Erica Sadun

Expanded Electronic Edition

Includes 9 New Chapters

The iOS 5 Developer's Cookbook

Essentials and Advanced Recipes for iOS Programmers

Developer's Library



From the Library of Eric Berna

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.

From the Library of Eric Berna

"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

From the Library of Eric Berna

The iOS 5 Developer's Cookbook: Expanded Electronic Edition

Essentials and Advanced Recipes for iOS Programmers

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

From the Library of Eric Berna

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

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: 978-0-133-02833-1
ISBN-10: 0-133-02833-X

Editor-in-Chief
Mark Taub

Senior Acquisitions Editor
Trina MacDonald

Senior Development Editor
Chris Zahn

Managing Editor
Kristy Hart

Project Editor
Anne Goebel

Copy Editor
Bart Reed

Indexer
Erika Millen

Proofreaders
Sheri Cain
Linda Seifert

Technical Reviewers
Jon Bauer
Tim Burks
Joachim Bean
Matt Martel

Editorial Assistant
Olivia Basegio

Cover Designer
Gary Adair

Composition
Nonie Ratcliff

From the Library of Eric Berna

❖

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.

❖

From the Library of Eric Berna

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**

16 Documents and Data Sharing **745**

17 Connecting to the Address Book **785**

18 iCloud Basics **827**

19 Core Location and MapKit **867**

20 Audio, Video, and MediaKit **899**

21 Push Notifications **953**

22 Accessibility **985**

23 StoreKit: In-App Purchasing **995**

24 GameKit **1019**

From the Library of Eric Berna

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

From the Library of Eric Berna

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

From the Library of Eric Berna

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

From the Library of Eric Berna

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

From the Library of Eric Berna

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

From the Library of Eric Berna

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

From the Library of Eric Berna

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

From the Library of Eric Berna

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

From the Library of Eric Berna

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

From the Library of Eric Berna

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

From the Library of Eric Berna

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

From the Library of Eric Berna

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

From the Library of Eric Berna

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

From the Library of Eric Berna

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

From the Library of Eric Berna

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

From the Library of Eric Berna

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

From the Library of Eric Berna

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

From the Library of Eric Berna

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

16 Documents and Data Sharing 745

Working with Uniform Type Identifiers 745

Determining UTIs from File Extensions 746

Moving From UTI to Extension or MIME Type 747

Testing Conformance 748

Retrieving Conformance Lists 748

Accessing the System Pasteboard 750

Storing Data 751

Storing Common Types 751

Retrieving Data 752

From the Library of Eric Berna

Recipe: Passively Updating the Pasteboard 752
Recipe: Enabling Document File Sharing 754
Recipe: Monitoring the Documents Folder 757
The Document Interaction Controller 760
Creating Document Interaction Controller Instances 761
Document Interaction Controller Properties 763
Providing Document Quick Look Support 764
Providing Printer and Copy Support 764
Recipe: Checking for the Open Menu 766
Declaring Document Support 767
Recipe: Implementing Document Support 768
Exported Type Declarations 772
Creating URL-Based Services 773
Using URL Schemes 773
Service Downsides 773
Cross-Promotion 774
Registering Schemes: Declaring the URL 774
Registering Schemes: Adding the Handler Method 775
Returning Control to a Calling Application 776
Recipe: Adding Custom Settings Bundles—Or Not 776
The Settings App 777
Avoid Sensitive Information 778
Settings Schema 778
Defining a Settings Bundle 779
Settings and Users 782
Checking User Defaults 783
Summary 783

17 Connecting to the Address Book 785
The AddressBook Frameworks 785
AddressBookUI 785
AddressBook and Its Databases 786
Records 786
The Custom ABStandin Class 787

From the Library of Eric Berna

Querying the Address Book 788
Wrapping the AddressBook Framework 788
Working with Record Functions 789
Retrieving and Setting Strings 790
Working with Date Properties 792
Multivalue Record Properties 793
Storing Multivalue Data 796
Working with Multivalue Items 797
Addresses, Social Profile, and Instant-Message Properties 798
Address Book Images 800
Creating, Adding, and Deleting Records 801
Searching for Contacts 803
Sorting Contacts 804
Working with Groups 804
ABContact, ABGroup, and ABContactsHelper 807
Recipe: Searching the Address Book 807
Recipe: Accessing Contact Image Data 810
Recipe: Picking People 811
Recipe: Limiting Contact Picker Properties 814
Recipe: Adding and Removing Contacts 816
Modifying and Viewing Individual Contacts 819
User Edits 819
Delegate Methods 819
Recipe: The "Unknown" Person Controller 820
One More Thing: Adding Random Contact Art 823
Summary 826

18 iCloud Basics 827
Setting Up an iCloud-Compatible Project 827
Determining Your Team Identifier Prefix 828
Containers 829
Retrieving Identifiers in Code 830
How iCloud Works 831
iCloud Container Folders 831
Building a Shared Document Home 832
Writing To and Reading From Ubiquitous Storage 833

From the Library of Eric Berna

Editing iCloud Storage 834
Designing for iCloud Ubiquity 834
Recipe: Trying Out iCloud 835
Working with UIDocument 838
Establishing the Document as a File Presenter 839
Moving to the Cloud 839
Evicting Items from the Cloud 840
Opening and Saving Files 841
Subscribing to UIDocument Notifications 842
Recipe: Subclassing UIDocument 843
Metadata Queries and the Cloud 845
Handy Routines 847
Local URLs 847
Locating the Ubiquitous Documents Folder 847
Retrieving Files 848
Setting Ubiquity 849
Deleting Files 851
Retrieving Modification Dates 853
Recipe: Accessing the Ubiquitous Key-Value Store 853
How Key-Value Stores Work 854
Accessing the Key-Value Store 854
Subscribing to Key-Value Notifications 855
Recipe: UIManagedDocument and Core Data 856
Remove All Context Saves 857
Establish Your Identifiers and URLs 857
Establish the Document 858
Opening the Document 858
Start Observing 859
Summary 864

19 Core Location and MapKit 867
How Core Location Works 867
GPS Positioning 867
Wi-Fi Positioning 868
Cell Tower Positioning 868
Internet Provider Positioning 868
Hybridizing the Approaches 869

From the Library of Eric Berna

Testing Core Location Features 869
Recipe: Core Location in a Nutshell 870
Location Properties 873
Region Detection 874
Recipe: Tracking Speed 874
Recipe: Computing Speed and Distance 876
Recipe: Keeping Track of "North" by Using Heading Values 878
Recipe: Forward and Reverse Geocoding 881
Recipe: Viewing a Location 883
Finding the Best Location Match 885
Recipe: User Location Annotations 888
Recipe: Creating Map Annotations 890
Creating, Adding, and Removing Annotations 890
Annotation Views 891
Customizing Annotation Views 891
Responding to Annotation Button Taps 893
Summary 897

20 Audio, Video, and MediaKit 899
Recipe: Playing Audio with AVAudioPlayer 899
Initializing an Audio Player 899
Monitoring Audio Levels 901
Playback Progress and Scrubbing 902
Catching the End of Playback 903
Recipe: Looping Audio 908
Recipe: Handling Audio Interruptions 911
Audio That Ignores Sleep 913
Recipe: Adding Background Audio 915
Finishing Up Processing 918
Recipe: Recording Audio 919
Recipe: Recording Audio with Audio Queues 925
Recipe: Playing Video with the Media Player 930
Recipe: Recording Video 934
Recipe: Picking and Editing Video 938

From the Library of Eric Berna

Recipe: Picking Audio with the MPMediaPickerController 941
Creating a Media Query 944
Building a Query 944
Using Predicates 945
Recipe: Using the MPMusicPlayerController 946
Summary 950

21 Push Notifications 953
Introducing Push Notifications 953
How Push Works 954
Multiple Provider Support 955
Security 956
Push Limitations 956
Push Notifications Versus Local Notifications 957
Provisioning Push 957
Generate a New Application Identifier 957
Generate Your SSL Certificate 958
Push-Specific Provisions 961
Creating a Push-Compatible Application 962
Registering Your Application 962
Retrieving the Device Token 963
Handling Token Request Errors 963
Responding to Notifications 964
Recipe: Push Client Skeleton 966
Building Notification Payloads 972
Localized Alerts 973
Transforming from Dictionary to JSON 973
Custom Data 974
Receiving Data on Launch 974
Recipe: Sending Notifications 975
Sandbox and Production 976
Feedback Service 980
Designing for Push 981
Summary 982

From the Library of Eric Berna

22 Accessibility 985

Accessibility in Interface Builder 985

Labels 986

Hints 987

Enabling Accessibility 988

Traits 988

Working with Accessibility from Code 988

Testing with the Simulator 989

Broadcasting Updates 991

Testing Accessibility on the iPhone 991

Summary 994

23 StoreKit: In-App Purchasing 995

Getting Started with StoreKit 995

Fulfillment 996

The StoreKit Development Paradox 996

Developing and Testing 998

Submission 998

Creating Test Accounts 998

Creating New In-App Purchase Items 1000

Filling Out the Details Section 1000

Adding Localized Descriptions 1001

Filling Out the Pricing Section 1003

Submitting a Purchase GUI Screenshot 1003

Submitting In-App Purchases Products for Review 1004

Building a Storefront GUI 1004

Purchasing Items 1007

Signing Out of Your iTunes Account for Testing 1008

Regaining Programmatic Control After a Purchase 1008

Registering Purchases 1010

Restoring Purchases 1012

Purchasing Multiple Items 1013

Handling Delays in Registering Purchases 1013

Validating Receipts 1013

Summary 1016

From the Library of Eric Berna

24 GameKit 1019

Enabling Game Center 1019
Testing for Game Center Availability 1020
Testing Game Center Compliance 1020
Designing Leaderboards and Achievements 1022
Building Leaderboards 1022
Creating Achievements 1023
Accessing Leaderboards 1024
The GKLeaderboard Class 1025
Leaderboard View Controller 1026
Submitting Scores 1028
Unlocking Achievements 1028
Checking Achievements 1029
Reporting Achievements (Unlocking) 1030
Resetting Achievements 1031
Matchmaking: Multiplayer GameKit Matches 1032
Matchmaker Required Methods 1033
Starting a Match 1033
Starting the Game 1034
Creating an Invitation Handler 1034
Managing Match State 1036
Player State Changes 1037
Retrieving Player Names 1037
Game Play 1038
Serializing Data 1039
Synchronizing Data 1042
Turn-by-Turn Matches 1044
Starting a Match 1044
View Controller Delegate Methods 1045
Match Found 1048
Ending the Match 1050
Force Quitting Matches 1052
Game Center Voice 1053
Testing for Chat Availability 1053
Establishing a Play and Record Audio Session 1053
Creating a Voice Chat 1054

From the Library of Eric Berna

Starting and Stopping a Chat 1054
Chat State Monitoring 1054
Implementing a Chat Button 1055
Controlling Volume 1056
GameKit Peer Services 1057
GameKit Bluetooth Limitations 1057
Bonjour Sessions 1058
Servers, Clients, Peers 1059
The Peer Connection Process 1059
Sending and Receiving Data 1063
State Changes 1063
Creating a GameKit Helper 1064
Peer-to-Peer Voice Chat 1064
Implementing Voice Chat 1064
Creating an “Online” GameKit Connection 1066
Summary 1067

From the Library of Eric Berna

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.

From the Library of Eric Berna

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.

From the Library of Eric Berna

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!

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.

From the Library of Eric Berna

- **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.

From the Library of Eric Berna

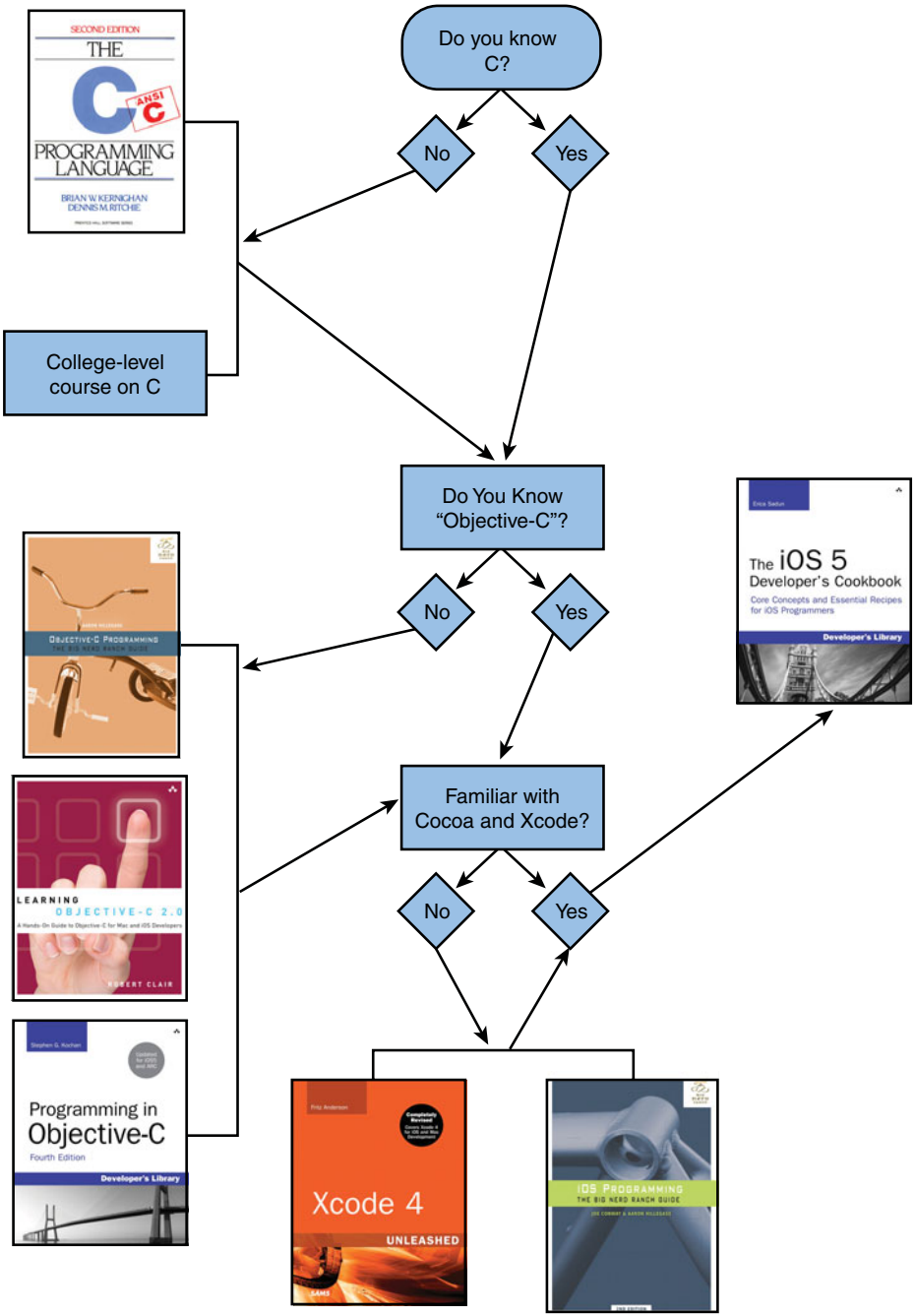


Figure P-1 What it takes to be an iOS programmer.

From the Library of Eric Berna

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.

> **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.

From the Library of Eric Berna

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

From the Library of Eric Berna

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.

From the Library of Eric Berna

- **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.
- **Chapter 16, "Documents and Data Sharing"**—iOS applications can share information and data as well as move control from one application to another using a variety of system features. Chapter 16 introduces the technologies that support those features from the pasteboard to custom URL schemes, and beyond. You'll discover how to integrate these features into your applications and use them smartly to make your app a cooperative citizen of the iOS ecosystem.
- **Chapter 17, "Connecting to the Address Book"**—The iPhone's Address Book frameworks allow you to programmatically access and manage the contacts database. Chapter 17 introduces the Address Book and demonstrates how to use its frameworks in your applications. You read about accessing information on a contact-by-contact basis, how to modify and update contact information, and how to use predicates to find just the contact you're interested in. This chapter also covers the GUI classes that provide interactive solutions for picking, viewing, and modifying contacts.
- **Chapter 18, "iCloud Basics"**—iCloud lets you share data beyond the limitations of individual hardware and into a central data storage hub. Chapter 18 introduces iCloud, the technology that allows your applications to access and update shared documents without explicit synchronization. This chapter introduces iCloud, shows you how it works, introduces the key classes you'll use, and walks you through several recipes and code snippets showing you how to use it in your own projects.

From the Library of Eric Berna

- **Chapter 19, "Core Location and MapKit"**—Core Location infuses the iPhone with on-demand geopositioning based on a variety of technologies and sources. MapKit adds interactive in-application mapping, allowing users to view and manipulate annotated maps. With Core Location and MapKit, you can develop applications that help users meet up with friends, search for local resources, or provide location-based streams of personal information. Chapter 19 introduces these location-aware frameworks and shows you how you can integrate them into your iPhone applications.
- **Chapter 20, "Audio, Video, and MediaKit"**—The iPhone is a media master; its built-in iPod features expertly handle both audio and video. The iPhone SDK exposes that functionality to developers. A rich suite of classes simplifies media handling via playback, search, and recording. Chapter 20 introduces recipes that use these classes, presenting media to your users and letting your users interact with that media. You see how to build audio and video players as well as audio and video recorders. You discover how to browse the iPod library and how to choose what items to play.
- **Chapter 21, "Push Notifications"**—When developers need to communicate directly with users, push notifications provide the solution. They deliver messages directly to the iPhone screen via a special Apple service. Push notifications let the iPhone display an alert, play a custom sound, or update an application badge. In this way, off-phone services connect with an iPhone-based client, letting them know about new data or updates. Chapter 21 introduces push notifications. In this chapter, you learn how push notifications work and dive into the details needed to create your own push-based system.
- **Chapter 22, "Accessibility"**—Chapter 22 offers a brief overview of VoiceOver accessibility. You read about adding accessibility labels and hints to your applications and testing those features in the simulator and on the iOS device. Accessibility is available and can be tested on third generation or later devices, including all iPads, the iPhone 3GS and later, and the third generation iPod touch and later.
- **Chapter 23, "StoreKit: In-App Purchasing"**—StoreKit offers in-app purchasing that integrates into your software. Chapter 23 introduces StoreKit and shows you how to use the StoreKit API to create purchasing options for users. In this chapter, you read about getting started with StoreKit. You learn how to set up products at iTunes Connect and localize their descriptions. And you see what it takes to create test users and how to work your way through various development/deployment hurdles. This chapter teaches you how to solicit purchase requests from users and how to hand over those requests to the store for payment. This chapter covers the entire StoreKit picture, from product creation to sales.
- **Chapter 24, "GameKit"**—Chapter 24 introduces GameKit, a feature that allows your applications to move beyond a single player/single device scenario using both Game Center and device-to-device networking. Game Center adds a centralized

From the Library of Eric Berna

service that allows your game to offer shared leader boards and internet-based matches. GameKit also provides an ad hoc networking solution for peer-to-peer connectivity. Built on a technology called Bonjour, ad hoc connection offers simple, no-configuration communications between devices. In this chapter, you discover how to use GameKit to build connected applications. You see how to create Game Center features. You read about adding GameKit Voice to your code for walkie-talkie–style voice chats.

