoint

You can set additional breakpoints during a debugging session. For example, add a second breakpoint just after the line that sets the text alignment to center, on the line that sets the background color. Just click in the gutter next to `label.backgroundColor`, or wherever you want to set the breakpoint.

Confirm that the current alignment is set to 0, the default value, by inspecting the label's `textLabelFlags`—you will have to open the disclosure triangles in the column to the left of the lldb interface to see these label attributes. With the new breakpoint set, click the Resume button. It appears as a right-pointing triangle with a line to its left.

HelloWorld resumes execution until the next breakpoint, where it stops. The green arrow should now point to the `backgroundColor` line, and the explicit alignment flag updates from 0 to 1 as that code has now run, changing the value for that variable.

```
(lldb) p (int) [label textAlignment]
(int) $19 = 1
```

You can further inspect items by selecting them in the left-hand debugging inspector, right-clicking, and choosing Print Description from the contextual pop-up menu. This sends the `description` method to the object and echoes its output to the console.

> **Note**
>
> Remove breakpoints by dragging them out from the left column or by deleting them in the Breakpoint Navigator.

List your breakpoints by issuing the `breakpoint list` command. Lldb prints out a list of all active breakpoints.

```
(lldb) breakpoint list
Current breakpoints:
1: file ='main.m', line = 26, locations = 1, resolved = 1
baton: 0x11b2dd380
  1.1: where = Hello World`-[TestBedAppDelegate helloController] + 365 at
main.m:26,
  address = 0x0000211d, resolved, hit count = 1

3: file ='main.m', line = 30, locations = 1, resolved = 1
baton: 0x12918d6c0
  3.1: where = Hello World`-[TestBedAppDelegate helloController] + 912 at
main.m:30,
  address = 0x00002340, resolved, hit count = 0
```

> **Note**
>
> Learn more about setting and controlling breakpoints by visiting the LLVM website. The official lldb tutorial is found at lldb.llvm.org/tutorial.html.

## Backtraces

The bottom pane of the debugging window offers text-based debugger output that can mirror results from other panes. For example, open the Debug Navigator and view the application by thread, disclosing the trace details for Thread 1. This provides a trail of execution, showing which functions and methods were called, in which order to get you to the current point where your application is.

Next, type **backtrace** or **bt** at the text debugger prompt to view the same trace that was shown in that navigator. After stopping at the second breakpoint, the backtrace should show that you are near (for example, line 26 in the source from main.m). You will see this line number toward the beginning of the trace, with items further back in time appearing toward the end of the trace. This is the opposite of the view shown in the Debug Navigator, where more recent calls appear toward the top of the pane.

### Console

This bottom text-based debugger is also known as the console. This pane is where your `printf`, `NSLog`, and `CFShow` messages are sent by default when running in standard debug mode or when you use the simulator. You can resize the console both by adjusting the jump bar up or down and also by dragging the resize bar between the left and right portions of the debugger. If you want, you can use the show/hide buttons at the top-right of the debugger. Three buttons let you hide the console, show both the console and the visual debugger, or show only the console. To the left of these three buttons is a Clear button. Click this to clear any existing console text.

To test console logging, add a `NSLog(@"Hello World!");` line to your code; place it after adding the label as a window subview. Remove any existing breakpoints and then compile and run the application in the simulator. The log message appears in the Debug area's console pane. The console keeps a running log of messages throughout each execution of your application. You can manually clear the log as needed while the application is running.

### Add Simple Debug Tracing

If you're not afraid of editing your project's .pch file, you can add simple tracing for your debug builds. Edit the file to add the following macro definition:

```
#ifdef DEBUG
    #define DebugLog(...) NSLog(@"%s (%d) %@", __PRETTY_FUNCTION__, __LINE__,
    [NSString stringWithFormat:__VA_ARGS__])
#else
    #define DebugLog(...)
#endif
```

## Memory Management

iOS does not offer garbage collection. It relies on a reference counted memory management system. The new LLVM ARC extensions introduce automated reference counting, letting the compiler take care of many management issues for you. ARC automates when objects are retained and released, simplifying development. That doesn't mean you don't have to worry about memory:

- Even with ARC, you remain responsible for letting go of resources that create low-memory conditions. If you hold onto lots of multimedia assets such as video, audio, and images, you can exhaust memory—even in ARC-compiled applications.
- Many developers continue to use manual retain/release (MRR) development, especially to avoid refactoring production-critical code. Using MRR means you must control when objects are created, retained, and released in that code because ARC will not handle that for you.
- ARC does not automatically extend to Core Foundation and other C-based class code. Even if CF classes are toll-free bridged, ARC does not assume control of their instances until they are bridged into the Objective-C world.

As a developer, you must strategize how to react to low-memory conditions. Use too much memory and the iPhone warns your application delegate and `UIViewControllers`. Delegates receive `applicationDidReceiveMemoryWarning:` callbacks; view controllers get `didReceiveMemoryWarning`. Continue to use too much memory and the iPhone will terminate your application, crashing your user back to the iOS home screen. As Apple repeatedly points out, this is probably not the experience you intend for your users, and it will keep your application from being accepted into the App Store.

You must carefully manage memory in your programs and release that memory during low-memory conditions. Low memory is usually caused by one of two problems: leaks that allocate memory blocks that can't be accessed or reused, and holding onto too much data at once. Even on newer iOS devices, such as the iPad 2, your application must behave itself within any memory limits imposed by the operating system.

Every object in Objective-C is created with an integer-based retain count. So long as that retain count remains at 1 or higher, objects will not be deallocated. That rule applies in ARC code just as it applies in MRR. It is up to you as a developer to implement strategies that ensure that objects get released at the time you will no longer use them.

Every object built with `alloc`, `new`, or `copy` starts with a retain value of 1. Whether developing with ARC or MRR, if you lose access to an object without reducing the count to 0, that lost object creates a leak (that is, memory that is allocated and cannot be recovered). The following code leaks an array:

```
NSArray *leakyArray = [NSArray arrayWithObjects:@"Hello", @"World", nil];
CFArrayRef leakyRef = (__bridge_retained CFArrayRef) leakyArray;
leakyRef = nil;
```

## Recipe: Using Instruments to Detect Leaks

Instruments plays an important role in tuning your applications. It offers a suite of tools that lets you monitor and evaluate performance. For example, its leak detection lets you track, identify, and resolve memory leaks within your program. Recipe 3-1 shows an application that creates two kinds of leaks on demands: one created by using CF bridging

without a proper release, the other by introducing a strong reference cycle that cannot be resolved by ARC at the termination of its method.

To see Instruments in action, load the sample project for Recipe 3-1. Choose one of the simulator options as your destination from the leftmost pop-up in the toolbar at the top of your workspace, just to the right of the Run and Stop buttons. Then click and hold the Run button until the pop-up shown in Figure 3-12 appears. Choose Profile and then agree to whatever impediments Xcode throws in your direction (if any), such as stopping any currently running application, and so on.

Figure 3-12 Select Profile from the Run button's pop-up.

Instruments launches and asks you to select a Trace Template. Choose iOS Simulator > Leaks and then click Profile. Instruments opens a new profiling window, launches the application in the simulator, and starts running. Click the Stop button at the top-left of the Instruments window. There are some adjustments you'll want to make.

In the left-hand column, under the individual Instruments traces, you'll see an item labeled "Allocations." It is just below a slider and just above the Heapshot Analysis. This is a pop-up, as indicated by the arrows to its right. Use the pop-up to change from Allocations to Leaks.

With Leaks selected, the first item is now Snapshots. Change the Snapshot Interval from 10 seconds to 1. This lets you see updates in "real" time; even so, be patient. Instruments detects leaks during its snapshots, with a slight lag after performing the snapshot.