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, ensur-

From the Library of Eric Berna

ing 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.

From the Library of Eric Berna

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.

From the Library of Eric Berna

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

From the Library of Eric Berna

1

Introducing the iOS SDK

The iOS family includes the iPhone, the iPad, and the iPod touch. These devices offer innovative mobile platforms that are a joy to program. They are the founding members of Apple's family of pocket-based computing devices. Despite their relatively diminutive proportions compared to desktop systems, they run a first-class version of OS X with a rich and varied SDK that enables you to design, implement, and realize a wide range of applications. For your projects, you can take advantage of iOS's multitouch interface and powerful onboard features using Xcode, Apple's integrated design environment. In this chapter, you discover the components of the SDK and explore the product it creates: the iPhone application. You learn about Apple's various iPhone developer programs and how you can join. You explore the iPhone application-design philosophy and see how applications are put together. Finally, you read about setting up your program credentials so you can put that philosophy to use and start programming.

iOS Developer Programs

Are you ready to start programming for iOS? Ready to see what all the fuss is about? Apple's iOS software development kit (SDK) is readily available to members of Apple's iPhone developer programs. There are four. These programs include the free online program, the paid enterprise program for in-house development, the paid standard program that allows developers to submit their products to the App Store, and a special university program (see Table 1-1).

Table 1-1 **iOS Developer Programs**

Program	Cost	Audience
Online Developer Program	Free	Anyone interested in exploring the iOS SDK without commitment
iOS Developer Program	$99/Year	Developers who want to distribute through the App Store
iOS Developer Enterprise Program	$299/Year	Large companies building proprietary software for employees

From the Library of Eric Berna

Table 1-1 **iOS Developer Programs**

Program	Cost	Audience
iOS Developer University Program	Free	Free program for higher education institutions that provide an iPhone development curriculum

Each program offers access to the iOS SDK, which provides ways to build and deploy your applications. The audience for each program is specific.

Online Developer Program

The free program is meant for anyone who wants to explore the full iOS SDK programming environment but who isn't ready to pay for further privileges. The free program limits your deployment options to the Mac simulator. Although you can run your applications in the simulator, you cannot install those applications to the device or sell them in the App Store.

Although each version of the simulator moves closer to representing actual device performance, you should not rely on it for evaluating your application. An app that runs rock solid on the simulator may be unresponsive or even cause crashes on the actual device. The simulator does not, for example, support vibration or accelerometer readings. These and other features present on the device are not always available in the simulator. A discussion about the simulator and its restrictions follows later in this chapter in the section "Simulator Limitations."

You can now download Xcode free from the Mac App Store without joining the free online program, although joining the program allows you to receive helpful e-mail updates.

Standard Developer Program

To receive device and distribution privileges, you must pay a $99/year program fee for the standard iOS Developer Program. You can join as an individual or as a company. Once you have paid, you gain access to the App Store distribution and can test your software on actual iPhone hardware. This program adds ad hoc distribution as well, allowing you to distribute prerelease versions of your application to up to 100 registered devices. The standard program provides the most general solution for the majority of iOS programmers who want to be in the App Store. If you intend to conduct business through selling applications, this is the program to sign up for.

The standard iOS Developer Program also offers early access to beta versions of the SDK. This is a huge advantage for developers who need to prepare products for market in a timely manner and to match Apple's firmware upgrade dates.

From the Library of Eric Berna

> **Note**
>
> In early 2010, Apple restructured its Macintosh Developer Program to match the success of the iPhone Developer Program. Costing $99 per year, the restructured Mac program offers the same kind of resources as the iPhone program—namely code-level technical support incidents, developer forum membership, and access to prerelease software. Neither program offers hardware discounts. The Mac Developer Program does not give access to iOS software, and vice versa.

Developer Enterprise Program

The $299/year Enterprise Program is meant for in-house application distribution. It's targeted at companies with 500 employees or more. Enterprise memberships do not offer access to the App Store. Instead, you can build your own proprietary applications and distribute them to your employees' hardware through a private storefront. The Enterprise Program is aimed at large companies that want to deploy custom applications to their employees, such as ordering systems.

Developer University Program

Available only to higher education institutions, the Developer University Program is a free program aimed at encouraging universities and colleges to form an iPhone development curriculum. The program allows professors and instructors to create teams with up to 200 students, offering them access to the full iOS SDK. Students can share their applications with each other and their teachers, and the institution itself can submit applications to the App Store.

Registering

Register for the free or paid program at the main Apple developer site at http://developer.apple.com/programs/register.

Getting Started

Regardless of which program you sign up for, you must have access to an Intel-based Mac running a current version of Mac OS X. It also helps to have at least one—and preferably several—iPhone, iPad, and iPod touch units to test on to ensure that your applications work properly on each platform, including legacy units.

Often, delays are associated with signing up for paid programs. After registering, it can take time for account approval and invoicing. Once you actually hand over your money, it may take another 24 to 72 hours for your access to advanced portal features to go live.

From the Library of Eric Berna

Registering for iTunes Connect, so you can sell your application through the App Store, offers a separate hurdle. Fortunately, this is a process you can delay until after you've finished signing up for a paid program. With iTunes Connect, you must collect banking information and incorporation paperwork prior to setting up your App Store account. You must also review and agree to Apple's distribution contracts. Apple offers full details at itunesconnect.apple.com.

> **Note**
>
> Make sure to fill out the extra forms for Japan; otherwise, 20% of your in-country sales will be withheld.

Downloading the SDK

Download your copy of the iOS SDK from the main iOS developer site at http://developer.apple.com/ios. You must use your program credentials to access the download page, so be sure you've signed up for a developer program before attempting to download. The free program offers access only to fully released SDKs. The paid program adds early looks at SDK betas, letting you develop to prerelease firmware.

The kit, which typically runs a few gigabytes in size, installs a complete suite of interactive design tools onto your Macintosh. This suite consists of components that form the basis of the iOS development environment. iOS-specific components include the following software:

- **Xcode**—Xcode is the most important tool in the iOS development arsenal. It provides a comprehensive project development and management environment, complete with source editing, comprehensive documentation, and a graphical debugger. Xcode has recently been updated to version 4.2, which is distributed as part of the iOS 5.x SDK, offering a major overhaul to a well-loved product. Xcode now offers the LLVM 3.0 compiler and has recently introduced a major Objective-C language upgrade with the introduction of automatic reference counting (ARC) as well as support for C++0x, the planned new standard for the C++ programming language.
- **Interface Builder**—Interface Builder (IB) provides a rapid prototyping tool for laying out user interfaces graphically and linking to those prebuilt interfaces from your Xcode source code. With IB, you place out your interface using visual design tools and then connect those onscreen elements to objects and method calls in your application. With Xcode 4, IB has been integrated into the primary Xcode IDE rather than working as a standalone application, as it had throughout earlier versions of the SDK. With the iOS 5.*x* SDK, IB introduces Storyboarding. This new feature allows you to lay out all your application screens at once, defining the ways each screen moves to the next.
- **Simulator**—The iOS simulator runs on the Macintosh and enables you to create and test applications on your desktop. You can test programs without connecting to an actual iPhone, iPad, or iPod touch. The simulator offers the same API used on iOS devices and provides a preview of how your concept designs will look. When

From the Library of Eric Berna

working with the simulator, Xcode compiles Intel x86 code that runs natively on the Macintosh rather than ARM-based code used on the iPhone.

- **Instruments**—Instruments profiles how iPhone applications work under the hood. It samples memory usage and monitors performance. This lets you identify and target problem areas in your applications and work on their efficiency. Instruments offers graphical time-based performance plots that show where your applications are using the most resources. Instruments is built around the open-source DTrace package developed by Sun Microsystems. Instruments plays a critical role in making sure your applications run efficiently on the iPhone platform.
- **Shark**—Shark provides performance tuning by analyzing where an application spends most of its time. It locates and identifies bottlenecks, enabling you to speed your application performance. As of Xcode 4, Shark's performance analysis has been rolled into Instruments.

Together, the components of this iOS SDK suite enable you to develop your applications. From a native application developer's point of view, the most important components are Xcode, Interface Builder, and the Simulator, with Instruments providing an essential tuning tool. In addition to these tools, there's an important piece not on this list. This piece ships with the SDK but is easy to overlook. I refer to Cocoa Touch.

Cocoa Touch is the library of classes provided by Apple for rapid iOS application development. Cocoa Touch, which takes the form of a number of API frameworks, enables you to build graphical event-driven applications using user interface elements such as windows, text, and tables. Cocoa Touch on iOS is analogous to Cocoa and AppKit on Mac OS X and supports creating rich, reusable interfaces on the iPhone.

Many developers are surprised by the code base size of iPhone applications; they're tiny. Cocoa Touch's library support is the big reason for this. By letting Cocoa Touch handle all the heavy UI lifting, your applications can focus on getting their individual tasks done. The result is compact, focused code that does a single job at a time.

Using Cocoa Touch lets you build applications with a polished look and feel, consistent with those developed by Apple. Remember that Apple must approve your software. Apple judges applications on the basis of appearance, operation, and even content. Using Cocoa Touch helps you better approximate the high design standards set by Apple's native applications.

Development Devices

A physical iPhone, iPad, or iPod touch provides a key component of the software development kit. Testing on the device is vital. As simple and convenient as the SDK simulator is, it falls far short of the mark when it comes to a complete iOS testing experience. Given that the iOS device family is the target platform, it's important that your software runs its best on its native system rather than on the simulator. The iOS device itself offers the fully caffeinated, un-watered-down testing platform you need.

Apple regularly suggests that the development unit needs to be devoted exclusively to development. Reality has proven more hit and miss on that point. Other than early betas,

From the Library of Eric Berna

firmware has proven stable enough that you can use your devices for both development and day-to-day tasks, including making calls on iPhones.

According to Apple disclaimers, using a device as a development unit means that it is subject to onboard data changes and might no longer work reliably as a field unit. Experience shows that once you're past those unstable early betas of new SDKs, the devices seem to hold up fine for regular day-to-day use. It's still best to have extra units on hand devoted solely to development, but if you're short on available units, you can probably use your main iPhone for development; just be aware of the risks, however small.

Devices must be proactively set up for development use with Xcode's organizer. The organizer also lets you register your device with Apple, without having to enter its information by hand at the provisioning portal.

When developing, it's important to test on as many iOS platforms as possible. Be aware that real platform differences exist between each model of iPhone, iPad, and iPod touch. For example, the fourth-generation iPod offers built-in cameras; the third generation and earlier did not. The second-generation iPad uses a faster processor than the first generation. iPhones all have cameras, which none of the iPod touches offered until the fourth generation. Certain models of the iPad and the iPhone offer GPS technology; other models do not. A discussion of model-specific differences follows later in this chapter.

> **Note**
>
> iOS developers do not receive hardware discounts for development devices. You must pay the full nonsubsidized price for extra iOS devices purchased without service.

Simulator Limitations

Each release of the Macintosh-based iPhone simulator continues to improve on previous technology. That having been said, there are real limitations you must take into account. From software compatibility to hardware, the simulator approximates but does not equal actual device performance.

The simulator uses many Macintosh frameworks and libraries, offering features that are not actually present on the iPhone or other iOS devices. Applications that appear to be completely operational and fully debugged on the simulator may flake out or crash on the device itself due to memory or performance limitations on iOS hardware. Instruction set differences may cause apps to crash on older devices when they are built to support only newer versions of the ARM architecture. You simply cannot fully debug any program solely by using the simulator and be assured that the software will run bug-free on iOS devices.

The simulator is also missing many hardware features. You cannot use the simulator to test the onboard camera, or accelerometer and gyro feedback. Although the simulator can read acceleration data from your Macintosh using its sudden motion sensor if there's one onboard (usually for laptops), the readings will differ from iOS device readings and are not practical for development or testing. The simulator does not vibrate or offer multitouch input (at least not beyond a standard "pinch" gesture).

From the Library of Eric Berna

> **Note**
>
> The open-source accelerometer-simulator project at Google Code (http://code.google.com/p/accelerometer-simulator/) offers an iPhone application for sending accelerometer data to your simulator-based applications, allowing you to develop and debug applications that would otherwise require accelerometer input. A similar commercial product called iSimulate is available on the App Store for purchase.

From a software point of view, the basic keychain security system is not available on the simulator. You cannot register an application to receive push notification either. These missing elements mean that certain kinds of programs can only be properly used when deployed to an iPhone or other iOS device.

Another difference between the simulator and the device is the audio system. The audio session structure is not implemented on the simulator, hiding the complexity of making things work properly on the device. Even in areas where the simulator does emulate the iOS APIs, you may find behavioral differences because the simulator is based on the Mac OS X Cocoa frameworks. Sometimes you have the opposite problem. Some calls do not appear to work on the simulator but work correctly on the device.

That's not to say that the simulator does not play an important testing role. It's quick and easy to try out a program on the simulator, typically much faster than transferring a compiled application to an iOS unit. The simulator lets you rotate your virtual device to test reorientation, produce simulated memory warnings, and try out your UI as if your user were receiving a phone call. It's much easier to test out text processing on the simulator because you can use your desktop keyboard rather than hook up an external Bluetooth keyboard to your system *and* you can copy and paste text from local files; this simplifies repeated text entry tasks such as entering account names and passwords for applications that connect to the Net.

In the end, the simulator offers compromise. You gain a lot of testing convenience but not so much that you can bypass actual device testing.

> **Note**
>
> The simulator supports Video Out emulation. There's no actual Video Out produced but the simulated device responds as if you've added a compliant cable to its (nonexistent) connector. You can view the "external" video in a floating simulator window.

Tethering

Apple is moving away from tethered requirements in iOS, but has not introduced a way to develop untethered at the time this book is being written. Until now, all interactive testing has been done using a USB cable. Apple provided no way to transfer, debug, or monitor applications wirelessly as you develop. That meant you performed nearly all your work tethered over a standard iPhone USB cable.

From the Library of Eric Berna

The physical reality of tethered debugging can be problematic. Reasons for this include the following points:

- When you unplug the cable, you unplug all the interactive debugging, console, and screenshot features. So you need to keep that cable plugged in all the time.
- You cannot reasonably use iPhones and iPods with a dock. Sure, the dock is stable, but touching the screen while testing interfaces is extremely awkward when the iPhone is seated at a 75-degree angle. A docked iPad is much easier to test with.
- For the iPhone and iPod touch, tethers come to the bottom, not the top of the unit, meaning it's easy to catch that cable and knock your device to the floor.

Obviously, untethered testing vastly improves many of these issues. When tethered, you can Rube Goldberg-ize your iOS device to get around these problems. One solution is to attach Velcro to the back of a case—a case that leaves the bottom port connector open—and use that to stabilize your iPhone on your desk. It's ugly, but it keeps your iPhone from moving around or even getting knocked to the floor at times. You can also now purchase third-party cradles for the iPhone that help with development work. These stands hold the iPhone a few inches off the desk and keep the cable directed toward the back. The iPad with its larger form factor rarely presents any trouble when kept on a stand for debugging and offers easy access to its docking port in most orientations.

Always try to tether your unit to a port directly on your Mac for best results. If you must use a hub, connect to a powered system that supports USB 2.0. Most older keyboards and displays only provide unpowered USB 1.1 connections. When you're testing, it helps to choose a reliable, powered 2.0 port you can count on.

When it comes to the iPad, you may want to untether your device between testing periods and plug it directly to the wall using its 10W power adapter to let it fully recharge. Some USB ports provide sufficient power to charge the iPad while you're using it, but this is not a universal situation.

> **Note**
>
> When testing applications that employ Video Out, you can use the Apple-branded component and composite cables ($49 each at the Apple Store) or the HDMI digital adapter ($39 at the Apple Store, for iPad 2 and fourth-generation or later devices). These provide both Video Out and USB connections to allow you to tether while running your applications. The Apple-branded VGA cable ($29 at the Apple Store) does not offer this option. You will need to redirect any testing output to the screen or to file because you cannot tether while using VGA output. You may also want to pick up a second-generation or later Apple TV to test your application with AirPlay screen mirroring support.

Understanding Model Differences

When it comes to application development, many iOS apps never have to consider the platform on which they're being run. Most programs rely only on the display and touch input. They can be safely deployed to all the current iOS family devices; they require no special programming or concern about which platform they are running on.

From the Library of Eric Berna

There are, however, real platform differences. These differences are both significant and notable. They play a role in deciding how you tell the App Store to sell your software and how you design the software in the first place. Should you deploy your software only to the iPhone family or only to the iPad? To the iPhone, the iPad, and the second-generation and later iPod touch? Or should your application be targeted to every platform? This section covers some issues to consider.

Screen Size

The iPad family and devices that employ Retina displays use a different screen size than the older members of the iOS family. First- and second-generation iPads use a 1024-by-768-pixel resolution; the iPhone 4's Retina display screen resolution is 960-by-640 pixels. Earlier iPhones and iPod touch devices use 320-by-480-pixel displays.

With a completely revised set of human interface guidelines, developing for the iPad involves creating unified interfaces rather than the staged screen-by-screen design used by the earlier iPhone and iPod touch units with their reduced window size. Applications that rely on the greater screen scope that the iPad provides may not translate well to the smaller members of the device family.

Although newer iPhones and iPod touches provide greater pixel density and offer the enhanced Retina display that better matches human vision, their screen dimension remains at 3.5-inches diagonal. That limited geometry combined with the physical realities of the human hand and its fingers prevents these units from providing the same kind of user interaction experience possible on the iPad. The interaction guidelines for the newest units remain in lock step with the earlier members of the iPhone and iPod touch family.

Camera

Original iPhones shipped with one back-facing camera. Starting with the iPhone 4, the fourth-generation iPod touch, and iPad 2, two cameras became the norm. Earlier iPod touches and the original iPad did not offer cameras at all. These cameras are useful. You can task the camera to take shots and then send them to Flickr or Twitter. You can use the camera to grab images for direct manipulation, augmented reality, and so forth. The iOS SDK provides a built-in image picker controller that offers camera access to your users, but only on camera-ready platforms. Video services are limited to the 3GS model and later.

When building camera-ready applications, know that you cannot deploy them to older iPods and iPads. Camera services are limited to newer models or the iPhone family as a whole. The first- and second-generation iPhone's built-in 2 megapixel camera will never win awards. The third-generation 3GS camera was much improved, offering autofocus, macro photography, video recording, and better low-light sensitivity. Starting with the iPhone 4, iOS cameras really began to realize their potential, providing high-quality video and still interaction for all sorts of camera-based tasks.

From the Library of Eric Berna

Audio

First-generation iPod touches lacked the built-in speaker found on the iPhone, the iPad, and the second-generation and later iPod touch. Although the 1G touch was perfectly capable of powering third-party speakers through its bottom connector port, Apple considered those to be unauthorized accessories and their use was rare. Newer models of the iPod touch, particularly the fourth generation and later, have come into much closer feature matching with the rest of the iOS device line.

Newer devices (iPad, iPhone 3GS and later, third-generation iPod touch and later) provide a number of accessibility features, including the VoiceOver screen reader. You can build descriptions into your GUI elements to allow your applications to take advantage of VoiceOver, so your interfaces can describe themselves to visually impaired end users.

> **Note**
>
> The iPhone 4 first introduced an improved dual-microphone system that provides enhanced noise suppression for better audio input.

Telephony

It may seem an overly obvious point to make, but the iPhone's telephony system, which handles both phone calls and SMS messaging, can and will interrupt applications when the unit receives an incoming telephone call. Sure, users can suspend out of apps whenever they want on the iPhone, iPad, and iPod platforms, but only the iPhone has to deal with the kind of transition that's forced by the system and not a choice by the user.

Consider how the different kinds of interruptions might affect your application. It's important to keep all kinds of possible exits in mind when designing software. Be aware that the choice to leave your app may not always come from the user, especially on the iPhone. Applications that use audio need to take special care to restore the correct state after phone call interruptions.

Another fallout of telephony operations is that more processes end up running in the background on iPhones than on iPads and iPod touches, even those iPads which provide 3G cellular data support. This means that as a rule, the amount of free memory is likely to be slightly reduced on the iPhone compared to the touch or iPad. This is one reason that making the iPhone your primary smaller-screen development device over the iPod touch may be a smart move. Working within the iPhone's greater limitations may produce software that operates robustly on both the iPhone and touch platforms. Be aware, however, that the most recent hardware units with their A4 and A5 processors and generous RAM really aren't affected much by onboard telephony.

Core Location and Core Motion Differences

Core Location depends on three different approaches, each of which may or may not be available on a given platform. These approaches are limited by each device's onboard capabilities. Wi-Fi location, which scans for local routers and uses their MAC addresses to search a central position database, is freely available on all iPhone, iPad, and iPod touch platforms.

From the Library of Eric Berna

Cell location, however, depends on an antenna that is available only on the iPhone and possibly on the iPad, although this latter technology has not yet been confirmed. This technology triangulates from local cell towers, whose positions are well defined from their installations by telephone companies. The final and most accurate strategy, GPS location, is available only to second-generation and later iPhones and 3G iPads. GPS was not built in to the first-generation iPhone and is not currently available to any iPod touch units or on the Wi-Fi-only iPad.

The third-generation iPhone 3GS introduced a built-in compass (via a magnetometer) along with the Core Location APIs to support it. The iPhone 4 and iPad 2 added a three-axis gyro, which provides pitch, roll, and yaw feedback, all of which can be solicited via the Core Motion framework.

Vibration Support and Proximity

Vibration, which adds tactile feedback to many games, is limited to iPhones. The iPad and iPod touches do not offer vibration support. Nor do they include the proximity sensor that blanks the screen when holding an iPhone against your ear during calls. Until SDK 3.0, using the proximity sensor in your applications had been off limits; the `UIDevice` class now offers direct access to the current state of the proximity sensor.

Processor Speeds

When the second-generation iPod touch was introduced, its 532MHz processor offered the highest processing power in the iPhone family until it was supplanted by the iPhone 3GS, running at a reported 600MHz. That's nothing compared to the iPad. The iPad 2 used a custom 1GHz Apple A5 high-performance, low-power chip with a generous 512MB of onboard RAM, massively outperforming the ARM processors deployed on older systems (the iPad 1 used an A4 chip). The iPhone 4 apparently uses an underclocked A4 chip to save on power consumption. The point is this: Make sure to test your software on older, slower units as well as on the newer ones. Application response time can and will be affected by the device on which it's being run.

If your application isn't responsive enough on the older platforms, consider working up your code efficiency. There is no option in the App Store at this time that lets you omit earlier generation iPhone devices from your distribution base, although compiling to more modern firmware releases automatically disqualifies a certain number of aging units. The 5.0 iOS release, for example, supports the iPhone 3GS and newer, the iPod touch 3G and newer, and all iPad models. These are all quite able newer models.

OpenGL ES

OpenGL ES offers a royalty-free cross-platform API for 2D and 3D graphics development. It is provided as part of the iOS SDK. Not all iPhone models provide the same OpenGL ES support. The iPhone 3GS, iPads, 3G iPod touch, and newer models support both OpenGL ES 2.0 and 1.1. Earlier models, including the 2G and 3G iPhone as well as

From the Library of Eric Berna

the first- and second-generation iPod touch, ran only OpenGL ES 1.1. The 2.0 API provides better shading and text support, providing higher quality graphics.

To target all iPhones, develop your graphics using only 1.1. Applications leveraging the 2.0 API are limited to more current models but, in practical terms, cover the majority of your modern deployment audience. OpenGL ES 2.0 is significantly more complex to program but well worth it.

Platform Limitations

When talking about mobile platforms such as the iPhone, several concerns always arise, such as storage, interaction limits, and battery life. Mobile platforms can't offer the same disk space their desktop counterparts do. Along with storage limits, constrained interfaces and energy consumption place very real restrictions on what you as a developer can accomplish.

With the iPhone, you can't design for a desktop-sized screen, for a mouse, or for a physical always-on A/C power supply. Platform realities must shape and guide your development. Fortunately, Apple has done an incredible job designing a new platform that somehow leverages flexibility from its set of limited storage, limited interaction controls, and limited battery life.

Storage Limits

The iPhone hosts a powerful yet compact OS X installation. Although the entire iOS fills no more than a few hundred megabytes of space—almost nothing in today's culture of large operating system installations—it provides an extensive framework library. These frameworks of precompiled routines enable iPhone users to run a diverse range of compact applications, from telephony to audio playback, from e-mail to web browsing. The iPhone provides just enough programming support to create flexible interfaces while keeping system files trimmed down to fit neatly within tight storage limits.

> **Note**
>
> Each application is limited to a maximum size of 2GB. To the best of my knowledge, no application has ever fully approached this size, although there are some navigation apps that are pushing new records of deployment size, such as Navigon (1.5GB) and Tom Tom (1.4GB). Many users complain when applications exceed about 10MB. Apple currently restricts apps larger than 20MB to Wi-Fi downloading. This bandwidth was set at the time that Apple announced its new iPad device and the possibility of delivering universal applications that could run on both platforms. Apple's over-the-air restrictions help reduce cell data load when media-intense applications exceed 20MB and ease the pain of long download times. The 20MB limit is also an important design consideration. Keeping your size below the 20MB cutoff allows mobile users to make impulse application purchases, increasing your potential user base.

From the Library of Eric Berna

Data Access Limits

Every iOS application is sandboxed. That is, it lives in a strictly regulated portion of the file system. Your program cannot directly access other applications, certain data, and certain folders. Among other things, these limitations minimize or prevent your interaction with the iTunes library and the calendar.

Your program can, however, access any data that is freely available over the air when the iOS device is connected to a network—including any iCloud documents that it owns, data stored in the shared system pasteboard, and data shared using a document interaction controller, which introduces a limited way to share document data between applications. Apps that create or download data can now send those files to applications that can view and edit that data. In that situation, the data is fully copied from the originating application into the sandbox of the destination application.

Memory Limits

On iOS, memory management is critical. Apple has not enabled disk-swap-based virtual memory for iOS. When you run out of memory, iOS shuts down your application—as Apple puts it, random crashes are probably not the user experience you were hoping for. With no swap file, you must carefully manage your memory demands and be prepared for iOS to terminate your application if it starts swallowing too much memory at once. You must also take care concerning what resources your applications use. Too many high-resolution images or audio files can bring your application into the autoterminate zone.

In the past, Apple system engineers have suggested that applications need to stay within 20MB of RAM. The rough rule of thumb that circulated in developer circles was this: At about 20MB of use, the iPhone begins to issue memory warnings. At around 30MB, iOS shuts down the application. With newer models offering more onboard RAM, this rule is surely due for revision as iOS devices evolve. Unfortunately, Apple has not made any new rules clear at the time this book was being updated. Table 1-2 offers a basic overview of model-by-model features, including RAM availability. Grayed entries indicate models that are no longer supported by modern firmware releases.

> **Note**
>
> Xcode automatically optimizes your PNG images using the pngcrush utility shipped with the SDK. (You find the program in the iPhoneOS platform folders in /Developer.) Run it from the command line with the `-iphone` switch to convert standard PNG files to iPhone-formatted ones. For this reason, use PNG images in your iPhone apps where possible as your preferred image format. The open-source fixpng utility, which is hosted at http://www.cyberhq.nl, goes the opposite way. It restores compressed images back to Mac-friendly formats and is a valuable tool to have on hand for iPhone development. The venerable Graphics Converter application (http://lemkesoft.com, $35) also offers iPhone PNG support.

From the Library of Eric Berna

Table 1-2 **Historic iOS Devices and Their Capabilities**

Model	RAM/Storage	Processor	Display	Features
Original iPhone (*iPhone 1,1*)	128MB 4, 8, 16GB	620MHz Samsung RISC ARM (underclocked to 412MHz)	320×480 at 163 ppi	2.0 MP camera, speaker, microphone, accelerometer, brightness sensor, proximity sensor, telephony, vibration, 802.11b/g.
iPhone 3G (*iPhone 1,2*)	128MB 8, 16GB	620MHz Samsung RISC ARM (underclocked to 412MHz)	320×480 at 163 ppi	2.0 MP camera, speaker, microphone, GPS, accelerometer, brightness sensor, proximity sensor, telephony, vibration, 802.11b/g.
iPhone 3GS (*iPhone 2,1*)	256MB 8, 16, 32GB	833MHz ARM Cortex-A8 (underclocked to 600MHz)	320×480 at 163 ppi	3.0 MP autofocus video camera, speaker, microphone, accelerometer, brightness sensor, proximity sensor, GPS, voice control, magnetometer, telephony, vibration, 802.11b/g.
iPhone 4 (*iPhone 3,x*)	512MB 16, 32GB	1GHz Apple A4	640×960 at 326 ppi	Dual cameras, including 5.0 MP autofocus back camera and front video camera, LED flash, speaker, accelerometer, brightness sensor, proximity sensor, dual noise-cancelling microphones, GPS, voice control, magnetometer, gyroscope, telephony, vibration, 802.11b/g/n.
iPhone 4S *(iPhone 4,x)*	512MB	Apple dual-core A5	640×960	Dual cameras, including 8.0 MP autofocus back camera and front video camera, LED flash, speaker, accelerometer, brightness sensor, proximity sensor, dual noise-cancelling microphones, GPS, voice control, Siri voice assistant, magnetometer, gyroscope, telephony, vibration, 802.11b/g/n.

From the Library of Eric Berna

Table 1-2 **Historic iOS Devices and Their Capabilities**

Model	RAM/Storage	Processor	Display	Features
iPod touch 1G (*iPod 1,1*)	128MB 8, 16, 32GB	620MHz Samsung RISC ARM (underclocked to 412MHz)	320×480 at 163 ppi	Accelerometer, brightness sensor, 802.11b/g.
iPod touch 2G (*iPod 2,1*)	128MB 8, 16, 32GB	620MHz Samsung RISC ARM (underclocked to 532MHz)	320×480 at 163 ppi	Accelerometer, brightness sensor, speaker, 802.11b/g.
iPod touch 3G (*iPod 3,1*)	256MB 32, 64GB	833MHz ARM Cortex-A8 (underclocked to 600MHz)	320×480 at 163 ppi	Accelerometer, brightness sensor, speaker, voice control, 802.11b/g.
iPod touch 4G (*iPod 4,1*)	256MB 8, 16, 32, 64GB	1GHz Apple A4	640×960 at 326 ppi (Retina)	Accelerometer, brightness sensor, gyroscope, speaker, voice control, 802.11b/g/n, dual cameras, including 960×720 back camera with digital zoom and VGA front video camera.
iPad and iPad 3G (*iPad 1,1*)	256MB 16, 32, 64GB	1GHz Apple A4	768×1024 at 132 ppi	Accelerometer, brightness sensor, speaker, microphone, magnetometer, 802.11a/b/g/n. GPS on 3G model.
iPad 2 and iPad 2 3G (*iPad 2,x*)	512MB 16, 32, 64GB	1GHz dual-core Apple A5	768×1024 at 132 ppi	Accelerometer, brightness sensor, speaker, microphone, gyro, magnetometer, 802.11a/b/g/n, dual cameras, including 960×720 back camera with digital zoom and VGA front video camera. GPS on 3G model.
iPad 3 (iPad 3,x)	*TBD—likely 512MB, possibly 1GB*	*TBD—likely Apple A5*	*Likely 768×1024 or possibly 1536×2048*	*TBD—this model was not yet introduced at the time this book was being written.*

From the Library of Eric Berna

Interaction Limits

For the iPhone and iPod touch, losing physical input devices such as mice and working with a tiny screen doesn't mean you lose interaction flexibility. With multitouch and the on-board accelerometer, you can build user interfaces that defy expectations and introduce innovative interaction styles. The iPhone's touch technology means you can design applications complete with text input and pointer control using a virtual screen that's much larger than the actual physical reality held in your palm.

> **Note**
>
> iOS supports external keyboards starting with the 3.2 firmware for 3GS units and newer. You may connect Bluetooth and USB keyboards to iOS devices for typing. Although USB keyboards can be used with the Camera Connection Kit ($29 at the Apple Store) for development and testing, they are not officially supported for use by Apple.

A smart autocorrecting onscreen keyboard, built-in microphone (for all units except on the obsolete first-generation iPod touch), and an accelerometer that detects orientation provide just a few of the key technologies that separate the iOS family from the rest of the mobile computing pack. What this means, however, is that you need to cut back on things such as text input and scrolling windows, especially when developing for the smaller members of the iOS family. Don't ever assume that users will be dragging along a physical keyboard to use; design accordingly.

Focus your design efforts on easy-to-tap interfaces rather than on desktop-like mimicry. Remember to use just one conceptual window at a time—unlike desktop applications that are free to use a more flexible multiwindow display system.

> **Note**
>
> The iPhone screen supports up to five touches at a time, although it's rare to find any application that uses more than two at once. The iPad screen supports up to ten touches (approximately) at a time. With its larger screen the iPad invites multihand interaction and gaming in ways that the iPhone cannot, particularly allowing two people to share the same screen during game play. Apple has not specified the actual iPad touch limit at the time of writing this book.

Energy Limits

For mobile platforms, you cannot ignore energy limitations. That being said, Apple's SDK features help to design your applications to limit CPU use and avoid running down the battery. A smart use of technology (for example, properly suspending themselves between uses) lets your applications play nicely on the iPhone and keeps your software from burning holes in users' pockets (sometimes almost literally, speaking historically). Some programs, when left running, produce such high levels of waste heat that the phone becomes hot to the touch and the battery quickly runs down. The Camera application was one notable example.

Starting with the iPad, battery life took a huge leap forward. The iPhone 4 and fourth-generation iPod touch continued this trend, providing better and longer battery capability

From the Library of Eric Berna

for the handheld family of devices. Regardless, you should continue to keep energy consumption in mind when developing your applications.

Application Limits

Until iOS 4, Apple maintained a strong "one-application-at-a-time" policy. That meant as a third-party developer you could not develop applications that ran in the background like Apple's Mail, Phone, and Mobile Safari utilities. Each time your program ran, it had to clean up and metaphorically "get out of Dodge" before passing control on to the next application selected by the user. iOS 4 introduced a limited form of multitasking.

With iOS multitasking, applications can either allow themselves to be suspended completely between uses (the default behavior), to quit entirely between uses (set by an Info.plist key), to run for a short period of time to finish ongoing tasks, or to create background tasks that continue to run as other applications take control. Currently supported background tasks include playing music and other audio, collecting location data, and using Voice over IP (VoIP) telephony. Rather than running a simple background daemon, these tasks are event driven. Your application is periodically called by iOS with new events, allowing the application to respond to audio, location, and VoIP updates.

Multitasking applications cannot update views (especially OpenGL ES views, because OpenGL ES drawing commands will cause your application to be terminated) or use Bonjour from the background. Apple does not allow you to create interface-less Bonjour-powered services, which is a shame in my opinion because those services could provide powerful and flexible solutions between the iPhone and desktop systems as well as iPhone-to-iPhone solutions.

As another solution to multitasking needs, Apple supports push data from web services. Push sends processing off-device to dedicated web-based services, leveraging their always-on nature to limit on-device processing requirements. Registered services can push badge numbers and messages to users, letting them know that new data is waiting on those servers.

In addition, applications can pass control from one to the other by passing data (using the document interaction controller) and by opening custom URL schemes.

Apple strongly encourages developers to limit the amount of cell-based data traffic used by each application. The tendency of carriers to meter data usage and the overall movement away from unlimited data plans helps reinforce this requirement. Applications that are perceived to use too much cell bandwidth may be rejected or pulled from the store. If your application is heavily bandwidth dependent, you may want to limit that use to Wi-Fi connections.

802.11n support was first added to the iPad 1 and iPhone 4. Earlier devices provide only 802.11a, b, and g connections. This is limited "n" support. Newer units handle 2.4GHz connections only; there is no 5GHz support. The iPhone 4 also introduced 5.8Mbps HSUPA in addition to the 7.2Mbps HSDPA support that debuted with the iPhone 3GS.

From the Library of Eric Berna

> **Note**
>
> According to the iPhone Terms of Service, you may not use Cocoa Touch's plug-in architecture for applications submitted to the App Store. You can build static libraries that are included at compile time, but you may not use any programming solution that links to arbitrary code at runtime.

User Behavior Limits

Although this is not a physical device-based limitation, iPhone users approach phone-based applications sporadically. They enter a program, use it quickly, and then leave just as quickly. The handheld nature of the device means you must design your applications around short interaction periods and prepare for your application to be cut off as a user receives a phone call or sticks the phone back into a pocket. Keep your application state current between sessions and relaunch quickly after reboots to approximate the same task your user was performing the last time the program was run. This can demand diligence on the part of the programmer but is worth the time investment due to the payoff in user satisfaction.

SDK Limitations

As you might expect, building applications for the iPhone is similar to building applications for the Macintosh. Both platforms run a version of OS X. You use Objective-C 2.0 to develop your code. You compile by linking to an assortment of frameworks. In other ways, the iOS SDK is limited. Here are some key points to keep in mind:

- Garbage collection is missing in action and probably always will be. The LLVM 3.0 compiler's new automatic reference counting (ARC) optimizations make its absence less and less of an issue. ARC takes basic memory management out of the development equation, letting you focus on application semantics instead. The missing garbage collection can be explained in two ways. First, a constrained mobile platform like the iPhone demands precise performance characteristics, especially for processor-intense applications such as games. Garbage collection adds an unpredictable element to performance; it must freeze threads when it cleans up memory. Second, limited memory does not allow garbage collection to be implemented in any sane and useful manner. Garbage-collected applications use a higher watermark for memory usage. This subjects applications to more OS shutdowns.
- Many libraries are still only partly implemented, although this continues to improve with each release of iOS firmware. Core Animation is available through the Quartz Core framework, but some classes and methods remain missing in action; ditto for Core Image. The lesson here is that you're always working in early-release software, even though it has been quite some time since the first SDK debuted. Work around the missing pieces and make sure to submit your bug reports and feature requests to Apple at bugreport.apple.com so that it (one hopes) fixes the parts that need to be used. Be aware that Apple has deliberately restricted access to some proprietary classes and methods.