You are now ready to start a fresh trace. Click Record. The application relaunches in the simulator. With Instruments and the simulator both running, click one of the two buttons in the title bar to leak memory:

- The CF Bridging button leaks a simple 16-byte `NSArray`.
- The Retain Cycle button leaks two 16-byte `NSArray` objects that are connected to each other via a retain cycle, for a total of 32 bytes.

Memory leaks appear in Instruments as orange markers, scaled to the size of the leaks. The Leaked Blocks pane appears at the bottom of the window, as shown in Figure 3-13, once you click the Leaks trace row at the top of the window.

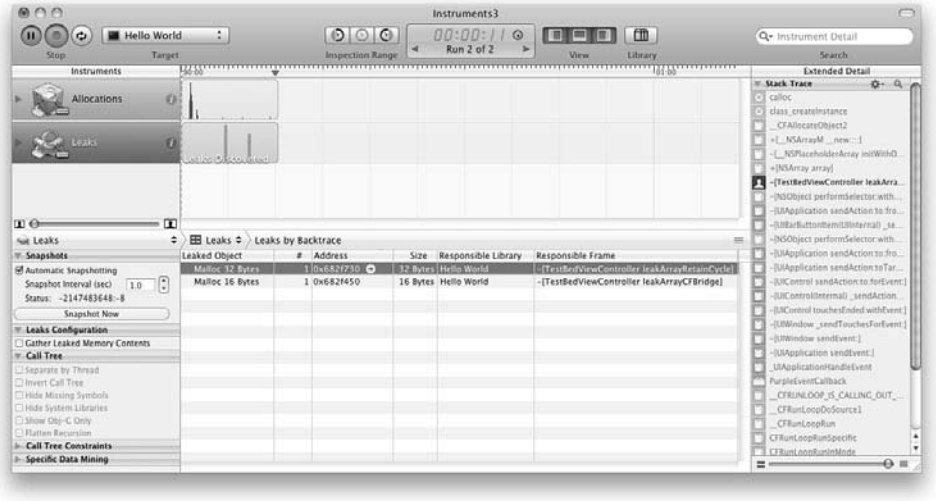


Figure 3-13 Instruments tracks leaks created by memory blocks that cannot be addressed or recovered by your code.

The trace shown in Figure 3-13 presents two leak events. The first orange marker corresponds to tapping the Retain Cycle button; the second to CF Bridging. The responsible frame for the first leak is shown to be the `leakArrayRetainCycle` method. A stack trace appears in the extended detail pane on the right side of the view. This pane is shown or hidden using the rightmost of the three View buttons at the top toolbar of the Instruments window. My "Hello World" code is noted to be the responsible library for both of the leaks, allowing me to further recognize that the leaked memory originated in my code.

> **Note**
>
> When working with possible retain cycles, use the jump bar in the center of the window (the quartered square followed by the word "Leaks" in Figure 3-13) to choose Cycles. This is a feature that Apple is still evolving. During the time this book was being written (that is, the beta period), it functioned less and crashed more. Hopefully Apple will finish refining this potentially valuable feedback before the iOS 5 beta goes golden.

Recipe 3-1 **Creating Programmatic Leaks**

```
- (void) leakArrayRetainCycle
{
    NSMutableArray *array1 = [NSMutableArray array];
    NSMutableArray *array2 = [NSMutableArray array];
    [array1 addObject:array2];
    [array2 addObject:array1];
}
```

```
- (void) leakArrayCFBridge
{
    NSArray *array = [NSArray arrayWithObjects:
        @"Hello", @"World", nil];
    CFArrayRef leakyRef = (__bridge_retained CFArrayRef) array;
    leakyRef = NULL;
}
```

> **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 3 and open the project for this recipe.

# Recipe: Using Instruments to Monitor Cached Object Allocations

When you load too much data at once, you can also run short of memory. Holding onto everything in your program when you are using memory-intense resources such as images, audio, or PDFs may cause problems. A strategy called **caching** lets you delay loads until resources are actually needed and release that memory when the system needs it.

The simplest approach involves building a cache from an `NSMutableDictionary` object. A basic object cache works like this: When queried, the cache checks to see whether the requested object has already been loaded. If it has not, the cache sends out a load request based on the object name. The object load method might retrieve data locally or from the Web. After the data is loaded, the cache stores the new information in memory for quick recall.

This code performs the first part of a cache's duties. It delays loading new data into memory until that data is specifically requested. (In real life, you probably want to type your data and return objects of a particular class rather than use the generic `id` type.)

```
- (id) retrieveObjectNamed: (NSString *) someKey
{
    id object = [self.myCache objectForKey:someKey];
    if (!object)
    {
        object = [self loadObjectNamed:someKey];
        [self.myCache setObject:object forKey:someKey];
    }
    return object;
}
```

The second duty of a cache is to clear itself when the application encounters a low-memory condition. With a dictionary-based cache, all you have to do is remove the objects. When the next retrieval request arrives, the cache can reload the requested object.

```
- (void) respondToMemoryWarning
{
    [self.myCache removeAllObjects];
}
```

Combining the delayed loads with the memory-triggered clearing allows a cache to operate in a memory-friendly manner. Once objects are loaded into memory, they can be used and reused without loading delays. However, when memory is tight, the cache does its part to free up resources that are needed to keep the application running.

## Simulating Low-Memory Conditions

One feature of the simulator allows you to test how your application responds to low-memory conditions. Selecting Hardware > Simulate Memory Warning sends calls to your application delegate and view controllers, asking them to release unneeded memory. Instruments, which lets you view memory allocations in real time, can monitor those releases. It ensures that your application handles things properly when warnings occur. With Instruments, you can test memory strategies such as caches, discussed earlier in this chapter.

Recipe 3-2 creates a basic object cache. Rather than retrieve data from the Web or from files, this cache builds empty `NSData` objects to simulate a real-world use case. When memory warnings arrive, as shown in Figure 3-14, the cache responds by releasing its data.

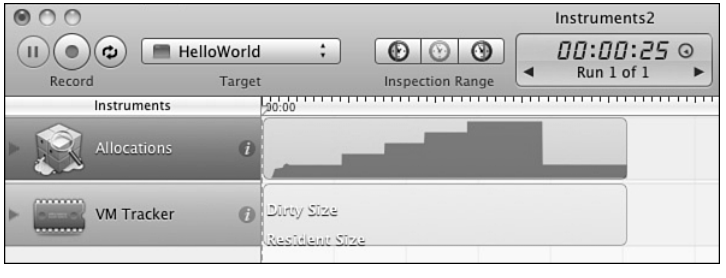


Figure 3-14 Instruments helps monitor object allocations, letting you test your release strategies during memory warnings.

The stair-step pattern shown here represents four memory allocations created by pressing the Consume button while using Instrument's Allocation profiler. After, the simulator issued a memory warning. In response, the cache did its job by releasing the images it had stored. The memory then jumped back down to its previous levels.

Instruments lets you save your trace data, showing the application's performance over time. Stop the trace and then choose File > Save to create a new trace file. By comparing runs, you can evaluate changes in performance and memory management between versions of your application.

Some SDK objects are automatically cached and released as needed. The `UIImage imageNamed:` method retrieves and caches images in this manner, as does the `UINib nibWithNibName:bundle:`, which preloads NIBs into a memory cache for faster loading. When memory grows low, these classes empty their caches to free up that memory for other use.

Recipe 3-2 **Object Cache Demo**

```
@implementation ObjectCache
@synthesize myCache, allocationSize;

// Return a new cache
+ (ObjectCache *) cache
{
    return [[ObjectCache alloc] init];
}

// Fake loading an object by creating NSData of the given size
- (id) loadObjectNamed: (NSString *) someKey
{
    if (!allocationSize) // pick your allocation size
        allocationSize = 1024 * 1024;

    char *foo = malloc(allocationSize);
    NSData *data = [NSData dataWithBytes:foo length:allocationSize];
    free(foo);
    return data;
}

// When an object is not found, it's loaded
- (id) retrieveObjectNamed: (NSString *) someKey
{
    if (!myCache)
        self.myCache = [NSMutableDictionary dictionary];
    id object = [myCache objectForKey:someKey];
    if (!object)
    {
        if ((object = [self loadObjectNamed:someKey]))
            [myCache setObject:object forKey:someKey];
    }
    return object;
}

// Clear the cache at a memory warning
- (void) respondToMemoryWarning
```

```
{
    [myCache removeAllObjects];
}
@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 3 and open the project for this recipe.

# Analyzing Your Code

The LLVM/Clang static analyzer automatically helps detect bugs in Objective-C programs. It's a terrific tool for finding memory leaks and other issues, especially with Core Foundation and MRR code. In Xcode, choose Product > Analyze (Command-Control-B). The issue markers shown in Figure 3-15 guide you through all suspected leaks and other potential problems. Use the Issue Navigator pane to dive into each issue and walk through the logic that leads the analyzer to raise a flag of concern.

Issues found by the static analyzer are not necessarily bugs. It's possible to write valid code that Clang identifies as incorrect. Always critically evaluate all reported issues before making any changes to your code.

## From Xcode to Device: The Organizer Interface

Choose Window > Organizer (Command-Shift-2) to open the Xcode Organizer window shown in Figure 3-16. This window forms the control hub for access between your development computer and your iOS testbed. It allows you to manage your credentials, add and remove applications, examine crash logs, and snap screenshots of your unit while testing your application.

The Organizer consists of two primary sections, the Library and the Devices, which have overlapping functionality. The library offers what is, basically, a merged inbox of device logs, provisioning profiles, and screenshots. These are broken out on a per-device basis in the Devices section.

In addition, the library has a Developer Profile section for organizing your developer certificates and a Software Images section for managing firmware bundles. The Devices list adds per-device consoles and per-device application management.

## Devices

The Devices list shows the name and status of those devices you've authorized as development platforms. The indicators to the right of each name show whether the device is attached (green light) or not (white light). A blank to the right of the device name indicates a unit that has not been set up for development or that has been "ignored"—that is, removed from the active list. An amber light appears when a device has just been attached.

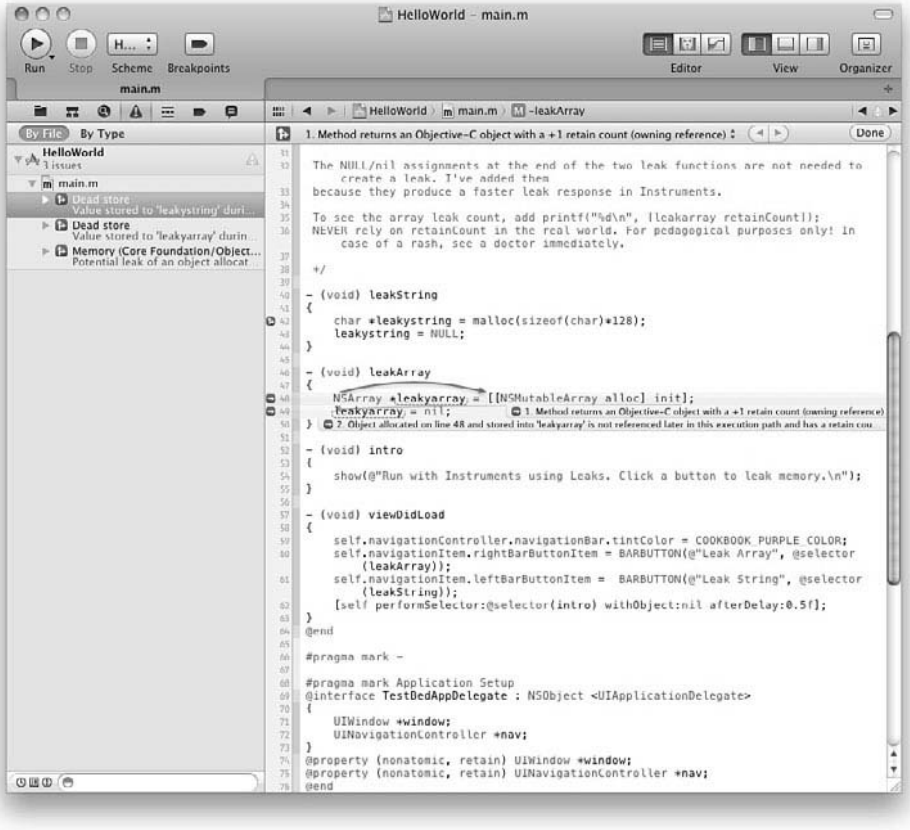


Figure 3-15 The Clang static analyzer creates bug reports for source code and embeds them into your Xcode editor window. Clang is useful for both MRR (as shown here) and ARC and provides reports specific to each compilation style.

Should the light remain amber colored, you may have encountered a connection problem. This may be due to iTunes syncing, and the unit is not yet available, or there may be a problem connecting with the onboard services, in which case a reboot of your iOS device usually resolves any outstanding issues.

A disclosure button appears to the left of each device, offering access to device information specific to that device. The items you find here mirror many of those listed in the Library section of the Organizer, including device logs, screenshots, and provisioning profiles. In addition, you have access to the device console and an application list.

Figure 3-16 Xcode's Organizer window provides a central hub for managing your devices, certificates, and provisions.

## Summary

Selecting a device name offers an overview of that device, including the capacity, serial number, and identifier of your unit. Here is also where you can load the latest firmware onto a device. Select a firmware version from the software pop-up and click Restore. Other items on this screen include overviews of current provisions, applications, device logs, and screenshots.

Be warned. Device downgrades are not always possible when you've upgraded to a newer (especially beta) iOS version. Downgrades must be authorized by Apple's signature servers, even when you have stored the older firmware locally on your computer. These servers allow Apple to control which older firmware they will and will not allow to be installed, essentially providing them with a rolling firmware recall system that disallows older firmware over time.

## Provisioning Profiles

Each developer license allows you to provision your personal or corporate iOS devices for testing. The Provisioning list shows a list of application provisions available to your unit. You can add or delete provisions from this screen.

Provisions determine which applications may or may not be run on the device. As a rule, only development and ad hoc distribution provisions are listed here, which makes sense. Distribution provisions are used to sign applications for the App Store, not for any specific device.

## Device Logs

Get direct access to your crash logs by selecting a particular crash (labeled with the application name and the date and time of the crash) from the scrolling list on this screen. The crash details, including a stack trace, thread information, exception types, and so forth, appear in the right-hand pane. You can import and export logs using the buttons on this screen.

In addition to crash logs that you generate yourself, you can also retrieve crash reports from users from their home computer and from iTunes Connect. The iPhone automatically syncs crash reports to computers when units back up to iTunes. These reports are stored in different locations depending on the platform used to sync the device:

- **Mac OS X**—/Users/*UserName*/Library/Logs/CrashReporter/MobileDevice/*DeviceName*
- **Windows XP**—C:\Documents and Settings*UserName*\Application Data\Apple Computer\Logs\CrashReporter\MobileDevice*DeviceName*
- **Windows Vista**—C:\Users*UserName*\AppData\Roaming\Apple Computer\Logs\CrashReporter\MobileDevice*DeviceName*

iTunes Connect collects crash log data from your App Store users and makes it available to you. Download reports by selecting Manage Your Applications > App Details > View Crash Report for any application. There you find a list of the most frequent crash types and Download Report buttons for each type.

Copy reports into the Mac OS X crash reporter folder and they load directly into the Organizer. Make sure to load them into the device folder for the currently selected device. The reports appear in LIBRARY > Device Logs.

Once in the Organizer, Xcode uses the application binary and .dSYM file to replace the hexadecimal addresses normally supplied by the report with function and method names. This process is called "symbolication." You don't have to manually locate these items; Xcode uses Spotlight and the application's unique identifier (UID) to locate the original binary and .dSYM files so long as they exist somewhere in your home folder. Xcode's archive feature offers the best way to keep these materials together and persistently available.

As with crash logs in the Organizer, the reports from users provide a stack trace that you can load into Xcode to detect where errors occurred. The trace always appears in reverse chronological order, so the first items in the list were the last ones executed.

In addition to showing you where the application crashed, Crash Reports also tell you why they crashed. The most common cause is `EXC_BAD_ACCESS`, which can be generated by accessing unmapped memory (`KERN_INVALID_ADDRESS`) or trying to write to read-only memory (`KERN_PROTECTION_FAILURE`).

Other essential items in the crash report include the OS version of the crash and the version of the application that crashed. Users do not always update software to the latest release, so it's important to distinguish which crashes arose from earlier, now potentially fixed, versions.

> **Note**
>
> See Apple Technical Note TN2151 for more details about iOS crash reporting.

## Applications

Each device offers a browseable list of installed applications. Use the – button to remove selected applications. To install an application, drag it onto the list or use the + button to browse for it. Make sure your application is compiled for iOS and that the device is provisioned to run that application. If you self-sign an application and install it to a device—the how-to process for doing so is described later in this chapter—your application uses your team provision. When added, applications immediately sync over to the device. Applications installed from the App Store do not appear in the application list any more, the way they did in Xcode 3.

To download the data associated with an application, click the Download button to the right of its name. Choose a destination and click Save. Xcode builds an archive bundle and populates it with the contents of the sandbox—namely the Documents, Library, and tmp directories. Xcode also adds the folder to the Projects and Sources list, where you can browse the contents directly from the Organizer.

You can reverse this process and add edited sandboxes back to the device. Locate the bundle you created. Drop new items into any of the enclosed subfolders and then drag the entire folder back onto the application name at the bottom of the Summary pane. Xcode reads the new items and instantly transfers them to the device. This is a great way to pre-populate your sandbox with test material.

## Console

Use the console to view system messages from your connected units. This screen shows `NSLog()` calls and other messages sent to `stderr` (standard error output) as you're running software on the tethered iPhone. You need not be using Xcode's debugger to do this. The console listens in to any application currently running on the device.

In addition to the debugging messages you add to your iPhone applications, you also see system notices, device information, and debugging calls from Apple's system software.

It can be viewed as basically a text-based mess, but there's a lot of valuable information you can gain. Click Save Log As to write the console contents out to disk or click Clear to empty the Console backlog.