From the Library of Eric Berna

> **Note**
>
> Apple expanded the iOS version of Objective-C 2.0 starting with the 4.0 SDK to introduce blocks. Blocks are a technology that has been around for decades in languages such as Scheme, Lisp, Ruby, and Python. They allow you to encapsulate behavior as objects, so you can pass that behavior along to methods as an alternative to callbacks. This new feature is introduced in Chapter 2, "Objective-C Boot Camp."

Using the Provisioning Portal

The iPhone Developer Program's provisioning portal hosts all the tools needed to set up your system for iPhone development. It is found at http://developer.apple.com/ios/manage/overview/index.action, and you will not have access to it unless you have signed up for one of the two paid iOS Developer Programs. Here is where you can set up your development team, obtain your certificates, register development devices and application identifiers, and build your provisioning profiles so you can properly sign your applications.

Because the details are subject to change, this overview focuses on the big picture. Should Apple alter any of the particulars, you'll still know what the major milestones are, so you can adjust accordingly. Figure 1-1 shows the key points of the process.

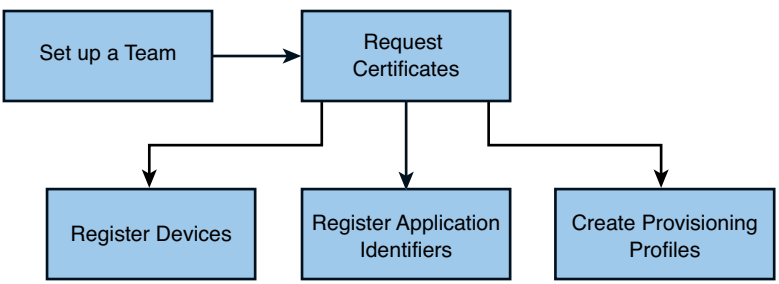


Figure 1-1 Basic functions of the iOS provisioning portal.

Setting Up Your Team

An iOS development team consists of one or more members. The primary member of the team, called the "agent," is the original person who enrolled into the iOS Developer Program. The agent has basic administrative powers over the account: He or she can add other members to the team if this is not an individual account, approve certificate requests, and so forth. In addition, the agent can grant administrative privileges to other members, who are called, unsurprisingly, "admins." Members without administrative privileges can request new provisions and download them, but that's pretty much the limit.

Admins can invite new members at the portal using the Team screen. This is also where you can update e-mail, check on certificates, and add and remove members. Additional tabs in this screen let you check your technical support incidents and review your developer agreements with Apple.

From the Library of Eric Berna

Requesting Certificates

Certificates play a major role in iOS development. A **certificate** is a verified signature that ensures your software came from you rather than an untrusted source. You cannot deploy applications to iOS devices, even for testing, without a valid development certificate. You also need a distribution certificate for selling applications through the App Store. You can request and download these certificates from the portal.

Start by generating a certificate request from your Macintosh's Keychain Access utility:

1. Launch the program from the /Applications/Utilities folder.
2. Choose Keychain Access > Certificate Assistant > Request a Certificate from a Certificate Authority. Check your e-mail address, fill in the Common Name, choose Saved to Disk, and click Continue.
3. Select where to save the certificate (the desktop is a good choice) and click Save. Wait for the certificate to generate and then click Done.

You then upload the request at the portal to create either your development or distribution certificate. The portal walks you through the process. Each certificate must be approved by the team agent before it is issued. Once it is approved, you can download it from the Certificates window on the portal site.

Install the new certificate into your keychain by double-clicking it. Certificates are currently good for one year. Make sure you remove any expired certificates from your keychain because Xcode cannot readily distinguish between them. You will encounter problems compiling until you do so. Select the expired certificate in the Macintosh Keychain Access application (/Applications/Utilities/Keychain Access.app) and delete it.

In addition to these two certificates, you must also install the WWDR intermediate certificate issued by Apple's worldwide developer relations. It can be downloaded from the portal or directly at http://developer.apple.com/certificationauthority/AppleWWDRCA.cer. Make sure you add this to your keychain as well.

Should you need to develop on more than one machine at a time, you can export your developer profile from Xcode's Organizer. Click Export Developer Profile, enter a password that you will remember, and then verify that password. Save the file to a convenient location such as the desktop. You will generally have to enter an administrator password during the process to allow Xcode to access your local keychain. Once this is created, you can transfer the profile file to another Macintosh system and import it through the same Xcode Organizer screen. You will be prompted for the password.

Registering Devices

You must register all development devices either at the program portal or, more conveniently, through Xcode. In the standard program, you can register up to 100 devices at any time. Once it is registered, you may use that device for both development and ad hoc provisions. Registered devices are eligible for beta OS installs.

From the Library of Eric Berna

At the portal, you provide a device name and its unique device identifier (UDID). When you have a development device on hand, you can submit that device for registration directly from Xcode's device organizer window.

In Xcode, simply open the organizer window (Window > Organizer, Command-Shift-2). Select a device, right-click (Control-click) it, and choose Add Device to Provisioning Portal. You will be prompted to log in to the portal with your developer credentials. Enter your account name and password and then click Log In. Xcode will upload your device details and then download an updated "team" provision that is automatically managed by Xcode.

To manually register a device, typically one that you do not physically have present, navigate to the developer portal and view the Devices screen. Click Add Device. Enter a name, enter a UDID, and click Submit. This process does not affect any existing provisions. If you want to add the device you just registered to a provision, you will have to do so separately from registering it.

Finding UDIDs is not complicated: You can easily recover a device UDID from iTunes. When the device is docked, select its name from the sources list (the left iTunes column) and view the Summary tab. Click the words Serial Number. This changes the display from Serial Number to Identifier (UDID). Choose Edit > Copy (Command-C), and the UDID transfers to your system clipboard. You can then paste that number into a file.

Alternatively, have your users download a copy of Ad Hoc Helper (http://itunes.com/apps/adhochelper) to their iOS device. It is a free utility I created to help people e-mail their device IDs directly to a developer. When launched, it automatically starts a new e-mail that is populated with the user's UDID and device information. Users add your address as the recipient and tap Send.

Apple offers several ways to register multiple devices at once. The most reliable option is to enter several items into the Add Devices screen before clicking the Add Device button. You can also use Apple's Configuration Utility to manage UDIDs. It is available for download at the portal site but has had its ups and downs in terms of stability.

Please note that Unregister does not immediately free up slots on your 100-slot devices list. Due to some developers abusing the system, there is a one-year timeout period before a slot can be reused. You can contact Apple at devprograms@apple.com and ask them to override this setting if there is a valid reason your slots need to be reused within the year.

Registering Application Identifiers

Each application you build should use an exclusive identifier. This string enables your application to uniquely present itself to iOS and guarantees that it will not conflict with another application. Most typically, you build your identifiers using reverse domain notation (for example, com.sadun.myApplicationName, uk.co.sadun.myApplicationName, org.sadun.myApplicationName, and so on). Avoid special characters in your application identifiers.

During development, you can rely on team provisioning to cover any on-device testing. Team provisioning is automatically generated from Xcode and simplifies what used to be a more complex process.

From the Library of Eric Berna

You should register at least one "wildcard" identifier at the portal. By this, I mean an identifier that uses an asterisk as a wildcard matching character (for example, com.sadun.*). You can use this single identifier to create a distribution provision that works with all your applications for application signing. A wildcard provision properly signs all applications whose identifiers match its pattern. You compile for distribution with this provision, but you will still need to register an identifier to submit to iTunes Connect.

iTunes Connect requires you to register each application at the portal that you will be submitting to the App Store for distribution. You need not register an individual application identifier until you are ready to submit your application to the App Store. One exception to this rule is application identifiers meant to be used with push notifications, and another is iCloud. You will need to register those identifiers for development and create provisions for them so you can test your applications from the sandbox and properly deploy them to the App Store.

Register new application identifiers by visiting the App IDs page in the provisioning portal. Click New App ID. Enter a freeform text description (for example, "My Sample App"), and a bundle identifier (for example, com.sadun.mysampleapp).

Locate the Bundle Seed ID pop-up. If your application is guaranteed to never need to share a keychain with another application, go ahead and leave the selection at Generate New and then click Submit. When you do need shared keychains, be sure to pick a shared common bundle seed prefix, typically matching your new application identifier prefix to an existing application whose keychain you want to share.

For the most part, the examples for this book use a single application identifier, com.sadun.helloworld. You'll be able to sign these applications *for development only* using your team provisioning profile because it matches all identifiers. I chose to use just one identifier to avoid clogging up your iPhone with dozens of examples at once. Each example replaces the previous one, ensuring that your home screen remains relatively uncluttered.

> **Note**
>
> If you're wondering what those random characters that precede your registered IDs are, they are bundle seed IDs and are meant to be used with applications that share keychain data. Consult the Apple portal for more details about using seed IDs. Seed IDs are different from the development team identifiers used for sharing iCloud documents between applications.

Provisioning

Provisioning profiles provide a way to associate registered developers and registered devices with a specific iPhone development team. They are used in Xcode to sign your code, authorizing the software to run on the device or to be allowed in the App Store.

Most developers use two key provisions: a team development provision, which is managed by Xcode, and a wildcard distribution provision. In addition, most developers eventually build one or more ad hoc provisions—which allow you to distribute your application outside the App Store to devices you have registered at the portal—and some developers create special-purpose provisions for applications that support remote notification.

From the Library of Eric Berna

Xcode now automatically creates and manages your team provisioning profile, connecting directly to the iPhone provisioning portal. This feature deprecated the need to create a separate wildcard development provision.

This team profile allows you to share provisioning among all devices registered to your account, even if your team members are remote, and to deploy development builds to any of those devices. Your team provision matches all application identifiers, allowing you to compile and test all your apps on your locally connected devices.

Distribution profiles are used to build applications for the App Store. Create your App Store distribution profile at the Provisioning screen of the program portal website. Choose the Distribution tab, click Add Profile, check the certificate name box, and choose your wildcard application ID. For development and ad hoc provisions, you must select the devices that are included. Click Submit and then refresh the screen a few times. It usually takes less than a minute for the provision to be generated and made available for download.

Should you need to add devices at a later time, you can easily do so. Expand the device user base by editing your already-issued provisions. Choose Edit > Modify, check the new devices, and click Submit. Re-download the updated provisioning profile by clicking Download.

To install provisions, drag them onto the Xcode icon or (for development and ad hoc provisions only) drop them into the Xcode Organizer window for the device. Xcode automatically reads them in and installs them into your home folder in ~/Library/MobileDevice/Provisioning Profiles. To remove a provision, use the Xcode Organizer's Provisioning Profiles pane.

Xcode automatically installs provisions onto devices to ensure that applications compiled with those provisions can run properly. To remove a provision from a device, open Settings > General > Profiles on the iPhone, iPad, or iPod touch in question. Select a profile and then click the red Remove button. When you remove a device provision, you won't be able to run any applications signed with that provision.

When you update provisions, you will need to provide updated provision profiles to any test users. You may want to add dates and version information to profile names, especially for ad hoc provisions. This avoids confusion when deciding which provision to remove.

You are automatically notified on your device whenever a provision is about to expire. Over and over and over.

Putting Together iPhone Projects

iPhone Xcode projects contain varied standard and custom components. Figure 1-2 shows a minimal project. Project elements include source code, linked frameworks, and media such as image and audio files. Xcode compiles your source, links it to the frameworks, and builds an application bundle suitable for iPhone installation. It adds your media to this application bundle, enabling your program to access that media as the application runs on the iPhone.

From the Library of Eric Berna

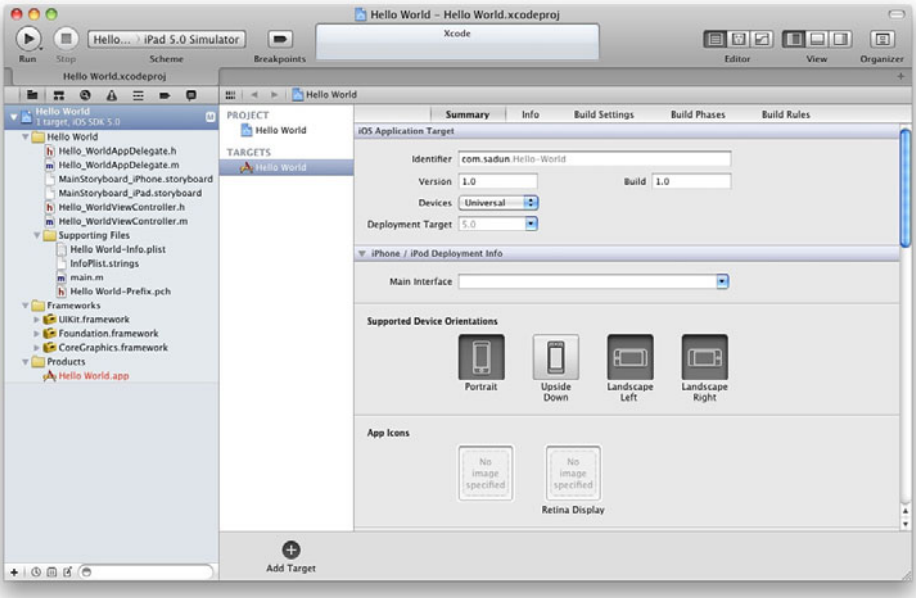


Figure 1-2 Xcode projects bring source code, frameworks, and media together to form the basis for iPhone applications.

iPhone code is normally written in Objective-C. This is an object-oriented superset of ANSI C, which was developed from a mix of C and Smalltalk. Chapter 2 introduces the language on a practical level. If you're looking for more information about the language, Apple provides several excellent online tutorials at its iPhone developer site. Among these are an introduction to object-oriented programming with Objective-C and an Objective-C reference (http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/ObjectiveC/).

> **Note**
>
> Xcode's compiler lets you mix C++ and Objective-C code in the same project. The resulting Objective-C++ hybrid projects let you reuse existing C++ libraries in Objective-C applications. LLVM 3.0 now supports C++0x. Consult Apple's documentation for details.

Frameworks are software libraries provided by Apple that supply the reusable class definitions for Cocoa Touch. Add frameworks to Xcode by dragging them onto your project's Frameworks folder. After including the appropriate header files (such as UIKit/UIKit.h or QuartzCore/QuartzCore.h), you call their routines from your program.

From the Library of Eric Berna

Associated media might include audio, image, and video files to be bundled with the package as well as text-based files that help define your application to the iPhone operating system. Drop media files into your project and reference them from your code.

The project shown in Figure 1-2 is both simple and typical despite its fairly cluttered appearance. It consists of source files (main.m, Hello_WorldAppDelegate.h, Hello_WorldAppDelegate.m, Hello_WorldViewController.h, Hello_WorldViewController.m) and interface files (MainStoryboard_iPhone.storyboard, MainStoryboard_iPad.storyboard) along with the default iPhone project frameworks (UIKit, Foundation, and Core Graphics) and a few supporting files (Hello_World-Info.plist, InfoPlist.strings). Together these items form all the materials needed to create an extremely basic application. As you discover in Chapter 3, "Building Your First Project," Xcode can generate most of these elements automatically for you. You then edit them as needed to add functionality.

> **Note**
>
> The HelloWorld-Prefix.pch file is created automatically by Xcode. It names header files to be precompiled.

The iPhone Application Skeleton

Nearly every iPhone application you build will contain a few key source files. Figure 1-3 shows the most common source code pattern: a main.m file, an application delegate, and a view controller. These files (more if you use Interface Builder XIBs, and more if you are building a universal application that is meant to run both on the iPad as well as iPhone devices) provide all the components necessary to create a simple Hello World–style application that displays a view onscreen.

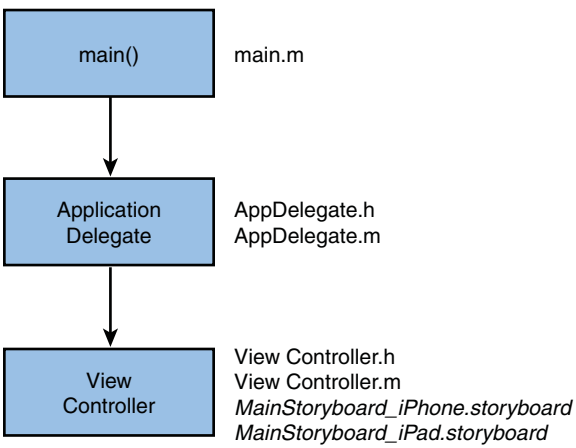


Figure 1-3 These files comprise the most common source code configuration for a minimal iOS application. You may or may not choose to use storyboards to define interfaces.

From the Library of Eric Berna

Some of these components may be familiar. Others may not. Here's a rundown of the file types:

- The implementation files use an .m extension and not a .c extension. These .m files contain Objective-C method implementations in addition to any C-style functions. The project in Figure 1-3 uses three .m files.
- iPhone source files use the standard C-style .h extension for the header files. Header files offer public declarations of class interfaces, constants, and protocols. You usually pair each class implementation file (in this case, the application delegate and view controller .m files) with a header file, as you can see in Figure 1-3.
- Storyboard (.storyboard) and XIB files (.xib) are created in Interface Builder. These XML-based user interface definition files are linked to your application and called by your app at runtime in their compiled formats. The project in Figure 1-3 uses two storyboards to define view controller story arcs for iPhone and iPad platforms. The custom view controller subclass can be used with any or all the scenes in the storyboard.

Here is a quick rundown of those files, what they are, and what role they play in the actual application.

main.m

The main.m file has two jobs. First, it creates a primary autorelease pool for your application. In manually managed memory applications (called manual retain/release, or MRR), this takes the form of an `NSAutoreleasePool` object. In LLVM applications, it's added via an autorelease pool construct, which supports creating and managing autorelease pools more efficiently. Second, it invokes the application event loop. These two elements provide the critical pathway to get your application started and running. Listings 1-1 and 1-2 show how those two items are defined for ARC and MRR apps.

Listing 1-1 **main.m for LLVM Apps**

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

From the Library of Eric Berna

Listing 1-2 **main.m for non-LLVM/MRR Apps**

```
int main(int argc, char *argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil,
        @"MyAppDelegateClass");
    [pool release];
    return retVal;
}
```

> **Note**
>
> The `argc` and `argv` variables passed to `main()` refer to command-line arguments. Because the iPhone does not use a command line to launch its programs (all applications are run from a common graphical user interface), these elements are not used. They are included for consistency with standard ANSI C practices.

Autorelease Pools

Autorelease pools are objects that support the iPhone's memory management system. This memory system is normally based on keeping track of reference counts, counting how many objects refer to an allocated part of memory. Autorelease pools allow software to delay releasing objects until the end of every event loop cycle. They are used in both ARC and MRR compilation, although with ARC you no longer have to use patterns like this:

```
[[[Someclass alloc] init] autorelease]
```

With ARC, the compiler takes charge of all releasing behavior. You specify what *kind* of behavior storage items use in your applications (hold onto this until reassignment, create it to act as autoreleasing, use a zero-referencing pointer to this) but you don't have to send `release` or `autorelease` messages yourself.

In both ARC and MRR, once added to the autorelease pool, objects pass their release responsibilities along to the pool. At the end of each event loop, the pool drains and sends releases to every object it owns. The iPhone expects that there will always be an autorelease pool available, however created, so memory can be recovered from these objects at the end of their lifetime. If you ever create a secondary thread in your application, you need to provide it with its own autorelease pool. Autorelease pools and the objects they contain are discussed further in Chapter 2.