### Screenshots

Snapshot your tethered iPhone's screen by clicking the New Screenshot button on the Screenshot display. The screenshot feature takes a picture of whatever is running on the iPhone, whether or not your applications are open. So you can access shots of Apple's built-in software and any other applications running on the iPhone.

Once snapped, images can be dragged onto the desktop or saved as an open project's new Default.png image ("Save as Launch Image"). Archival shots appear in a library on the left side of the window. To delete a screenshot, select one and press the Delete key to permanently remove it. Other features on this screen allow you to export images and compare images to highlight differences between separate shots.

> **Note**
>
> Screenshots are stored in your home Library/Application Support/Developer/Shared/Xcode/Screenshots folder.

## Building for the iOS Device

Building for and testing in the simulator takes you only so far. The end goal of iOS development is to create applications that run on actual devices. There are three ways to do so: building for development, for App Store distribution, and for ad hoc deployment. These three, respectively, allow you to test locally on your device, to build for the App Store, and to build test and review versions of your applications that run on up to 100 registered devices. Chapter 1 introduced mobile provisions and showed how to create these in the Apple iOS Developer Program portal. Now it's time to put these to use and deploy a program to the device itself.

### Using a Development Provision

A development provision is a prerequisite for iOS deployment. Your team provisioning profile is automatically created and managed by Xcode. You can also create your own wildcard dev provision at Apple's provisioning portal if desired, but it's not really necessary anymore for basic development now that Xcode has introduced the team profile.

The Xcode Organizer (Command-Shift-2) provides the hub around which you can manage your provisions, certificates, and devices. Figure 3-16 shows the Organizer window with the Provisioning Profiles organizer displayed.

To enable automatic device provisioning, check the Automatic Device Provisioning box shown at the bottom of Figure 3-16. This option allows you to register new devices with Apple directly from the Organizer. Xcode automatically uploads device information to the developer portal and downloads an updated provision that adds the new device.

For the times you need to add provisions to Xcode directly, click the Import button at the bottom of the window. Navigate to the provision, select it, and click Open. The Provisioning Profiles organizer also allows you to view the provision creation and expiration dates.

The Developer Profile organizer lists all your iOS and Mac developer certificates for both development and distribution. The Import and Export buttons at the bottom of this organizer allow you to package up your developer identities for easy secure transfer to other computers. These certificates are stored in your system keychain. You may want to review your keychain and ensure that the WWDR (Worldwide Developer Relations) certificate is available for use. It is not listed in the Developer Profile organizer directly.

During compilation, Xcode matches the currently selected provision against your keychain identities. These must match or Xcode will be unable to finish compiling and signing your application. To check your certificates, open Keychain Access (from /Applications/Utilities) and type **developer** in the search box on the top right. You should see, at a minimum, your Apple Worldwide Developer Relations certifications authority and one labeled iPhone Developer followed by your (company) name.

## Enable a Device

Tether a device that you wish to test on to your computer. You may need to wait for it to finish syncing in iTunes, first. For serious development, you can open Preferences in iTunes (Command-,), select the Devices tab, and check Prevent iPods, iPhones, and iPads from Syncing Automatically. Click OK to apply your new settings.

You can add devices to your account directly from Xcode. Select a device in the Xcode organizer (Window > Organizer, or Command-Shift-2). Right-click (or Control-click) its name and choose Add Device to Provisioning Portal (see Figure 3-17).

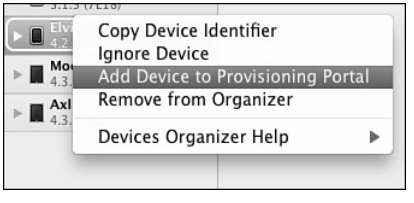


Figure 3-17 Use the Device organizer to add devices to the iOS provisioning portal.

Xcode will prompt you to log in to the iPhone provisioning portal with your program credentials. Once you're authenticated, it will upload the device details and generate (or regenerate) your team provisioning profile.

First-time developers are sometimes scared that their device will be locked in some "development mode," mostly due to Apple's standard warning text; in reality, I have heard of no long-lasting issues. Regardless, do your homework before committing your device as a development unit. Read through the latest SDK release notes for details.

## Inspect Your Application Identifier

Your project application identifier can be inspected and updated as needed. Select the project in the Project Navigator and choose TARGETS > Project Name. Select the Info tab to reveal the Custom iOS Target Properties, as shown in Figure 3-18. The application identifier can be set manually by editing the Bundle Identifier field. Xcode defaults to using your RFC 1034 reverse domain root identity followed by the product name.

Figure 3-18 The Info tab allows you to edit the Bundle identifier.

Your team development provision automatically matches all projects; its registered identifier is a single wildcard asterisk (*). Other provisions may or may not match the application identifier you are using. If you registered a wildcard application identifier of, say, com.sadun.* and used that to generate a provisioning profile, it would match com.sadun.helloworld or com.sadun.testing, for example, but not helloworld or com.mycompany.helloworld.

## Set Your Device and Code Signing Identity

After checking your identifier, click PROJECT > *project name* > Build Settings. Enter **device** into the search field at the top-right of the Build Settings pane. This should match one setting: Targeted Device Family. Use this pop-up to select which devices you wish to compile for: iPhone, iPad, or iPhone/iPad. This last choice allows you to build a universal application that can install and run on both devices, taking advantage of each system's native geometry.

Next, confirm your code-signing identity. Make sure you are looking at All settings (not just Basic ones). Enter **signing** in the top-right search field. Select your identity from the pop-up lists that appear to the right of each build type. As you start to accumulate provisions and identities, the list of options can become long, especially if you get involved in beta testing for third parties.

The two Automatic Profile Selectors automatically pick the first matching profile. I am paranoid enough to always inspect both the certificate name and the profile identity just above that name before choosing a profile. Apple recommends using automatic selection.

## Set Your Base and Deployment SDK Targets

The Base SDK target setting specifies what version of the SDK is used to compile your application. Open PROJECT > *project name* > Build Settings and locate Base SDK at the top of the list. As a rule, you may keep this option set to Latest iOS. It will automatically match the most recently installed SDK. That means your code will not fail compilation if it uses the newest introduced APIs. The compiler will handle these correctly. However, your code may still fail at execution if new APIs are called on devices whose firmware does not yet support them—for example, calling a 5.1 API on a 4.3 device. That's a problem you handle not with the Base SDK, but with the deployment target (see Figure 3-19).

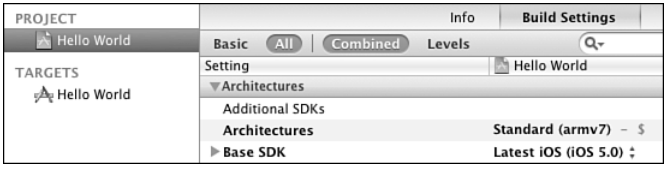


Figure 3-19 The Base SDK sets the iOS version used to compile your applications. The armv7 architecture is currently available on the iPhone 3GS and newer, the iPod touch 3G and newer, and all iPads.

Set your deployment target in TARGET > project name > Summary > iOS Application Target in the first section of the Summary view. This pop-up (see Figure 3-20) specifies the earliest device that you wish to allow your application to install to.

If you compile in 5.*x* and deploy to 4.*x*, you can use 5.*x* calls in your code but you will need to use runtime checks to ensure that you do not call APIs on platforms that do not support them and weak linking for any frameworks that aren't found on the deployment target. Both runtime and compile-time code checks are covered later in this chapter. Setting the deployment target to the base SDK target ensures that you will never have to make any runtime API checks but limits your audience to only those customers who have updated their units to the latest firmware. The more you support earlier firmware releases, especially within the same iOS release family, such as 4.*x*, 5.*x*, and so forth, the more you increase your potential user base.

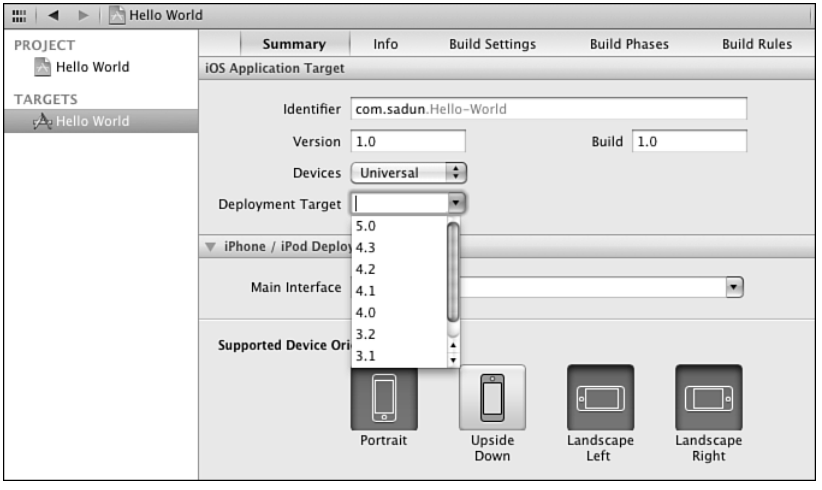


Figure 3-20 Deployment Target sets the earliest iOS version that is permitted to run your application.

## Compile and Run the Hello World Application

Finally, it's time to test Hello World on an actual iPhone, iPod touch, or iPad. Before you compile, you must tell Xcode to build for the iOS device's ARM architecture rather than the simulator's Intel one. Locate the scheme pop-up at the top-left of your workspace's toolbar and open it. It should look something like Figure 3-21. Device names appear at the top of the list, simulator choices at the bottom. Xcode highlights devices using firmware matching the deployment target but that are earlier than the Base SDK. Select a device.

Figure 3-21 Xcode makes a point of highlighting devices whose firmware lags behind the Base SDK. Older firmware installations form a vital component of your testing base to help ensure your code works on all valid deployment targets.

Choose Product > Run (Command-R) or click the Run button (it looks like a Play button) at the left of the workspace's toolbar. Assuming you have followed the directions earlier in this chapter properly, the Hello World project should compile without error, copy over to the iPhone, and start running.

If the project warns you about the absence of an attached provisioned device, open the Xcode Organizer window and verify that the dot next to your device is green. If this is not the case, you may need to restart Xcode or reboot your device or your computer.

### Signing Compiled Applications

You can sign already compiled applications at the command line using a simple shell script. This works for applications built for development. Signing applications directly helps developers share applications outside of ad hoc channels.

```
#! /bin/bash

export CODESIGN_ALLOCATE=/Developer/Platforms/iPhoneOS.platform/Developer/usr/
bin/codesign_allocate

codesign -f -s "iPhone Developer" $1.app
```

If you use several iPhone Developer profiles in your keychain, you may need to adapt this script so that it matches only one of those; otherwise, codesign complains about ambiguous matching.

I personally use this approach to distribute test versions of the sample code from this book. Using developer code-signing skips the hassles of ad hoc distribution, allowing a rapid testing turn around without using up valuable device slots.

## Detecting Simulator Builds with Compile-Time Checks

Xcode directives issue instructions to the compiler that can detect the platform you're building for. This lets you customize your application to safely take advantage of device-only features when they're available. Adding `#if` statements to your code lets you block or reveal functionality based on these options. To detect if your code is compiled for the simulator or for an iOS device, use a compile-time check:

```
#if TARGET_IPHONE_SIMULATOR
    Code specific to simulator
#else
    Code specific to iPhone
#endif
```

## Performing Runtime Compatibility Checks

There is a real and measurable adoption lag among iOS device users. To sell your application to the greatest number of customers, do not build for any SDK higher than your lowest desired customer. Use common sense when picking that SDK. You may find it best to support the most recent firmware dot release that lags a few weeks or months behind

the latest iOS update, or you may want to extend support to all versions of the current iOS firmware.

The hardest decisions come when iOS moves forward to a new version. That's when the adoption rate slows down the most. The 3.*x*/4.*x* overlap lasted for some time before developers gained confidence in moving forward to 4.*x*-only releases. iOS 5 and its successors continue that challenge.

To build for a range of possible deployment targets, set your Base SDK to the highest version of the OS you want to target—usually the current SDK release. Set your iOS deployment target to the lowest OS version you intend to build for.

When compiling to a deployment target that is earlier than your base SDK, you'll need to perform runtime checks for any classes or API calls that might not yet have been introduced to a given firmware platform. For example, an iOS 5-introduced API call will not be available on an iOS 4.2 installation. The following tests allow you to check at runtime for classes and methods that may or may not be available depending on the deployment platform:

- **Check platform geometry**—Introduced in iOS 3.2, the idiom check lets you determine if your code is running on an iPhone-like unit (iPhone, iPod touch) or on an iPad. I have not included extensions here to see whether the check itself is implemented simply because I cannot recommend deploying to pre-3.2 platforms anymore. As of the Summer of 2011, 3.*x* platforms made up only about 5% of all iOS installations. Pre-3.2 installs were a vanishingly small percentage of that.