The `UIApplicationMain` Function

The `UIApplicationMain` function provides the primary entry point for creating a new application object. It creates the new application instance and its delegate. The delegate is responsible for handling application status changes and providing program-specific responses to those changes.

The third and fourth arguments of the `UIApplicationMain` function specify the name of the principal application class and its delegate, respectively. If the third argument

From the Library of Eric Berna

is omitted (set as `nil`), which is the default, the iPhone defaults to using the standard `UIApplication` class.

`UIApplicationMain` also establishes the application's event loop. An event loop repeatedly looks for low-level user interactions such as touches on the screen or sensor triggers. Those events are captured by the iPhone's kernel and dispatch an event queue, which is forwarded to the application for handling.

Event loops let you design your program around callbacks. Callbacks are where you specify how the application should respond to these events. In Objective-C, this corresponds to method invocations. For example, you can build methods to determine how the application should reorient itself when the user moves the screen from portrait to landscape or how views should update when a finger is dragged onscreen. This style of programming is based on the underlying event loop, which is set up in main.m.

Application Delegate

An **application delegate** implements how your program should react at critical points in the application life cycle, as shown in Figure 1-4. The delegate is responsible for initializing a windowing system at launch and wrapping up business at termination. It also acts as the key player for handling memory warnings. Here are the more important delegate methods your applications will implement:

- **The `application:didFinishLaunchingwithOptions:` method**—This method is the first thing triggered in your program after the application object has been instantiated. Upon launch, this is where you create a basic window, set its contents, and tell it to become the key responder for your application. If your application was launched via URL, control passes to `application:openURL: sourceApplication:annotation:` or `application:handleOpenURL:` for further handling. Otherwise your launched-with-options method can handle conditions from standard launching by tapping to launching via remote notifications, to app-to-app document sharing, and more.
- **The `applicationWillResignActive:` and `applicationDidBecomeActive:` methods**—Use these methods to prepare the user interface for shutdown and restore it during relaunch. Pause tasks, disable timers, and throttle OpenGL ES frame rates. The resign-active method is triggered during incoming phone calls, SMS messages, and calendar notifications, and the become-active method is triggered when these items are ignored or declined.
- **The `applicationWillEnterBackground:` and `applicationWillEnter Foreground:` methods**—These methods enable you to store and retrieve application state when handing control to and receiving control back from the iOS home screen. Use this to save defaults, update data, and close files as well as to invalidate or reactivate timers.

From the Library of Eric Berna

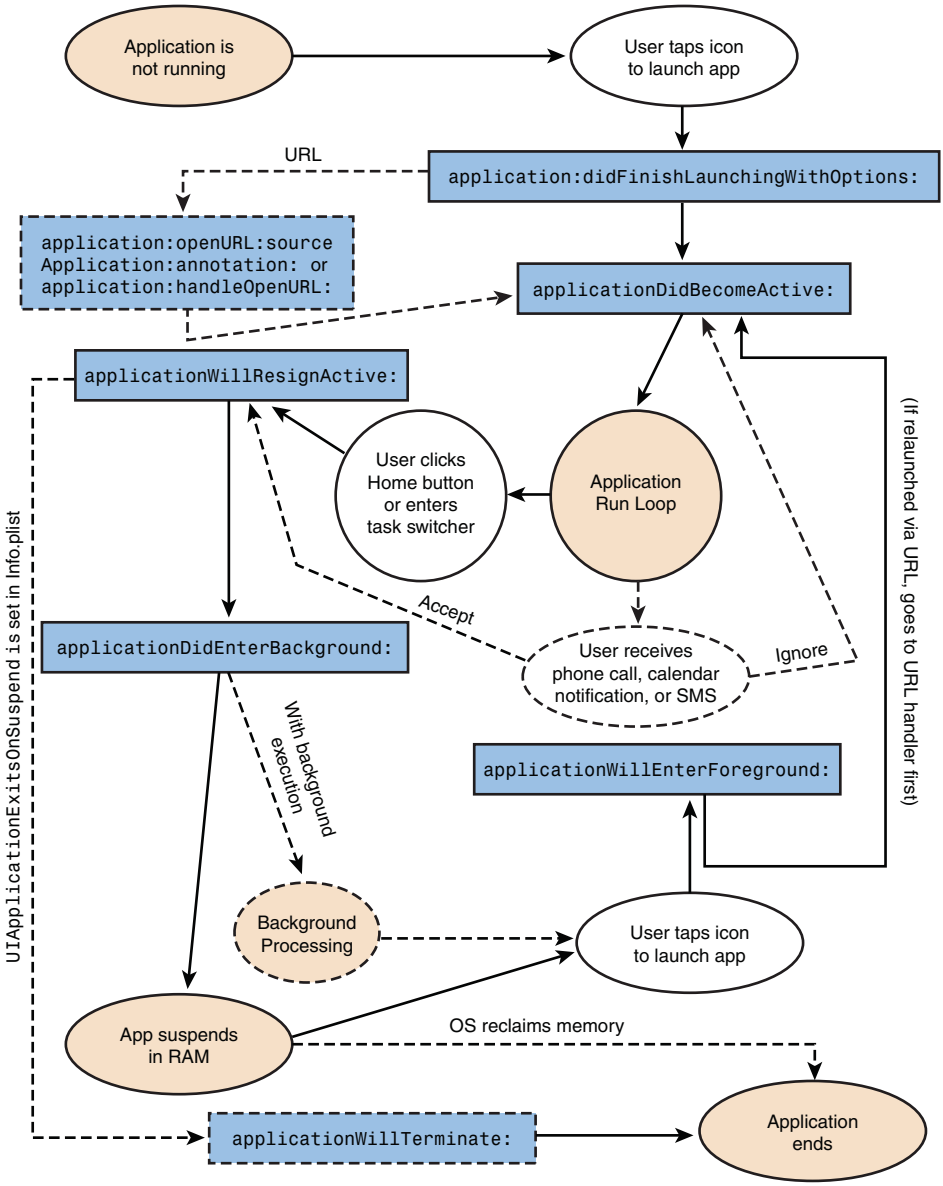


Figure 1-4 The iOS application life cycle.

From the Library of Eric Berna

- **The `applicationDidReceiveMemoryWarning` method**—When this method is called, your application must free up memory to whatever extent possible. This method works hand in hand with the `UIViewController`'s `didReceiveMemoryWarning:` method. If your application is unable to release enough memory, the iPhone terminates it, causing your user to crash back to the home screen.

The application delegate also handles responsibility for managing status bar changes, responding to system notifications, remote and local notifications, and content protection changes using the new iOS 5 protected data file system.

After launching and loading the window, the application delegate takes a back seat. Nearly all application semantics move over to some child of a `UIViewController` class. The application delegate typically does not take a role again until the application is about to finish or if memory issues arise.

> **Note**
>
> Although the application delegate is called for certain events including entering the background or the application becoming active, any object can register itself to receive these kinds of notifications. You'll find a complete list of `UIApplication` notifications in the `UIApplication` class reference documentation. In addition, you can always reference the delegate object from your application via `[[UIApplication sharedApplication] delegate]`.

View Controller

In the iOS programming paradigm, view controllers provide the heart of how an application runs. Here is where you normally implement how the application responds to selections, button presses, as well as sensor triggers. If you haven't used Interface Builder to create a precooked presentation, the view controller is where you load and lay out your views as well as the object that handles device reorientation. Whereas the main.m and application delegate files are typically small, view controller source code is normally extensive, defining all the ways your application accesses resources and responds to users. Some of the key methods for managing your view include the following:

- **The `loadView` and `viewDidLoad` methods**—These methods let you either create (`loadView`) or customize (`viewDidLoad`) your views to prepare them for use. The `loadView` method allows you to set up the screen and lay out any subviews apart from using storyboard or XIB files. Make sure to call `[super loadView]` whenever you inherit from a specialized subclass such as `UITableViewController` or `UITabBarController`. This allows the parent class to properly set up the view before you add your customizations to that setup. Use `viewDidLoad` to finish setup once your views have been created and loaded into memory.
- **The `shouldAutorotateToInterfaceOrientation:` method**—Add the `should-autorotate` method to allow the `UIViewController` method to

From the Library of Eric Berna

automatically match your screen to the iPhone's orientation. You must define how the screen elements should update. Applications developed for the iPad must be usable in multiple orientations (all, portrait, or landscape). The guidelines for iPhone and iPod touch development are less restrictive. You can limit the rotation check to rotate only for certain acceptable orientations, such as only portrait or only landscape.

- **The `viewWillAppear:` and `viewDidAppear:` methods**—These methods are called whenever a view is ready to appear onscreen or a view has fully appeared onscreen, often as part of `UINavigationController` interaction. The `viewWillAppear:` method should update information for views that are about to display. When called, your view may not have been loaded yet. If you rely on accessing IBOutlets connected to subviews, poke `self.view` to ensure the view hierarchy gets loaded. Use `viewDidAppear:` to trigger behavior once the view is fully transitioned onscreen, such as for any animations. With this method, you are guaranteed that your views have been instantiated and you can safely reference onscreen items.

The number and kind of interface files varies with how you design your project. Figure 1-3 assumes you've created a single storyboard for each target platform. You can use Interface Builder to design additional components or skip IB entirely and create your interfaces programmatically. Chapter 4, "Designing Interfaces," introduces a number of scenarios for how you can combine these methods in your applications.

> **Note**
>
> Only `UIView` instances can directly receive touch calls due to being a child class of `UIResponder`; `UIViewController` objects cannot. This "gotcha" will get nearly all (new) iPhone developers at least once. See Chapter 8, "Gestures and Touches," to learn more about directly managing and interpreting touches and gestures in your application. In the most recent versions of Xcode, you can assign view delegates in Interface Builder and add touch event handlers directly into your `UIViewController` implementation file.

A Note about the Sample Code in This Book

For the sake of pedagogy, this book's sample code usually presents itself in a single main.m file. 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.

These coding examples are not intended as standalone applications and do not reflect normal multifile coding practices. 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 system and layout.

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.

From the Library of Eric Berna

Second, standard implementation 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.

iOS Application Components

Compiled iOS applications live in application bundles. Like their Macintosh cousins, these application bundles are just folders named with an .app extension. Your program's contents and resources reside in this folder, including the compiled executable, supporting media (such as images and audio), and a few special files that describe the application to the OS. The folder is treated by the operating system as a single bundle.

Application Folder Hierarchy

iOS bundles are simple. Unlike the Mac, iOS bundles do not use Contents and Resources folders to store data or a MacOS folder for the executable. All materials appear at the top level of the folder. For example, instead of putting a language support (.lproj) folder into Contents/Resources/, Xcode places it directly into the top .app folder. You can still use subfolders to organize your project, but these developer-defined folders do not follow any standard.

The iOS SDK's core OS support includes the `NSBundle` class. This class offers access to the files stored in the application bundle. `NSBundle` makes it easy to locate your application's root folder and to navigate down to your custom subfolders to point to and load built-in resources such as sounds, images, and data files.

> **Note**
>
> As on a Macintosh, user domains mirror system ones. Official Apple-distributed applications reside in the primary /Applications folder. Third-party applications live in /var/mobile/Applications. The underlying UNIX file system is obscured by the iPhone's sandbox, which is discussed later in this section.

The Executable

The executable application file of your application resides at the top-level folder of the application bundle. It carries executable permissions so it can run and is signed as part of the application bundle during the compilation process. You may only load and run applications that have been signed with an official developer certificate. Those certificates are issued by Apple via the iOS developer program portal at the official developer website.

Apple offers several kinds of signing profiles called "mobile provisions" that vary by how the application will be deployed. You use separate provisions for applications that will

From the Library of Eric Berna

be tested during development on a local device, for applications that will be sent out to registered devices for testing, and for those that will be distributed through the App Store. You've already read about creating your provisions earlier in this chapter. The actual application-signing process is discussed in further detail in Chapter 2.

The Info.plist File

As on a Macintosh, the iOS application folder contains that all-important Info.plist file. Info.plist files are XML property lists that describe an application to the operating system. Property lists store key-value pairs for many different purposes and can be saved in readable text-based or compressed binary formats. In an Info.plist file, you specify where the application's executable (`CFBundleExecutable`, "Executable file") can be found, the text that appears under the application icon (`CFBundleDisplayName`, "Bundle display name"), and the application's unique identifier (`CFBundleIdentifier`, "Bundle identifier").

Be careful when setting the display name. Titles that are too long to display properly are truncated; the iPhone adds ellipses as needed. So your application named "My Very First iPhone App" may display as "My Very F..." This provides less information to your end user than a simpler title such as "First App" would offer.

The application identifier typically uses Apple's reverse domain naming format (for example, com.sadun.appname). The identifier plays a critical role for proper behavior and execution; it must not duplicate any other identifier on the App Store. In use, the product identifier registers your application with SpringBoard, the "Finder" of the iOS. SpringBoard runs the home screen from which you launch your applications. The product identifier also forms the basis for the built-in preferences system called the user defaults.

The identifier is case sensitive and must be consistent with the provisions you generate at the developer portal. Problems with misnamed bundle identifiers have cost developers many hours of wasted time. Specify the identifier by editing your project's settings in Xcode (see Figure 1-5).



Figure 1-5 Customize your application's bundle identifier by editing target properties. Edits here are reflected in your application's Info.plist file.

Application preferences are automatically stored in the application sandbox. The sandbox mimics the domains and folders normally found in the core OS. On iOS, preferences

From the Library of Eric Berna

appear in a local Library folder and use the application identifier for naming. This identifier is appended with the .plist extension (for example, com.sadun.appname.plist), and the preferences are stored using a binary .plist format. You can examine a binary .plist by transferring it to a Macintosh via Xcode's Organizer.

> **Note**
>
> To copy application data from iOS to your Macintosh, open the Organizer window (Windows > Organizer). Select your device and then an item from the applications list. Click the arrow next to the name to reveal the Application Data bundle and then drag that bundle to the desktop. It expands to a standard folder named with the application identifier and the date and time the data was retrieved. Xcode limits this access to applications that have been installed outside of App Store channels.

You can edit property list files directly in Xcode or use the Property List Editor that ships as part of Xcode's utilities. It's located in /Developer/Applications/Utilities and offers a user-friendly GUI. Use Apple's plutil utility to convert property lists from binary to a text-based XML format: `plutil -convert xml1 plistfile`. Apple uses binary plists to lower storage requirements and increase system performance.

As with the Macintosh, Info.plist files offer further flexibility and are highly customizable. With them, you can set application-specific variables (`UIRequiresPersistentWiFi`) or specify how your icon should display (`UIPrerenderedIcon`). These variables are powerful. They can define multiple roles for a single application, although this functionality is not available to third-party development. For example, the Photos and Camera utilities are actually the same application, MobileSlideShow, playing separate "roles."

Other standard Info.plist keys include `UIStatusBarStyle` for setting the look and color of the status bar and `UIStatusBarHidden` for hiding it altogether. `UIInterfaceOrientation` lets you override the accelerometer to create a landscape-only (`UIInterfaceOrientationLandscapeRight`) presentation. Register your custom application URL schemes (for example, myCustomApp://) by setting `CFBundleURLTypes`.

The Icon and Launch Images

The icon and launch images are key image files. The icon acts as your application's icon, the image used to represent the application on the iOS home screen. For the iPhone and iPod touch, the Default.png launch image provides the splash screen displayed during application launch. The iPad also uses Default files but with slightly different naming. As Figure 1-6 shows, Xcode 4 now offers an interactive editor for setting your iPhone and iPad icons and launch images.

Apple recommends matching Default.png to your application's background. Many developers use Default.png for a logo splash or a "Please wait" message. These go against Apple's human interface guidelines (launch images should provide visual continuity, not

From the Library of Eric Berna

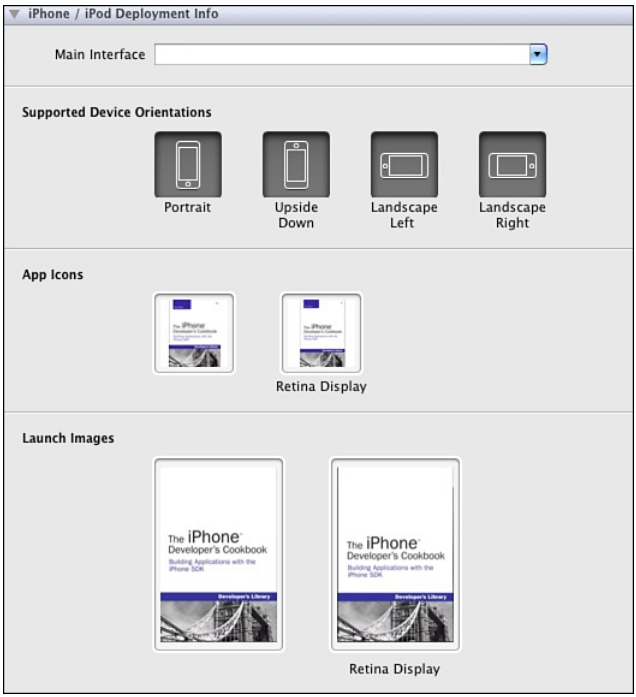


Figure 1-6 Xcode 4 offers an interactive editor where you can specify supported device orientations, application icons, and launch images for both iPhone and iPad deployment. This figure shows the iPhone/iPod deployment section.

advertising or excuses for delays) but are perfectly understandable uses if you really must go there. My personal rules of launch are these: 1. Launch quickly. 2. Don't logo. Your contractual obligations may differ.

Screenshots can work well as launch images. Xcode lets you take screenshots of your application in action using its Organizer window (Window > Organizer). It also offers the option to set one of those shots as your launch image.

For iPad-deployed applications, you can include up to six separate launch images. In addition to the (typically iPhone-sized) Default.png file, you can add Default.-Landscape.png, Default-Portrait.png, Default-PortraitUpsideDown.png, Default-LandscapeLeft.png, and Default-Landscape-Right.png. Each of these has an order of precedence, where a more specific launch image always is used before a more general one. So if you supply Default-LandscapeLeft.png, it is preferred over Default-Landscape.png, which in turn is preferred over Default.png. The interactive Xcode editor limits you to simple landscape and portrait options, which greatly simplifies your art choices.

iPad launch images must take the status bar into account. Whereas the iPhone/iPod Default.png is always 320×480 pixels in size for normal displays and 640×960 pixels for

From the Library of Eric Berna

Retina displays, the iPad versions use slightly different proportions when compared to their screen hardware pixels. iPad launch images must be either 1024×748 pixels in size for landscape versions or 768×1004 pixels for portrait ones, both exclude 20 pixels for the status bar.

When working with universal applications—that is, applications meant for deployment on both iPhone and iPad devices—make sure you provide an array of at least two icons, and preferably three, for the `CFBundleIconFiles` key. The official iPhone application icon size is 57-by-57 pixels for standard units and 114-by-114 pixels for Retina units; it is 72-by-72 pixels for the iPad. Use the handy Xcode Target Summary screen to set your artwork or provide all applicable filenames in the Info.plist array for this key (for example, icon.png, icon@2x.png, icon-iPad.png). Each device will choose the most appropriate icon based on the pixel size of the icons files you provide.

In addition to your iPhone- and iPad-sized icons, you can supply a 50-by-50-pixel icon for iPad apps, to be used by iPad Spotlight, and a 29-by-29-pixel icon to be used on the iPad for settings and on the iPhone for Spotlight and settings. These Spotlight icons are optional but are Apple recommended. Settings icons are also optional; they are recommended only for those applications that include settings bundles. Xcode does not provide an automated interface for these items, so add them by hand to your Info.plist.

The iPhone home screen automatically scales larger icon art, but it's best to supply properly sized art from the start. Provide flat (not glossy) art with squared corners. The iOS home screen smoothes and rounds those corners and adds an automatic gloss and shine effect. If for some compelling reason you need to use prerendered art, set `UIPrerenderedIcon` to `<true/>` in your Info.plist file.

As with all on/off Info.plist items, make sure to set the value for `UIPrerenderedIcon` to the Boolean value true (`<true/>`, the checked box in the Xcode GUI). Using a string for "true" (`<string>true</string>`) may work on the simulator while producing no effect on the iPhone. Also remember that the Xcode property list editor hides the actual key name. Add a field for the "Icon already includes gloss and bevel effects" key and check the box that appears in the value column.

When submitting your application to the App Store, you need to create a high-resolution (512-by-512-pixel) version of your icon. Although you can up-sample your smaller icon.png art, it won't look good. Going the other way allows you to maintain high-quality art that you can compress to your icon as needed. Keep your art simple and compressible. An icon that looks stunning at 512×512 looks muddled and sloppy at 57×57 when overly detailed.

> **Note**
>
> You may include a 29-by-29-pixel image called Icon-settings.png in your project. This image represents your application in the Settings application. Most developers skip this option when they do not include custom settings bundles for their applications. If not included, Settings will omit an image, leaving a blank area to the left of your application name.

From the Library of Eric Berna

Interface Builder Files

Interface Builder creates .storyboard and .xib files that store precooked addressable user interface layouts in XML format. (If you're curious, you can open these files in your favorite text editor and peek at the XML.) Most IB-based applications contain several .storyboard or .xib files that define various view components. Typical contents might include window layouts, custom table cells, pop-up dialogs, and more.

When creating your application bundles, Xcode compiles the XML data and places it alongside the executable and any other application components. The XIB's compiled format is NIB, which somewhat archaically stands for NeXT Interface Builder, the ancestor of the OS X Interface Builder used to build iPhone applications. The .nib files appear at the top level of your application bundle and are used directly from your program when loading screens.

> **Note**
>
> When you develop programs that do not use Storyboard or XIB Interface Builder bundles, remove the main interface key in the deployment settings and discard any automatically generated interface files from your project. This reduces clutter in your program and keeps your application from trying to load an interface file that you never fully defined.

Universal applications—that is, applications that run on both the iPhone and the iPad and share the same code—may use multiple platform-based Storyboard and NIB files. When creating universal applications, Xcode automatically generates both storyboards for both platforms. iOS automatically handles the loading to ensure the proper interface file loads on the proper device. There is no way, at the time this book is being written, to automatically create and use separate XIB files to distinguish between device models, such as the 3GS and the iPhone 4, although you can easily program around this.

Files Not Found in the Application Bundle

As with the Macintosh, things you do not find inside the application bundle include preferences files (generally stored in the application sandbox in Library/Preferences), application plug-ins (stored in /System/Library at this time and not available for general development), and documents (stored in the sandbox in Documents).

At this time, the iOS SDK does not let you prepopulate these folders. Because your program cannot edit or overwrite any files in the application bundle, copy any files that need to be changed, such as database files, to another folder on the first run of your program. As a rule, only user-controlled files should live in the Documents folder. As its name suggests, the folder stores application documents. You can also share your Documents folder directly with your users using the desktop-computer-based iTunes, via the Apps tab.

Another thing that seems to be missing (at least from the Macintosh programmer point of view) is Application Support folders. You should copy your support data, which more rightfully would be placed into an Application Support structure, to your Library folder from the application bundle when your application is first launched. Thereafter, check to make sure that data is there and recopy/reinitialize the data if needed.

From the Library of Eric Berna

In early SDK releases, many developers freely used the Documents folder as an extra Library storage area. Now that that folder is visible to your users, you should reserve it exclusively for the users' control, especially if you enable `UIFileSharingEnabled`, which allows iTunes file management.

IPA Archives

When users purchase your application, they download an .ipa file from iTunes. This file is actually a zipped archive. It contains a compressed payload—namely the app bundle you built from the components just described. iTunes stores .ipa archives in the Mobile Applications folder in the iTunes Library. If you rename a copy of any .ipa file to use the .zip extension, you can easily open it using standard compression software.

Each application is customized on download to ensure that it can only be installed and run on the iPhone devices authorized by your iTunes account. This prevents the application from being shared freely over the Internet. Although software pirates have created cracking tools, these are not as widely used in the wild as developers might fear. As a general rule, avoid integrating antipiracy measures into your applications unless piracy places a real and meaningful support or overhead cost on your business. For most developers, even a small chance of alienating legitimate users places a far higher penalty on their success than any outlying piracy costs. Apple's basic protections ensure that for the most part only those who have purchased and downloaded the application from iTunes can run your software.

You can create well-formed IPA archives directly from Xcode and share them by e-mail or upload them to iTunes Connect, the web portal that powers the App Store. This is most helpful for distributing ad hoc builds in a format that can be easily installed into iTunes. Rather than distribute both an app bundle and a provision file, the single IPA archive contains all the materials needed for the application to run properly on the target device.

> **Note**
>
> When you add files to your project (for example, Notes.rtf) that are meant solely for your own use, be sure to remove those files from your target so they will not go out as part of your application distribution. Developers who have forgotten this lesson in the past have included such gems in their application bundles as server login information, Amazon secret keys, and other private data. Savvy users can read these files. All they have to do is unzip the IPA archive and look in the application bundle folder.

Sandboxes

iOS restricts all SDK development to application "sandboxes" for the sake of security. The iPhone sandbox limits your application's access to the file system to a minimal set of folders, network resources, and hardware. In some ways, it's like attending a restrictive school with a strict principal:

- Your application can play in its own sandbox, but it can't visit anyone else's sandbox.
- You cannot share toys. In other words, you cannot share data (except via the user-controlled system pasteboard and the document interaction controller class). You

From the Library of Eric Berna

cannot mess in the administrative offices. Your files must stay in the folders provided to you by the sandbox, and you cannot copy files to or from other application folders.

- You cannot peek over the fence. Reading from or attempting to write to files outside your sandbox is grounds for App Store rejection. Your application is prevented from writing to most folders outside the sandbox by iOS.
- Your application owns its own Library, Documents, and /tmp folders. These mimic the standard folders you'd use on a less-restrictive platform but specifically limit your capability to write and access this data.

In addition to these limitations, your application must be signed digitally and must authenticate itself to the operating system with a coded application identifier, which you create at Apple's developer program site. Details on how to do this follow in Chapter 3. As a final (more upbeat) note, sandboxing ensures that all program data gets synced whenever your device is plugged into its home computer. This includes any files in the Documents folder and non-cache files in the Library folder.

Programming Paradigms

iPhone programming centers on two important paradigms: object-oriented programming and the Model-View-Controller (MVC) design pattern. The iOS SDK is designed around supporting these concepts in the programs you build. To do this, it has introduced delegation (Controller) and data source methods (Model) and customized view classes (View). Here is a quick rundown of some of the important iPhone/Cocoa Touch design vocabulary used throughout this book.

Object-Oriented Programming

Objective-C is heavily based on Smalltalk, one of the most historically important object-oriented languages. Object-oriented programming uses the concepts of encapsulation and inheritance to build reusable classes with published external interfaces and private internal implementation. You build your applications out of concrete classes that can be stacked together like LEGO toys, because it's always made clear which pieces fit together through class declarations.

Pseudo-multiple inheritance (via invocation forwarding and protocols) provides an important feature of Objective-C's approach to object-oriented programming. iOS classes can inherit behaviors and data types from more than one parent. Take the class `UITextView`, for example. It's both text *and* a view. Like other view classes, it can appear onscreen. It has set boundaries and a given opacity. At the same time, it inherits text-specific behavior. You can easily change its display font, color, or text size. Objective-C and Cocoa Touch combine these behaviors into a single easy-to-use class.

From the Library of Eric Berna

Model-View-Controller

MVC separates the way an onscreen object looks from the way it behaves. An onscreen button (the view) has no intrinsic meaning. It's just a button that users can press. That view's controller acts as an intermediary. It connects user interactions such as button taps to targeted methods in your application, which is the model. The application supplies and stores meaningful data and responds to interactions such as these button taps by producing some sort of useful result. MVC is best described in the seminal 1988 paper by Glenn Krasner and Stephen Pope, which is readily available online.

Each MVC element works separately. You might swap out a pushbutton with, for example, a toggle switch without changing your model or controller. The program continues to work as before, but the GUI now has a different look. Alternatively, you might leave the interface as is and change your application where a button triggers a different kind of response in your model. Separating these elements enables you to build maintainable program components that can be updated independently.

The MVC paradigm on the iPhone breaks down into the following categories:

- **Model**—Model methods supply data through protocols such as data sourcing and meaning by implementing callback methods triggered by the controller.
- **View**—View components are provided by children of the `UIView` class and assisted by its associated (and somewhat misnamed) `UIViewController` class.
- **Controller**—The controller behavior is implemented through three key technologies: delegation, target action, and notification.

Together, these three elements form the backbone of the MVC programming paradigm. The following sections look at each of these elements of the iPhone MVC design pattern in a bit more detail. These sections introduce each element and its supporting classes.

View Classes

The iPhone builds its views based on two important classes: `UIView` and `UIViewController`. These two classes and their descendants are responsible for defining and placing all onscreen elements.

As views draw things on your screen, `UIView` represents the most abstract view class. Nearly all user interface classes descend from `UIView` and its parent, `UIResponder`. Views provide all the visual application elements that make up your application. Important `UIView` classes include `UITextView`, `UIImageViews`, `UIAlertView`, and so forth. The `UIWindow` class, a kind of `UIView`, provides a viewport into your application and provides the root for your display.

Because of their onscreen nature, all views establish a frame of some sort. This frame is an onscreen rectangle that defines the space each view occupies. The rectangle is established by the view's origin and extent.

Views are arranged hierarchically and are built with trees of subviews. You can display a view by adding it to your main window or to another view by using the `addSubview`

From the Library of Eric Berna

method to assign a child to a parent. You can think about views as attaching bits of transparent film to a screen, each piece of which has some kind of drawing on it. Views added last are the ones you see right away. Views added earlier may be obscured by other views sitting on top of them.

Despite the name, the `UIViewController` class does not act as a controller in the MVC sense. It more often acts as a view handler and model than as a controller. Although some will disagree, Apple terminology does not always match the MVC paradigm taught in computer science classes.

View controllers are there to make your life easier. They take responsibility for rotating the display when a user reorients his or her iPhone. They resize views to fit within the boundaries when a navigation bar or a toolbar is used. They handle all the interface's fussy bits and hide the complexity involved in directly managing interaction elements. You can design and build iOS applications without ever using a `UIViewController` or one of its subclasses, but why bother? The class offers so much convenience it's hardly worth writing an application without them.

In addition to the base controller's orientation and view resizing support, two special controllers, the `UINavigationController` and `UITabBarController`, magically handle view shifting for you on smaller iOS devices (the iPhone and iPod touch). The navigation version enables you to drill down between views, smoothly sliding your display between one view and the next. Navigation controllers remember which views came first and provide a full breadcrumb trail of "back" buttons to return to previous views without any additional programming.

The tabbed view controller lets you easily switch between view controller instances using a tabbed display. So if your application has a top ten list, a game play window, and a help sheet, you can add a three-buttoned tab bar that instantly switches between these views without any additional programming to speak of.

On the iPad, the `UISplitViewController` class offers a way to provide landscape- and portrait-savvy interfaces. The split view allows you to display persistent information in a separate left panel (such as the one used in the iPad Mail application) when working in landscape mode. After you switch to portrait mode, that left split view automatically disappears, and its data automatically transfers to a `UIPopoverController`, the class that provides iPad pop-up presentations.

Every `UIViewController` subclass implements a method to load a view, whether through implementing a procedural `loadView` method or by pulling in an already-built interface from a .storyboard or .xib file and calling `viewDidLoad`. This is the method that lays out the controller's main view. It may also set up triggers, callbacks, and delegates if these have not already been set up in Interface Builder.

So in that sense alone, the `UIViewController` does act as a controller by providing these links between the way things look and how interactions are interpreted. And, because you almost always send the callbacks to the `UIViewController` itself, it often acts as your model in addition to its primary role as a controller for whatever views you create and want to display. It's not especially MVC, but it is convenient and easy to program.

From the Library of Eric Berna

Controller

When Apple designs interactive elements such as sliders and tables, they have no idea how you'll use them. The classes are deliberately general. With MVC, there's no programmatic meaning associated with row selection or button presses. It's up to you as a developer to provide the model that adds meaning. The iPhone provides several ways in which prebuilt Cocoa Touch classes can talk to your custom ones. Here are the four most important: delegation, target-action, notifications, and blocks.

Delegation

Many `UIKit` classes use delegation to hand off responsibility for responding to user interactions. When you set an object's delegate, you tell it to pass along any interaction messages and let that delegate take responsibility for them.

A `UITableView` is a good example of this. When a user taps on a table row, the `UITableView` has no built-in way of responding to that tap. The class is general purpose and it has no semantics associated with a tap. Instead, it consults its delegate—usually a view controller class or your main application delegate—and passes along the selection change through a delegate method. This enables you to add meaning to the tap at a point of time completely separate from when the table class was first implemented. Delegation lets classes be created without that meaning while ensuring that application-specific handlers can be added at a later time.

The `UITableView` delegate method `tableView:didSelectRowAtIndexPath:` is a typical example. Your model takes control of this method and implements how it should react to the row change. You might display a menu or navigate to a subview or place a check mark next to the current selection. The response depends entirely on how you implement the delegated selection change method.

To set an object's delegate, assign its delegate property or use some variation on the `setDelegate:` method. This instructs your application to redirect interaction callbacks to the delegate. You let Objective-C know that your object implements delegate calls by adding a mention of the delegate protocol it implements in the class declaration. This appears in angle brackets, to the right of the class inheritance. Listing 1-3 shows a kind of `UIViewController` that implements delegate methods for `UITableView` views. The `MergedTableController` class is, therefore, responsible for implementing all required table delegate methods.

Xcode's documentation exhaustively lists all standard delegate methods, both required and optional. Open Help > Documentation (Command-Option-Shift-?) and search for the delegate name, such as `UITableViewControllerDelegate`. The documentation provides a list of instance methods that your delegate method can or must implement.

Delegation isn't limited to Apple's classes. It's simple to add your own protocol declarations to your classes and use them to define callback vocabularies. Listing 1-3 creates the `FTPHostDelegate` protocol, which declares the `ftpHost` instance variable. When used, that object must implement all three (required) methods declared in the protocol. Protocols are an exciting and powerful part of Objective-C programming, letting you

From the Library of Eric Berna

create client classes that are guaranteed to support all the functionality required by the primary class.

Each class can support multiple delegate protocols. They appear in a comma-separated list in the angle brackets that follow the class name. In Listing 1-3, the `MergedTableController` class declares both the `UITableViewDelegate` protocol and the `UITableViewDataSource` protocol.

> **Note**
>
> If your application is built around a central table view, use `UITableViewController` instances to simplify table creation and use.

Listing 1-3 **Defining and Adding Delegate Protocol Declarations to a Class Definition**

```
@protocol FTPHostDelegate <NSObject>
- (void) percentDone: (NSString *) percent;
- (void) downloadDone: (id) sender;
- (void) uploadDone: (id) sender;
@end

@interface MergedTableController : UIViewController
    <UITableViewDelegate, UITableViewDataSource>
{
    id <FTPHostDelegate>   *ftpHost;
    SEL                    finishedAction;
}
@end
```

Target-Action

Target-actions are a lower-level way of redirecting user interactions. You encounter these almost exclusively for children of the `UIControl` class. With target-action, you tell the control to contact a given object when a specific user event takes place. For example, you'd specify which object to contact when users press a button.

Here is a typical example. This snippet defines a `UIBarButtonItem` instance, a typical button-like control used in iPhone toolbars. It sets the item's target to `self` (the owning view controller) and the action to `@selector(setHelvetica:)`. When tapped, it triggers a call to the defining object sending the `setHelvetica:` message:

```
UIBarButtonItem *helvItem = [[[UIBarButtonItem alloc]
    initWithTitle:@"Helvetica" style:UIBarButtonItemStyleBordered
    target:self action:@selector(setHelvetica:)] autorelease];
```

As you can see, the name of the method (`setHelvetica:`) is completely arbitrary. Target-actions do not rely on an established method vocabulary the way delegates do. In use, however, they work exactly the same way. The user does something (in this case, presses a button), and the target implements the selector to provide a meaningful response.

From the Library of Eric Berna

Whichever object defines this `UIBarButtonItem` instance must implement a `setHelvetica:` method. If it does not, the program crashes at runtime with an undefined method call error. Unlike delegates and their required protocols, there's no guarantee that `setHelvetica:` has been implemented at compile time. It's up to the programmer to make sure that the callback refers to an existing method. A cautious programmer will test the target before assigning a target-action pair with a given selector. Here's an example:

```
if ([self respondsToSelector:@selector(setHelvetica:)])
```

Standard `UIControl` target-action pairs always pass either zero, one, or two arguments. These optional arguments are the interaction object (such as a button, slider, or switch that has been manipulated) and a `UIEvent` object that represents the user's input. Your selector can choose to pass any or all of these. In this case, the selector uses one argument, the `UIBarButtonItem` instance that was pressed. This self-reference, where the triggered object is included with the call, enables you to build more general action code. Instead of building separate methods for `setHelvetica:`, `setGeneva:`, and `setCourier:`, you could create a single `setFontFace:` method to update a font based on which button the user pressed.

> **Note**
>
> It's easy to build your own `UIControl` subclasses. Chapter 9, "Building and Using Controls," demonstrates how. To build target-action into your own `UIControl`-style classes that aren't actual `UIControl` subclasses, add a target variable of type `id` (any object class) and an action variable of type `SEL` (method selector). At runtime, use `performSelector:withObject:` to send the method selector to the object. To use selectors without parameters, for example, `@selector(action)`, pass `nil` as the object and eliminate the colon (:) from after the selector name.

Notifications

In addition to delegates and target-actions, the iPhone uses yet another way to communicate about user interactions between your model and your view—and about other events, for that matter. **Notifications** enable objects in your application to talk among themselves, as well as to talk to other applications on your system. By broadcasting information, notifications enable objects to send state messages: "I've changed," "I've started doing something," or "I've finished."

Other objects might be listening to these broadcasts, or they might not. For your objects to "hear" a notification, they must register with a notification center and start listening for messages. iOS implements many kinds of notification centers. For App Store development, only `NSNotificationCenter` is of general use.