```
if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) . . .
```

- **Check deployment platform**—Use the model property of the UIDevice class to determine whether the unit is running on an `@"iPhone"` or `@"iPad"`. More nuanced platform detection is discussed in Chapter 14, "Devices Capabilities."

```
if ([[UIDevice currentDevice].model isEqualToString:@"iPad"]) . . .
```

- **Check system prefix**—It's not the best way to approach coding (checking classes and instances directly is far more reliable), but you *can* check the current system version directly.

```
if ([[[UIDevice currentDevice] systemVersion] hasPrefix:@"5."])  . . .
```

- **Check for properties using key-value coding**—You can determine whether an object offers a value for a given key within the current scope, which allows you to test for properties.

```
UILabel *label = (UILabel *)[cell valueForKey:@"textLabel"];
if (label) [label setText:celltext];
```

- **Check for class existence**—If a class may not exist on a given deployment target, use `NSClassFromString()` to test whether the class can be produced from its name. Pass it an `NSString`, such as the following:

```
if (NSClassFromString(@"NSAttributedString")) . . .
```

- **Check for function existence**—You can test for functions before attempting to call them.

```
if(&UIGraphicsBeginImageContextWithOptions != NULL) . . .
```

- **Check for selector compliance**—You can test objects to see whether they respond to specific selectors. When newer APIs are supported, objects will report that they respond to those selectors, letting you call them without crashing the program. You may generate compile-time warnings about unimplemented selectors unless you use workarounds such as `performSelector:withObject:`. If you really want to go hard core, you can build `NSInvocation` instances directly, as discussed in Chapter 3.

```
if ([cell respondsToSelector:@selector(selectorName:)]) . . .
```

# Pragma Marks

Pragma marks organize your source code by adding bookmarks into the method list pop-up button at the top of each Xcode window. This list shows all the methods and functions available in the current document. Adding pragma marks lets you group related items together, as shown in Figure 3-22. By clicking these labels from the drop-down list, you can jump to a section of your file (for example, to `Utility` or `RAOP`) as well as to a specific method (such as `-hexStringFromBytes:`).

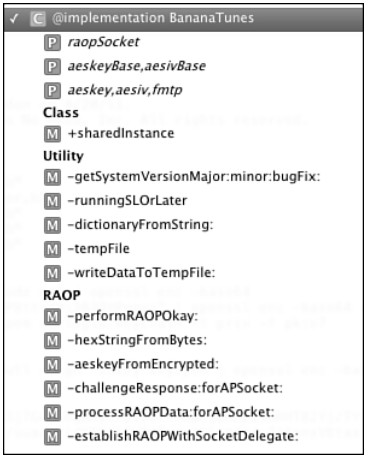


Figure 3-22 Use pragma marks to organize your method and function list.

To create a new bookmark, add a pragma mark definition to your code. To replicate the first group in Figure 3-22, for example, add the following:

```
#pragma mark Class
```

You can also add a separation line with a single dash. This uses a shortcut to add a spacer plus a mark:

```
#pragma mark -
```

or

```
#pragma mark - Marker Title
```

The marks have no functionality and otherwise do not affect your code. They are simply organizational tools you choose to use or not.

### Collapsing Methods

When you need to see more than one part of your code at once, Xcode lets you close and open method groups. Place your mouse in the gutter directly to the left of any method. A pair of disclosure triangles appears. Click a triangle, and Xcode collapses the code for that method, as shown in Figure 3-23. The ellipsis indicates the collapsed method. Click the disclosure triangle, and Xcode reveals the collapsed code. You can also collapse any compound statement.

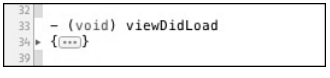


Figure 3-23 Xcode lets you collapse individual methods and functions. This allows you to see parts of your program that normally would not fit onscreen together.

## Preparing for Distribution

Building for distribution means creating a version of your application that can be submitted to Apple for sale in the App Store. Before you begin submitting, know how to clean up builds, how to edit and use schemes, and how to create an archive of your application. You want to compile for the App Store with precision. Cleaning first, then compiling for distribution helps ensure that your applications will upload properly. Archiving produces an application bundling that can be shared and submitted. The section covers these skills and others used with distribution compiles.

### Locating and Cleaning Builds

Xcode 4 uses a new approach to build code and store built products. In Xcode 3, compiled code was added to a "build" subfolder in your project folder. In Xcode 4, applications are created in your home library by default, in ~/Library/Developer/Xcode/DerivedData. You can locate the built product by selecting it in Project Navigator in the Products group. Right-click the application and choose

Show in Finder. Project Archives, which are used for building products that are shared with others or submitted to the App Store, are stored in ~/Library/Developer/Xcode/Archives.

Cleaning your builds forces every part of your project to be recompiled from scratch. Performing a clean operation also ensures that your project build contains current versions of your project assets, including images and sounds. You can force a clean by choosing Product > Clean (Command-Shift-K).

Apple recommends cleaning before compiling any application for App Store review, and it's a good habit to get into.

## Using Schemes and Actions

In Xcode, schemes store project build settings. They specify what targets to build, what configuration options to apply, and how the executable environment will be set up. Actions are individual tasks that you can perform to create builds as well as to test, profile, analyze, and archive your application.

Each scheme consists of a project and a destination. Figure 3-24 shows the scheme pop-up for the Hello World project from this chapter. Access this pop-up by selecting Hello World to the right of the Run button. This is a funny kind of pop-up in that it's actually split. Clicking to the left or the right of the embedded chevron provides two slightly different menus.

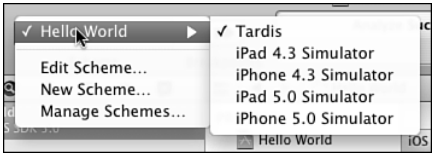


Figure 3-24 The scheme pop-up menu consists of schemes and their run destinations, followed by management options.

Clicking the left produces the pop-up shown in Figure 3-24. This pop-up includes a list of schemes (just one here) and their possible run destinations. Here, destinations include the device and four simulator styles (iPhone and iPad, for iOS 4.3 and iOS 5.0). Following the scheme list is a menu line followed by three options to edit, create, and manage schemes.

Clicking the right shows only the destination submenu, allowing you to pick a destination for the current scheme. Destinations include any provisioned and development-enabled iOS devices as well as all possible simulator choices for your deployment build settings.

> **Note**
>
> The plural of scheme in Xcode is schemes, not schema or schemata. As iOS developer Nate True puts it, "It's schemata non grata" in Xcode.

With your scheme selected, choose Edit Scheme from the pop-up menu to see a list of available actions. Figure 3-25 shows the actions associated with the Hello World scheme from Figure 3-24. Each action represents an individual development task. They include the following:

- **Build**—Building the code
- **Run**—Running the compiled code on a device or in the simulator
- **Test**—Using Xcode unit test tools that help stress application features
- **Profile**—Running the code with live sampling using Instruments
- **Analyze**—Applying the static analyzer to unresolved issues
- **Archive**—Preparing the application for sharing or submission to the App Store

Figure 3-25 The actions list for a scheme offer a set of tasks that may need to be performed during the creation of an application.

Use this scheme editor to customize how each action works in Xcode. For example, in the Run action shown in Figure 3-25, the debugger is set to LLDB. You can easily change this to GDB by selecting it from the pop-up.

You've already seen the action menu in the Instruments walkthrough earlier in this chapter. Access it by clicking and holding the Run menu at the top-left corner of the Xcode window. Select the action you want to perform from the pop-up. The Run button updates to show the new action item.

## Adding Build Configurations

Build configurations allow you to specify how your application should be built. They act as a quick reference to the way you want to have everything set up, so you can be ready to compile for your device or for the App Store just by selecting a configuration. They also are useful for adding instrumentation and conditional compilation. Standard Xcode projects offer Debug and Release configurations. You may want to create a few others, such as one for ad hoc distribution.

Configurations are created and managed on the PROJECT > *project name* > Info screen in the Configurations section (see Figure 3-26). Assuming you've been following along in this chapter, you have already set up the HelloWorld project and edited its debug build settings. It uses your team wildcard provision to sign the application. Instead of editing the build settings each time you want to switch the signing provision, you can create a new configuration instead. Typical configuration options include which architectures you wish to build for and your default code-signing identity.

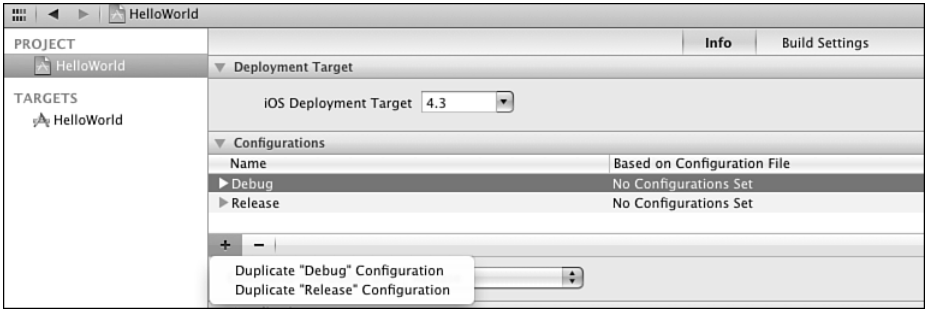


Figure 3-26 Use the Info > Configurations pane to create new configurations so you can build with preset options such as signing identities.

In the following sections, you'll read about adding an ad hoc configuration to your project. This is really for example purposes only—because I cannot really think of many other ways you'd want to change build settings for ad hoc versus App Store and because you can pick your signing identity in the Organizer when creating bundles for both destinations. Adapt this example and its steps for your real-world requirements.

> **Note**
>
> You can distinguish your ad hoc configuration build by using a different bundle ID, product name, or by adding extra alerts, to make it easier to run along the App Store version. Use `#ifdef` directives to distinguish ad hoc code changes.

## About Ad Hoc Distribution

Apple allows you to distribute your applications outside the App Store via ad hoc distribution. With ad hoc, you can send your applications to up to 100 registered devices and run those applications using a special kind of mobile provision that allows the applications to execute under the iPhone's FairPlay restrictions. Ad hoc distribution is especially useful for beta testing and for submitting review applications to news sites and magazines.

The ad hoc process starts with registering devices. Use the iPhone Developer Program portal to add device identifiers (Program Portal, Devices) and names to your account. Recover these identifiers from the iPhone directly (use the `UIDevice` class), from Xcode's Organizer (copy the identifier from the overview tab), from iTunes (click on Serial Number in the iPhone's Summary tab), or via the free Ad Hoc Helper application available from iTunes. Enter the identifier and a unique username.

To create a new ad hoc configuration, create a new provision at the iOS portal. Select Program Portal > Provisioning > Distribution. Click Add Profile. Select Ad Hoc, enter a profile name, your standard wildcard application identifier (for example, com.yourname.*), and select the device or devices to deploy on. Don't forget to check your identity and then click Submit and wait for Apple to build the new mobile provision. Download the provision file and drop it onto the Xcode application icon.

With your new provision on hand, duplicate the Release configuration. Xcode creates a copy and opens a text entry field for its name. Edit the name from Release copy to Ad Hoc.

Next, click the Build Settings tab and locate the Code Signing section. Assign your new ad hoc provision to your ad hoc configuration. Use the pop-up to the right of Any iOS SDK. The selected provision then appears in the Code Signing Identity list, as shown in Figure 3-27.

Figure 3-27 New configurations appear separately in the Code Signing section of your build settings.

To finish, choose your new build configuration in the Archive action section of the Scheme editor.

> **Note**
>
> In addition to adding new configurations by hand, you can also use build configuration files that contain setting definitions in text format. Create these files by selecting File > New > New File > Other > Configuration Settings. You can read more about creating configuration files in Apple's Xcode Build System Guide.

## Building Ad Hoc Packages

When you're ready to share your app with members of your ad hoc list, follow these steps:

1. If you wish to use a custom compiler scheme, choose Edit Scheme from the Scheme pop-up at the top of the Xcode window. Select Archive and set your Build Configuration as desired. Click OK. If you want to use the default compiler scheme, you may skip this step.
2. Choose Product > Archive from the main Xcode menu.
3. Wait for Xcode to work. The Organizer will open when it is finished, displaying the Archives tab. This screen allows you to review, manage, and use archived copies of your applications. You can also add free-form comments to each archive—a very handy feature—and a status line lets you track when archives have been successfully submitted to the App Store.
4. For ad hoc distribution, click Share. Choose iOS App Store Package (.ipa) as your destination.
5. Choose your ad hoc provision from the Identity pop-up and click Next.
6. Specify the name of your new package (for example, HelloApp) and where to save it. Click Save.

After following these steps, you can now e-mail the newly built .ipa file to members of your beta list or post it to a central site for download.

The .ipa bundle you built is actually a renamed ZIP file. If you extract it and look inside the Payload folder, you'll discover the compiled application bundle. This contains an embedded mobile provision that enumerates all the device IDs included in the ad hoc provisioning.

```
<string>Wildcard Ad Hoc </string>
<key>ProvisionedDevices</key>
```

> **Note**
>
> TestFlight offers over-the-air ad hoc management that has become quite popular in the developer community. Visit testflightapp.com for details.

# Over-the-Air Ad Hoc Distribution

You can distribute ad hoc ipa files over the air by creating links to a simple webpage. The itms-services: URL scheme, when pointing to an application manifest property list allows your users to install apps wirelessly. You provide the ipa and the manifest on the website. Here's how you might link to the manifest.

```
<a href="itms-services://?action=download-manifest&\
    url=http://example.com/manifest.plist">Install App</a>
```

Make sure your website is configured to support the following two MIME types.

```
application/octet-stream ipa
text/xml plist
```

## Building a Manifest

The manifest is an XML-based property list. It must contain six key/value pairs:

- **`URL`**—a fully-qualified URL pointing to the ipa file
- **`display-image`**—a fully-qualified URL pointing to a 57×57-pixel PNG icon used during download and installation
- **`full-size-image`**—a fully-qualified URL pointing to a 512×512-pixel PNG (not JPEG!) image that represents the iTunes app
- **`bundle-identifier`**—the app's standard application identifier string, as specified in the app's Info.plist file
- **`bundle-version`**—the app's current bundle version string, as specified in the app's Info.plist file
- **`title`**—a human-readable application name

In addition to these required keys, you can specify an optional md5 hash for file elements. Listing 3-3 shows a sample manifest provided by Apple.

Listing 3-3 **Apple Sample Manifest**

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
   <!-- array of downloads. -->
   <key>items</key>
   <array>
       <dict>
           <!-- an array of assets to download -->
           <key>assets</key>
           <array>
               <!-- software-package: the ipa to install. -->
               <dict>
```

```
            <!-- required.  the asset kind. -->
            <key>kind</key>
            <string>software-package</string>
            <!-- optional.  md5 every n bytes.  -->
            <!-- will restart a chunk if md5 fails. -->
            <key>md5-size</key>
            <integer>10485760</integer>
            <!-- optional.  array of md5 hashes -->
            <key>md5s</key>
            <array>
                <string>41fa64bb7a7cae5a46bfb45821ac8bba</string>
                <string>51fa64bb7a7cae5a46bfb45821ac8bba</string>
            </array>
            <!-- required.  the URL of the file to download. -->
            <key>url</key>
            <string>http://www.example.com/apps/foo.ipa</string>
        </dict>
        <!-- display-image: the icon to display during download. -->
        <dict>
            <key>kind</key>
            <string>display-image</string>
            <!-- optional. icon needs shine effect applied. -->
            <key>needs-shine</key>
            <true/>
            <key>url</key>
            <string>http://www.example.com/image.57x57.png</string>
        </dict>
        <!-- full-size-image: the large 512x512 icon used by iTunes. -->
        <dict>
            <key>kind</key>
            <string>full-size-image</string>
            <!-- optional.  one md5 hash for the entire file. -->
            <key>md5</key>
            <string>61fa64bb7a7cae5a46bfb45821ac8bba</string>
            <key>needs-shine</key>
            <true/>
            <key>url</key>
            <string>http://www.example.com/image.512x512.jpg</string>
        </dict>
    </array><key>metadata</key>
    <dict>
        <!-- required -->
        <key>bundle-identifier</key>
        <string>com.example.fooapp</string>
        <!-- optional (software only) -->
        <key>bundle-version</key>
        <string>1.0</string>
```

```
                <!-- required.  the download kind. -->
                <key>kind</key>
                <string>software</string>
                <!-- optional. displayed during download; -->
                <!-- typically company name -->
                <key>subtitle</key>
                <string>Apple</string>
                <!-- required.  the title to display during the download. -->
                <key>title</key>
                <string>Example Corporate App</string>
            </dict>
        </dict>
    </array>
</dict>
</plist>
```