The `NSNotificationCenter` class is the gold standard for in-application notification. You can subscribe to any or all notifications with this kind of notification center and listen as your objects talk to each other. The notifications are fully implemented and can carry data as well as the notification name. This name + data implementation offers great flexibility, and you can use this center to perform complex messaging.

From the Library of Eric Berna

It's easy to subscribe to a notification center. Register your application delegate or, more typically, your primary `UIViewController` as an observer. You supply an arbitrary selector to be called when a notification arrives—in this case, `trackNotifications:`. The method takes one argument, an `NSNotification`. Ensure that your callback method hears all application notifications by setting the name and object arguments to `nil`:

```
[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(trackNotifications:) name:nil object:nil];
```

All notifications contain three data elements: the notification name, an associated object, and a user information dictionary. If you're unsure what notifications `UIKit` objects in your application produce, have your callback print out the name from all the notifications it receives—for example, `NSLog(@"%@", [notification name])`.

The kinds of notification vary by the task you are performing. For example, notifications when rotating an application include `UIApplicationWillChangeStatusBarOrientationNotification` and `UIDeviceOrientationDidChangeNotification`.

Make sure you implement whatever callback method you registered with. For this example, it gets called in this case for all program notifications, regardless of name or object. Setting these to `nil` when listening acts as a wildcard.

```
(void) trackNotifications: (NSNotification *) theNotification
{
    NSLog(@"%@", [theNotification name]);
    NSLog(@"%@", [theNotification object]);
}
```

Remember to disable notifications being sent to an object before you release that object. Tell the notification center to `removeObject:` with the object in question. This method call globally removes that object as an observer, without having to do so on a notification-by-notification basis.

Key-value observing (KVO) refers to a paradigm that is very similar to notifications. Instead of observing a notification center for a particular notification name, KVO triggers callbacks when an object's properties update to new values. See Chapter 2 for a longer KVO description, along with examples.

> **Note**
>
> The recipes in this book use `printf` and `CFShow` as well as `NSLog`. Each debug feedback method has its advantages and disadvantages. The former have the advantage of not printing out the date and time, which results in cleaner output. How you choose to log information is strictly a matter of taste. There are no wrong or right ways to put print statements into your program. See Chapter 2 for more details about logging information.

Blocks

Blocks do not provide an independent controller mechanism. Instead, blocks help transform the way that traditional callbacks operate. Instead of defining a standalone callback method, blocks allow you to pass handler behavior directly as parameters to system

From the Library of Eric Berna

requests. This helps avoid writing single-purpose completion methods by adding code directly to your original calls.

Introduced in the iOS 4 SDK, blocks are Objective-C's version of lambda expressions or "closures," which have been used for decades in languages such as Scheme, Lisp, Python, and Ruby. They create a way to define behavior without creating standard methods or functions, and allow you to work with that behavior just as you would with any other Objective-C object.

Consider the following notification request. Unlike the example shown in the previous section, this call does not require a `trackNotifications:`-style callback method. The block passed as the fourth parameter performs actions identical to that method, without adding extraneous single-purpose methods to your codebase. This helps localize code to where it best makes sense—in the context of registering for the notification. Doing this increases code readability and ultimately makes your applications more maintainable.

```
[[NSNotificationCenter defaultCenter]
    addObserverForName:nil
    object:nil
    queue:[NSOperationQueue mainQueue]
    usingBlock:^(NSNotification *theNotification)
    {
        NSLog(@"%@", [theNotification name]);
        NSLog(@"%@", [theNotification object]);
    }
];
```

Blocks can be used for notifications handling, as you saw in this example, for completion handlers, error handlers, enumeration, sorting, and with view animations. Best of all, blocks share local lexical scope with the method that defines them. That means blocks have full read access to local variables and parameters from the calling method.

Model

You're responsible for building all application semantics—the model portion of any MVC app. You create the callback methods triggered by your application's controller and provide the required implementation of any delegate protocol. For relatively simple programs, model details often are added to a `UIViewController` subclass. With more complex code, avoid shoehorning that implementation into a `UIViewController`. Custom-built classes can better help implement semantic details needed to support an application's model.

There's one place that the iOS SDK gives you a hand with meaning, and that's with data sources. Data sources enable you to fill `UIKit` objects with custom content.

Data Sources

A **data source** refers to any object that supplies another object with on-demand data. Some UI objects are containers without any native content. When you set another object as its data source, by assigning its `dataSource` property or via a call such as `[myUIObject setDataSource:applicationobject]`, you enable the UI object (specifically, a view) to

From the Library of Eric Berna

query the data source (the model) for data such as table cells for a given `UITableView`. Usually the data source pulls its data in from a file such as a local database, from a web service such as an XML feed, or from other scanned sources. `UITableView` and `UIPickerView` are two of the few Cocoa Touch classes that support or require data sources. You may find yourself using data sources for your own custom classes as well.

Data sources are like delegates in that you must implement their methods in another object, typically the `UITableViewController` that owns the table. They differ in that they create/supply objects rather than react to user interactions.

Listing 1-4 shows a typical data source method that returns a table cell for a given row. Like other data source methods, it enables you to separate implementation semantics that fill a given view from the Apple-supplied functionality that builds the view container.

Objects that implement data source protocols must declare themselves just as they would with delegate protocols. Listing 1-3 showed a class declaration that supports both delegate and data source protocols for `UITableViews`. Apple thoroughly documents data source protocols. You find this documentation in Xcode's Documentation window (Help > Documentation).

Listing 1-4 **Data Source Methods Fill Views with Meaningful Content**

```
// Return a cell for the ith row, labeled with its number
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:@"any-cell"];
    if (!cell) {
        cell = [[[UITableViewCell alloc] initWithFrame:CGRectZero
             reuseIdentifier:@"any-cell"] autorelease];
    }
    // Set up the cell
    cell.text = [tableTitles objectAtIndex:[indexPath row]];
    return cell;
}
```

The `UIApplication` Object

In theory, you'd imagine that the iPhone "model" component would center on the `UIApplication` class. In practice, it does not, at least not in any MVC sense of the word model. In the world of the Apple SDK, each program contains precisely one `UIApplication` instance, which you can refer to via `[UIApplication sharedApplication]`.

For the most part, unless you need to respond to application life-cycle events, open a URL in Safari, recover the key window, or adjust the look of the status bar, you can completely ignore `UIApplication`. Build your program around a custom application delegate class that is responsible for setting things up when the application launches and closing

From the Library of Eric Berna

things down when the application suspends. Otherwise, hand off the remaining model duties to methods in your custom `UIViewController` classes or to custom model classes.

> **Note**
>
> Use `[[UIApplication sharedApplication] keyWindow]` to locate your application's foremost window object.

One More Thing: Uncovering Data Source and Delegate Methods

In addition to monitoring notifications, message tracking can prove to be an invaluable tool. Add the following snippet to your class definitions to track all the optional methods your class can respond to:

```
-(BOOL) respondsToSelector:(SEL)aSelector
{
   printf("SELECTOR: %s\n", [NSStringFromSelector(aSelector)
       UTF8String]);
   return [super respondsToSelector:aSelector];
}
```

At runtime, this method reports any delegate and data source methods for objects you have assigned to these roles.

Summary

This chapter introduced you to the iOS SDK, the developer portal, and the iPhone application. You saw how to choose a developer program and how to create provisions. You explored typical iPhone applications—from projects and source files to the application end product—and learned about design limitations that should influence your development. Here are a few thoughts you may want to take away with you before leaving this chapter:

- Most developers end up choosing the $99/year standard iPhone Developer Program. This is the best, most general program to sign up for because it allows you to test on real devices and gives you access to App Store distribution channels.
- There are significant differences between each iPhone, iPad, and iPod touch platform. Make sure your applications understand those differences to provide the best end-user experience.
- Don't bend over backward to support ever-deprecating earlier models. Do some research and find out what platforms your core users will be using and allow yourself to write to that newer-generation hardware. Providing pre-4.x firmware support may gain you a few outlier sales, but is it really worth the extra overhead that you'll be encountering?
- Developing for mobile platforms is not the same as developing for desktop systems. Keep this cardinal rule in mind: fingers big, screen small, attention span short.

From the Library of Eric Berna

Although iPad screens offer a significantly larger interface, these principles remain true for all iOS platforms.

- The iPhone application bundle is much simpler and less structured than its Macintosh brother, although it shares many common features such as Info.plist files and .lproj folders.
- If you come from a Cocoa background, you'll be prepared, if not overprepared, to create iPhone applications. Familiarity with Objective-C and Cocoa best practices will put you on a firm development footing.
- If you're more comfortable using C++ than Objective-C, Apple has made it possible to create hybrid projects that leverage your C++ expertise with a minimum of Objective-C overhead.

From the Library of Eric Berna

This page intentionally left blank

From the Library of Eric Berna

2

Objective-C Boot Camp

iOS development centers on Objective-C. It is the standard programming language for both the iPhone family of devices 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. In this chapter, you learn basic Objective-C skills that help you get started with iOS programming. You learn about interfaces, methods, properties, memory management, and more. To round things out, this chapter takes you beyond Objective-C into Cocoa to show you the core classes you'll use in day-to-day programming and offers you concrete examples of how these classes work.

The Objective-C Programming Language

Objective-C is a strict superset of ANSI C. C is a compiled, procedural programming language developed in the early 1970s at AT&T. Objective-C, which was developed by Brad J. Cox in the early 1980s, adds object-oriented features to C. It blends C language constructs with concepts that originated in Smalltalk-80.

Smalltalk is one of the earliest and best-known object-oriented languages. It was developed at Xerox PARC as a dynamically typed interactive language. Cox layered Smalltalk's object and message passing system on top of standard C to create his new language. This approach allowed programmers to continue using familiar C-language development while accessing object-based features from within that language. In the late 1980s, Objective-C was adopted as the primary development language for the NeXTStep operating system by Steve Jobs's startup computer company NeXT. NeXTStep became both the spiritual and literal ancestor of OS X.

Objective-C 2.0 was released in October 2007 along with OS X Leopard, introducing many new features like properties and fast enumeration. In 2010, Apple updated Objective-C to add blocks, a C-language extension that provides anonymous functions, letting developers treat code like objects. In the summer of 2011, Apple introduced automated reference counting, or ARC. This extension greatly simplified development, allowing programmers to focus on creating application semantics rather than worry about memory management.

From the Library of Eric Berna

Object-oriented programming brings features to the table that are missing in standard C. Objects refer to data structures that are associated with a publicly declared list of function calls. Every object in Objective-C has instance variables, which are the fields of the data structure, and methods, which are the function calls the object can execute. Object-oriented code uses these objects and methods to introduce programming abstractions that increase code readability and reliability.

Object-oriented programming lets you build reusable code units that can be decoupled from the normal flow of procedural development. Instead of relying on process flow, object-oriented programs are developed around the smart data structures provided by objects and their methods. Cocoa Touch on iOS and Cocoa on Mac OS X offer a massive library of these smart objects. Objective-C unlocks that library and lets you build on Apple's toolbox to create effective, powerful applications with a minimum of effort and code.

> **Note**
>
> iOS Cocoa Touch class names that start with NS, such as `NSString` and `NSArray`, harken back to NeXT. NS stands for NeXTStep, the operating system that ran on NeXT computers.

Classes and Objects

Objects form the heart of object-oriented programming. You define objects by building classes, which act as object creation templates. In Objective-C, a class definition specifies how to build new objects that belong to the class. So to create a "widget" object, you define the `Widget` class and then use that class to create new objects on demand.

Each class lists its instance variables and methods in a public header file using the standard C .h convention. For example, you might define a `Car` object like the one shown in Listing 2-1. The Car.h header file shown here contains the interface that declares how a `Car` object is structured.

Listing 2-1 **Declaring the `Car` Interface (Car.h)**

```
#import <Foundation/Foundation.h>
@interface Car : NSObject
{
    int year;
    NSString *make;
    NSString *model;
}
- (void) setMake:(NSString *) aMake andModel:(NSString *) aModel
    andYear: (int) aYear;
- (void) printCarInfo;
- (int) year;
@end
```

From the Library of Eric Berna

Note that all classes in Objective-C should be capitalized (`Car`) and that their methods should not be. In Listing 2-1, the methods are the declarations that start with a minus sign toward the bottom of the listing. Objective-C uses camel case. Instead of creating `identifiers_like_this`, Objective-C traditionally prefers `identifiersLikeThis`. You see that in Listing 2-1 with the first two method names.

In Objective-C, the `@` symbol indicates certain keywords. The two items shown here (`@interface` and `@end`) delineate the start and end of the class interface definition. This class definition describes an object with three instance variables: `year`, `make`, and `model`. These three items are declared between the braces at the start of the interface.

The `year` instance variable is declared as an integer (using `int`). Both `make` and `model` are strings, specifically instances of `NSString`. Objective-C uses this object-based class for the most part rather than the byte-based C strings defined with `char *`. As you see throughout this book, `NSString` offers far more power than C strings. With this class, you can find out a string's length, search for and replace substrings, reverse strings, retrieve file extensions, and more. These features are all built in to the base Cocoa Touch object library.

This class definition also declares three public methods. The first is called `setMake:andModel:andYear:`. This entire three-part declaration, including the colons, is the name of that single method. That's because Objective-C places parameters inside the method name, using a colon to indicate each parameter. In C, you'd use a function such as `setProperties(char *c1, char *c2, int i)`. Objective-C's approach, although heftier than the C approach, provides much more clarity and self-documentation. You don't have to guess what `c1`, `c2`, and `i` mean because their use is declared directly within the name:

```
[myCar setMake:c1 andModel:c2 andYear:i];
```

The three methods are typed as `void`, `void`, and `int`, respectively. As in C, these refer to the type of data returned by the method. The first two do not return data; the third returns an integer. In C, the equivalent function declaration to the second and third method would be `void printCarInfo()` and `int year()`.

Using Objective-C's method-name-interspersed-with-arguments approach can feel odd to new programmers but quickly becomes a much-loved feature. There's no need to guess which argument to pass when the method name itself tells you what items go where. In Objective-C, method names are also interchangeably called "selectors." You see this a lot in iOS programming, especially when you use calls to `performSelector:`, which lets you send messages to objects at runtime.

Notice that this header file uses `#import` to load headers rather than `#include`. Importing headers in Objective-C automatically skips files that have already been added. This lets you add duplicate `#import` directives to your various source files without penalties.

From the Library of Eric Berna

> **Note**
> The code for this example—and all the examples in this chapter—is found in the sample code for this book. See the Preface for details about downloading the book's sample code from the Internet.

Creating Objects

To create an object, you tell Objective-C to allocate the memory needed for the object and return a pointer to that object. Because Objective-C is an object-oriented language, its syntax looks a little different from regular C. Instead of just calling functions, you ask an object to do something. This takes the form of two elements within square brackets, the object receiving the message followed by the message itself:

```
[object message]
```

Here, the source code sends the message `alloc` to the `Car` class and then sends the message `init` to the newly allocated `Car` object. This nesting is typical in Objective-C.

```
Car *myCar = [[Car alloc] init];
```

The "allocate followed by `init`" pattern you see here represents the most common way to instantiate a new object. The class `Car` performs the `alloc` method. It allocates a new block of memory sufficient to store all the instance variables listed in the class definition, zeroes out any instance variables, and returns a pointer to the start of the memory block. The newly allocated block is called an "instance" and represents a single object in memory.

Some classes, like views, use specialized initializers such as `initWithFrame:`. You can write custom ones such as `initWithMake:andModel:andYear:`. The pattern of allocation followed by initialization to create new objects holds universally. You create the object in memory and then you preset any critical instance variables.

Memory Allocation

In this example, the memory allocated is 16 bytes long. Both `make` and `model` are pointers, as indicated by the asterisk. In Objective-C, object variables point to the object itself. The pointer is 4 bytes in size. So `sizeof(myCar)` returns 4. The object consists of two 4-byte pointers, one integer, plus one additional field that does not derive from the `Car` class.

That extra field is from the `NSObject` class. Notice `NSObject` at the right of the colon next to the word `Car` in the class definition of Listing 2-1. `NSObject` is the parent class of `Car`, and `Car` inherits all instance variables and methods from this parent. That means that `Car` is a type of `NSObject`, and any memory allocation needed by `NSObject` instances is inherited by the `Car` definition. So that's where the extra 4 bytes come from.

The final size of the allocated object is 16 bytes in total. That size includes two 4-byte `NSString` pointers, one 4-byte `int`, and one 4-byte allocation inherited from `NSObject`.

With the manual retain/release (MRR) compiler (that is, compiled with `-fno-objc-arc`), you can print out the size of objects by dereferencing them and using C's `sizeof()` function. The following code snippet uses standard C `printf` statements to

From the Library of Eric Berna

send text information to the console. `printf` commands work just as well in Objective-C as they do in ANSI C.

```
NSObject *object = [[NSObject alloc] init];
Car *myCar = [[Car alloc] init];

// This returns 4, the size of an object pointer
printf("object pointer: %d\n", sizeof(object));

// This returns 4, the size of an NSObject object
printf("object itself: %d\n", sizeof(*object));

// This returns 4, again the size of an object pointer
printf("myCar pointer: %d\n", sizeof(myCar));

// This returns 16, the size of a Car object
printf("myCar object: %d\n", sizeof(*myCar));
```

You cannot use this `sizeof()` check with the default ARC compiler. Instead, use an Objective-C runtime function to retrieve the same information. (Make sure to include `<objc/objc-runtime.h>` to compile.) The `class_getInstanceSize()` function returns the size of any object allocated by a given class, allowing you to see how much space any object instance occupies in memory:

```
printf("myCar object: %d\n", (int) class_getInstanceSize([myCar class]);
```

> **Note**
>
> As ARC gains traction in the iOS developer community, the phrase Manual Reference Counting (MRC) is starting to compete with Manual Retain Release (MRR). This chapter uses MRR throughout, due to the standards when the material was being prepared. I'm actually rooting for MRC over MRR, as I think it better describes the technology. Time will tell.

Releasing Memory

In C, you allocate memory with `malloc()` or a related call and free that memory with `free()`. In Objective-C, how you release that memory depends on whether or not you're using automated reference counting (ARC) compiler features or manual retain/release (MRR).

In both scenarios, you allocate memory with `alloc`. Objective-C also lets you allocate memory a few other ways, such as by copying other objects. With ARC, you never explicitly free memory. The compiler takes care of that for you. With MRR, you are responsible for releasing memory when you are done using it; free an object by sending it the `release` message:

```
[object release];
[myCar release];
```

From the Library of Eric Berna

This chapter discusses both ARC and MRR. If you are new to iOS 5 development and will not be working with any legacy code, you might assume you should only focus on using ARC code. The realities of iOS development challenge that assumption. Anyone who works with iOS should understand how both ARC and manual memory management work.

Understanding Retain Counts with MRR

Under MRR, releasing memory is a little more complicated than in standard C. That's because Objective-C uses a reference-counted memory system. Each object in memory has a retain count associated with it. You can see that retain count by sending `retainCount` to the object in any project that's not compiled with ARC. (Under ARC, you may not use `retainCount` directly. Similar prohibitions exist for `retain`, `release`, and `autorelease`.) Never rely on using `retainCount` in software deployment in the real world. It's used here only as a tutorial example to help demonstrate how retains work.