## Submitting to the App Store

To build your application in compliance with the App Store's submission policies, it must be signed by a valid distribution provision profile using an active developer identity. The steps are very close to those you used to create an ad hoc distribution, but you have additional bookkeeping that you must perform.

Make sure you have at least one 512×512 PNG image on hand as well as at least one screenshot for the next few steps. If you're still not ready with final art, you can upload placeholders and replace or delete them later as needed.

Start by visiting the iOS portal and register your application identifier in the App IDs tab of the portal. This takes just a second or two and requires a common name (for example, "Collage") and the identifier (for example, com.sadun.Collage). This identifier should exactly match the one you use in Xcode. Even though you usually sign your app with a general wildcard provision, the application identifier you need for iTunes must be specific.

Head over to iTunes Connect (iTunesConnect.apple.com). Choose Manage Your Applications > Add New App > iOS App. Enter the application name (it must be unique), the SKU number (it's up to you how you define this, but it must be unique to your account), and select your new identifier from the Bundle ID pop-up. Be as exact as possible on this screen and do not use placeholders during this step. Make *very sure* you select the proper identifier. Once set, it cannot be changed.

Enter all your metadata—it can all be placeholders here—and set your two images. iTunes Connect creates your new application page and adds a "Ready to Upload Binary" button. When you click that button and declare your export encryption compliance, you're given instructions on how to upload your binary. Your application state changes from Ready to Upload to Waiting for Upload. Click Continue.

When you use placeholders in your iPhone metadata, you now have as much time as you need to edit that material. Just be sure you get your descriptions and art into shape

*before* uploading your application from Xcode. Your upload kicks off the App Store review process. Do not waste Apple's app reviewers' time by uploading your app until it's ready to be reviewed.

Finally, make sure you read Apple's App Store submission guide, which walks you through the process and its guidelines, which help explain what apps are and are not suitable for the App Store.

> **Note**
>
> Take special note of the Review Notes area in the Application Metadata. This is your single line of human communication with Apple reviewers. Use this section to explain any areas of confusion you think the reviewer may encounter. Offer how-to procedures if needed and sample account credentials for testing your application.

When you are absolutely sure you're ready to submit your application, follow these steps:

1. Review your iTunes Connect metadata for clarity and correctness.
2. Archive your application (Product > Archive).
3. In the Organizer, click Submit.
4. Enter your developer login credentials and click Next.
5. Select your application from the top pop-up (see Figure 3-28). Only applications that are "Waiting for Upload" appear in this list.
6. Select your distribution identity from bottom pop-up. *Make sure it is your general distribution identity and not an ad hoc distribution one!* This is a common mistake that usually results in an e-mail from iTunes asking you to re-submit your application after it fails the code-signing check. Click Next.
7. Wait as Xcode validates and submits your application. When it is finished, the Status field next to your application name updates to match the status of your submission. Wait for an e-mail from iTunes saying that your application has changed state to Waiting for Review.

If you encounter any trouble uploading, you may want to clean your project and recompile from scratch. If you are unable to verify or submit your application, try re-installing Xcode. Sometimes when you install beta versions of the SDK on the same machine as your primary development Xcode SDK, those versions can overwrite the tools you need to properly submit your applications to iTunes Connect.

> **Note**
>
> Keep on top of your calendar. You must renew your developer membership yearly. Your certificates all have expiration dates as well. When renewing your developer and distribution certificates, you must reissue all your mobile provisions. Throw away the old ones and create new ones with your updated developer identity. Make sure to remove the outdated certificates from your keychain when replacing them with the new ones.

Figure 3-28 Only apps that are "Waiting for Upload" are listed in the Organizer's submission screen. Make sure your identity is not set to an ad hoc profile, also listed as iPhone Distribution. It's a common error that lots of developers make.

## Summary

This chapter covered a lot of ground. From start to finish, you saw how to create, compile, and debug Xcode projects. You were introduced to most of the major Xcode components you'll use on a day-to-day basis, and you read about many of the different ways you can produce and run iPhone projects. Here are some thoughts to take away from this chapter:

- Although Xcode provides easy-to-use templates, think of them as a jumping-off point, not an endpoint. You can customize and edit projects however you want.
- Interface Builder makes it really easy to lay out views. Although, technically, you're producing the same method calls and property assignments as if you'd designed by hand, IB's elegant GUI transforms those design tasks into the visual domain, which is a welcome place for many developers.
- Learning to navigate through Xcode's in-program reference documentation is an essential part of becoming an iPhone developer. No one can keep all that information in his or her head. The more you master the documentation interface, the better you'll be at finding the class, method, or property you need to move forward.

- Everything changes. Subscribe to iOS documentation in Xcode and ensure that your documentation remains as up to date as possible.
- Xcode's built-in debugger and Instruments tools help you fix bugs faster than trying to figure out everything by hand. The tools may seem complex at first but are well worth mastering for day-to-day development.
- Get to know and love the Organizer pane. It gives you critical feedback for knowing which devices are connected and what state they are in. And the other tools—including project archives with submission to the App Store, the documentation interface, and the screenshot utility—just add to its power.
- Xcode 4's massive update has more power and capabilities than this brief introduction can cover. Other features are discussed in Chapter 4, "Designing Interfaces," which provides a more detailed overview of Interface Builder.

*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**
**AllTouchEvents 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**
**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, 153-157**
**__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
popovers, 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**
**CTFramesetterSuggestFrameSizeWith Constraints 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
**fetchedObjects property, 619**
**fetchedResultsController 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:shouldRecognize SimultaneouslyWithGestureRecognizer: 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**
**imageNamed: 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, 230–233
- 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, xxviii**
**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**
**kSCNetworkReachabilityFlagsConnection OnTraffic 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**

**MKMapViews, 193**

**mobile provisions. *See* provisioning profiles, 22-23, 168**

**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**
**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
UINavigationItem 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
NSNumbers, 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
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
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

**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**
**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**
**TouchDownRepeat 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**

**UIActionSheet, 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**

**UIAlertViewStylePlainTextInput, 636**

**UIAlertViewStyleSecureTextInput, 636**

**UIAppFonts, 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**

**UINavigationBar, 195, 259**

**UINavigationController, 41, 197-198, 247-252.** ***See also*** **navigation controllers**

**UINavigationItem, 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**
**UITextFields, 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_unretailed 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**
**userInterfaceIdiom 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
UINavigationItem 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
UITabBarControllers, 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), xxix**
**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*