Every object is created with a retain count of 1. Sending `release` reduces that retain count by 1. When the retain count for an object reaches 0, or more accurately, when it is about to reach 0 by sending the release message to an object with a retain count of 1, it is released into the general memory pool.

```
Car *myCar = [[Car alloc] init];


// The retain count is 1 after creation
printf("The retain count is %d\n", [myCar retainCount]);

// This would reduce the retain count to 0, so it is freed instead
[myCar release];

// This causes an error. The object has already been freed
printf("Retain count is now %d\n", [myCar retainCount]);
```

Sending messages to freed objects will crash your application; you're addressing memory you no longer own. When the second `printf` executes, the `retainCount` message is sent to the already-freed `myCar`. This creates a memory access violation, terminating the program. As a general rule in MRR applications, it's good practice to assign instance variables to `nil` after the final release that deallocates the object. This prevents the `FREED(id)` error you see here, when you access an already-freed object:

```
The retain count is 1
objc[10754]: FREED(id): message retainCount sent to freed
object=0xd1e520
```

As a developer, you must manage your MRR objects. Keep them around for the span of their use and free their memory when you are finished. Basic memory management strategies and an ARC overview follow later in this chapter.

From the Library of Eric Berna

Methods, Messages, and Selectors

In standard C, you'd perform two function calls to allocate and initialize data. Here is how that might look, in contrast to Objective-C's `[[Car alloc] init]` statement:

```
Car *myCar = malloc(sizeof(Car));
init(myCar);
```

Objective-C doesn't use `function_name(arguments)` syntax. Instead, you send messages to objects using square brackets. Messages tell the object to perform a method. It is the object's responsibility to implement that method and produce a result. The first item within the brackets is the receiver of the message; the second item is a method name, and possibly some arguments to that method that together define the message you want sent. In C, you might write

```
printCarInfo(myCar);
```

but in Objective-C, you say:

```
[myCar printCarInfo];
```

Despite the difference in syntax, methods are basically functions that operate on objects. They are typed using the same types available in standard C. Unlike function calls, Objective-C places limits on who can implement and call methods. Methods belong to classes. And the class interface defines which of these are declared to the outside world.

Undeclared Methods

With ARC, the LLVM compiler will not allow you to send a message to an object that does not declare that method selector. Sending `printCarInfo` to an array object, for example, causes a runtime error and crashes the program. Under MRR, the compiler issues a warning about the method call. Under the ARC compiler's default settings, the call will not compile as the built-in static analyzer raises an objection (see Figure 2-1). Only objects that implement a given method can respond to the message properly and execute the code that was requested. Here's what happens when the MRR version is compiled and run, with the warnings ignored:

```
2009-05-11 09:04:31.978 HelloWorld[419:20b] *** -[NSCFArray printCarInfo]:
unrecognized selector sent to instance 0xd14e80
2009-05-11 09:04:31.980 HelloWorld[419:20b] *** Terminating app due to uncaught
exception
'NSInvalidArgumentException', reason: '*** -[NSCFArray
printCarInfo]: unrecognized selector sent to instance 0xd14e80'
```

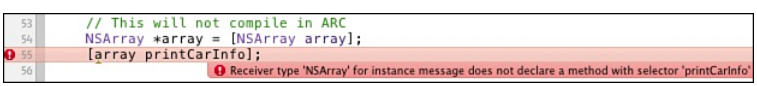


Figure 2-1 Xcode's ARC compilation will not compile method calls that do not appear to be implemented by the receiver.

From the Library of Eric Berna

With MRR's default settings, these warnings do not make the compilation fail, and it's possible that this code could run without error if `NSArray` implemented `printCarInfo` and did not declare that implementation in its published interface. Because `NSArray` does not, in fact, implement this method, running this code produces a runtime crash.

Pointing to Objects

Objective-C lets you point to the same kind of object in several different ways. Although `array` was declared as a statically typed `(NSArray *)` object, that object uses the same internal object data structures as an object declared as `id`. The `id` type can point to any object, regardless of class, and is equivalent to `(NSObject *)`. The following assignment is valid and does not generate any warnings at compile time:

```
NSArray *array = [NSArray array];
id untypedVariable = array; // This assignment is valid
```

To further demonstrate, consider a mutable array. The `NSMutableArray` class is a subclass of `NSArray`. The mutable version offers arrays that you can change and edit. Creating and typing a mutable array but assigning it to an array pointer compiles without error. Although `anotherArray` is statically typed as `NSArray`, creating it in this way produces an object at runtime that contains all the instance variables and behaviors of the mutable array class.

The problem with the following code is not the assignment, it's the method call. Although `anotherArray` is allocated and initialized as a mutable array, it's assigned to a normal array. The static analyzer complains at the `addObject:` method.

```
NSArray *anotherArray = [NSMutableArray array];
// This mutable-only method call will cause an error
[anotherArray addObject:@"Hello World"];
```

Although assigning a child class object to a pointer of a parent class generally works at runtime, it's far more dangerous to go the other way. A mutable array is a kind of array. It can receive all the messages that arrays do. Not every array, on the other hand, is mutable. Sending the `addObject:` message to a regular array is lethal. Doing so bombs at runtime, because arrays do not implement that method:

```
NSArray *standardArray = [NSArray array];
NSMutableArray *mutableArray;
// This line produces a warning
mutableArray = standardArray;
// This will bomb at runtime
[mutableArray addObject:@"Hello World"];
```

The code seen here produces just one warning, at the line where the standard array object is assigned to the mutable array pointer, namely "Incompatible pointer types." Parent-to-child assignments do not generate this warning. Child-to-parent assignments do. So do assignments between completely unrelated classes. Do not ignore this warning; fix your code. Otherwise, you're setting yourself up for a runtime crash.

From the Library of Eric Berna

> **Note**
>
> In Xcode, you can set the compiler to treat warnings as errors. Because Objective-C is so dynamic, the compiler cannot catch every problem that might crash at runtime the way static language compilers can. Pay special attention to warnings and try to eliminate them.

Inheriting Methods

Objects inherit method implementations as well as instance variables. A `Car` is a kind of `NSObject`, so it can respond to all the messages that an `NSObject` responds to. That's why `myCar` can be allocated and initialized with `alloc` and `init`. These methods are defined by `NSObject`. Therefore, they can be used to create and initialize any instance of `Car`, which is derived from the `NSObject` class.

Similarly, `NSMutableArray` instances are a kind of `NSArray`. All array methods can be used by mutable arrays, their child class. You can count the items in the array, pull an object out by its index number, and so forth.

A child class may override a parent's method implementation, but it can't negate that the method exists. Child classes always inherit the full behavior and state package of their parents.

Declaring Methods

As Listing 2-1 showed, a class interface defines the instance variables and methods that a new class adds to its parent class. This interface is normally placed into a header file, which is named with an .h extension. The interface from Listing 2-1 declared three methods:

```
- (void) setMake:(NSString *) aMake andModel:(NSString *) aModel
    andYear: (int) aYear;
- (void) printCarInfo;
- (int) year;
```

These three methods, respectively, return `void`, `void`, and `int`. Notice the dash that starts the method declaration. It indicates that the methods are implemented by object instances. For example, you call `[myCar year]` and not `[Car year]`. The latter sends a message to the `Car` class rather than an actual car object. A discussion about class methods (indicated by "+" rather than "-") follows later in this section.

Method calls can be complex. The following invocation sends a method request with three parameters. The parameters are interspersed inside the method invocation. The name for the method (that is, its selector) is `setMake:andModel:andYear:`. The three colons indicate where parameters should be inserted. The types for each parameter are specified in the interface after the colons: `(NSString *)`, `(NSString *)`, and `(int)`. Because this method returns `void`, the results are not assigned to a variable:

```
[myCar setMake:@"Ford" andModel:@"Prefect" andYear:1946];
```

From the Library of Eric Berna

Implementing Methods

Together, a method file and a header file pair store all the information needed to implement a class and announce it to the rest of an application. The implementation section of a class definition provides the code that implements functionality. This source is usually placed in an .m (for "method") file.

Listing 2-2 shows the implementation for the `Car` class example. It codes all three methods declared in the header file from Listing 2-1 and adds a fourth. This extra method redefines `init`. The `Car` version of `init` sets the `make` and `model` of the car to `nil`, which is the NULL pointer for Objective-C objects. It also initializes the `year` of the car to 1901.

The special variable `self` refers to the object that is implementing the method. That object is also called the "receiver" (that is, the object that receives the message). This variable is made available by the underlying Objective-C runtime system. In this case, `self` refers to the current instance of the `Car` class. Calling `[self message]` tells Objective-C to send a message to the object that is currently executing the method.

Several things are notable about the `init` method seen here. First, the method returns a value, which is typed to `(id)`. As mentioned earlier in this chapter, the `id` type is more or less equivalent to `(NSObject *)`, although it's theoretically slightly more generic than that. It can point to any object of any class (including Class objects themselves). You return results the same way you would in C, using `return`. The goal of `init` is to return a properly initialized version of the receiver via `return self`.

Second, the method calls `[super init]`. This tells Objective-C to send a message to a different implementation—namely the one defined in the object's superclass. The superclass of `Car` is `NSObject`, as shown in Listing 2-1. This call says, "Please perform the initialization that is normally done by my parent class before I add my custom behavior." Calling a superclass's implementation before adding new behavior demonstrates an important practice in Objective-C programming.

Finally, notice the check for `if (!self)`. In rare instances, memory issues arise. In such a case, the call to `[super init]` returns `nil`. If so, this `init` method returns before setting any instance variables. Because a `nil` object does not point to allocated memory, you cannot access instance variables within `nil`.

As for the other methods, they use `year`, `make`, and `model` as if they were locally declared variables. As instance variables, they are defined within the context of the current object and can be set and read as shown in this example. The `UTF8String` method that is sent to the `make` and `model` instance variables converts these `NSString` objects into C strings, which can be printed using the `%s` format specifier.

> **Note**
>
> You can send nearly any message to `nil` (for example, `[nil anyMethod]`). The result of doing so is, in turn, `nil`. (Or, more accurately, 0 casted as `nil`.) In other words, there is no effect. This behavior lets you nest method invocations with a failsafe should any of the individual methods fail and return `nil`. If you were to run out of memory during an allocation

From the Library of Eric Berna

with `[[Car alloc] init]`, the `init` message would be sent to `nil`, allowing the entire `alloc/init` request to return `nil` in turn.

Listing 2-2 **The `Car` Class Implementation (Car.m)**

```
#import "Car.h"

@implementation Car
- (id) init
{
    self = [super init];
    if (!self) return nil;

    // the make and model are initialized to nil by default
    year = 1901;

    return self;
}

- (void) setMake:(NSString *) aMake andModel:(NSString *) aModel
    andYear: (int) aYear
{
    // Note that this does not yet handle memory management properly
    // The Car object does not retain these items, which may cause
    // memory errors down the line
    make = aMake;
    model = aModel;
    year = aYear;
}

- (void) printCarInfo
{
    if (!make) return;
    if (!model) return;

    printf("Car Info\n");
    printf("Make: %s\n", [make UTF8String]);
    printf("Model: %s\n", [model UTF8String]);
    printf("Year: %d\n", year);
}

- (int) year
{
    return year;
}
@end
```

From the Library of Eric Berna

Class Methods

Class methods are defined using a plus (+) prefix rather than a hyphen (-). They are declared and implemented in the same way as instance methods. For example, you might add the following method declaration to your interface:

```
+ (NSString *) motto;
```

Then you could code it up in your implementation:

```
+ (NSString *) motto
{
    return @"Ford Prefects are Mostly Harmless";
}
```

Class methods differ from instance methods in that they generally cannot use state. They are called on the Class object itself, which does not have access to instance variables. That is, they have no access to an instance's instance variables (hence the name) because those elements are only created when instantiated objects are allocated from memory.

So why use class methods at all? The answer is threefold. First, class methods produce results without having to instantiate an actual object. This motto method produces a hard-coded result that does not depend on access to instance variables. Convenience methods such as this often have a better place as classes rather than instance methods.

You might imagine a class that handles geometric operations. The class could implement a conversion between radians and degrees without needing an instance—for example, `[GeometryClass convertAngleToRadians:theta]`. Simple C functions declared in header files also provide a good match to this need.

The second reason is that class methods can provide access to a singleton. **Singletons** refer to statically allocated instances. The iOS SDK offers several of these. For example, `[UIApplication sharedApplication]` returns a pointer to the singleton object that is your application. `[UIDevice currentDevice]` retrieves an object representing the hardware platform you're working on.

Combining a class method with a singleton lets you access that static instance anywhere in your application. You don't need a pointer to the object or an instance variable that stores it. The class method pulls that object's reference for you and returns it on demand.

Third, class methods tie into memory management schemes, for both ARC and MRR compilation. Consider allocating a new `NSArray`. You do so via `[[NSArray alloc] init]`, or you can use `[NSArray array]`. This latter class method returns an array object that has been initialized and set for autorelease. As you read about later in this chapter, Apple has provided a standard about class methods that create objects. They always return those objects to you already autoreleased. Because of that, this class method pattern is a fundamental part of the standard iOS memory management system.

From the Library of Eric Berna

Fast Enumeration

Fast enumeration was introduced in Objective-C 2.0 and offers a simple and elegant way to enumerate through collections such as arrays and sets. It adds a for-loop that iterates through the collection using concise `for/in` syntax. The enumeration is very efficient, running quickly. Attempts to modify the collection as it's being enumerated raise a run-time exception; don't do that.

```
NSArray *colors = [NSArray arrayWithObjects:
    @"Black", @"Silver", @"Gray", nil];
for (NSString *color in colors)
    printf("Consider buying a %s car", [color UTF8String]);
```

> **Note**
>
> Use caution when using a method such as `arrayWithObjects:` or `dictionaryWithKeysAndValues:`. These can be unnecessarily error-prone. A common error occurs when one of the mid-list arguments evaluates to `nil`. This is especially easy to miss for `NSDictionary`s, which can have missing key-value pairs as a result. Developers often use this method with instance variables without first checking whether these values are non-`nil`, which can cause runtime errors. Another common mistake is to forget the final "sentinel" `nil` at the end of your arguments. This missing `nil` may not be caught at compile time but will cause runtime errors as well.

Class Hierarchy

In Objective-C, each new class is derived from an already-existing class. The `Car` class described in Listings 2-1 and 2-2 is formed from `NSObject`, the root class of the Objective-C class tree. Each subclass adds or modifies state and behavior that it inherits from its parent, also called its "superclass." The `Car` class adds several instance variables and methods to the vanilla `NSObject` it inherits.

Figure 2-2 shows some of the classes found in the iOS SDK and how they relate to each other in the class hierarchy. Strings and arrays descend from `NSObject`, as does the `UIResponder` class. `UIResponder` is the ancestor of all onscreen iOS elements. Views, labels, text fields, and sliders are children, grandchildren, or other descendants of `UIResponder` and `NSObject`.

Every class other than `NSObject` descends from other classes. `UITextField` is a kind of `UIControl`, which is in turn a kind of `UIView`, which is a `UIResponder`, which is an `NSObject`. Building into this object hierarchy is what Objective-C is all about. Child classes can do the following:

- Add new instance variables that are not allocated by their parent, also called the superclass. The `Car` class adds three: the `make` and `model` strings, and the `year` integer.
- Add new methods that are not defined by the parent. `Car` defines several new methods, letting you set the values of the instance variables and print out a report about the car.

From the Library of Eric Berna

- Override methods that the parents have already defined. The `Car` class's `init` method overrides `NSObject`'s version. When sent an `init` message, a `car` object runs its version, not `NSObject`'s. At the same time, the code for `init` makes sure to call `NSObject`'s `init` method via `[super init]`. Referencing a parent's implementation, while extending that implementation, is a core part of the Objective-C design philosophy.

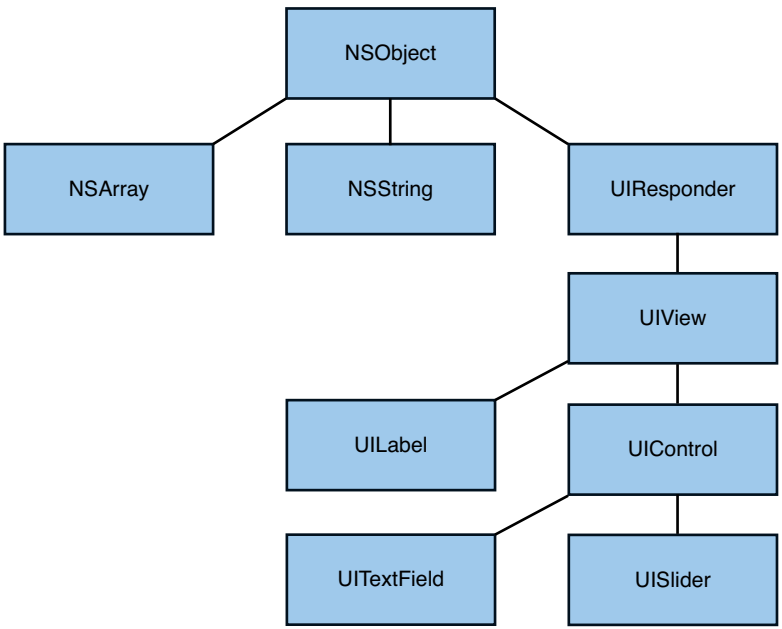


Figure 2-2 All Cocoa Touch classes are descended from `NSObject`, the root of the class hierarchy tree.

Logging Information

Now that you've read the basics about classes and objects, it's important to understand how to log information about them. In addition to `printf`, Objective-C offers a fundamental logging function called `NSLog`. This function works like `printf` and uses a similar format string, but it outputs to `stderr` instead of `stdout`. `NSLog` uses an `NSString` format string rather than a C string one.

`NSStrings` are declared differently than C strings. They are prepended with the `@` (at) symbol. A typical `NSString` looks `@"like this"`; the equivalent C string looks `"like this"`, omitting the `@`. Whereas C strings refer to a pointer to a string of bytes, `NSStrings` are objects. You can manipulate a C string by changing the values stored in

From the Library of Eric Berna

each byte. NSStrings are immutable; you cannot access the bytes to edit them, and the actual string data is not stored within the object.

```
// This is 12 bytes of addressable memory
printf("%d\n", sizeof("Hello World"));

// This 4-byte object points to non-addressable memory
NSString *string = @"Hello World";
printf("%d\n", sizeof(*string)); // Only with -fno-objc-arc
```

In addition to using the standard C format specifiers, NSLog introduces an object specifier, %@, which lets you print objects. It does so by sending an object the description message. It is then the object's job to return a human-readable description of itself. This behavior allows you to transform

```
printf("Make: %s\n", [make UTF8String]);
```

into

```
NSLog(@"Make: %@", make);
```

Table 2-1 shows some of the most common format specifiers. This is far from an exhaustive list, so consult Apple's String Programming Guide for Cocoa for more details.

Table 2-1 **Common String Format Specifiers**

Specifier	Meaning
%@	Objective-C object using the description or descriptionWithLocale: results
%%	The "%" literal character
%d, %i, %D	Signed integer (32-bit). Use %qi for signed 64-bit integers.
%u, %U	Unsigned integer (32-bit). Use %qu for unsigned 64-bit integers.
%hi	Signed short integer (16-bit)
%hu	Unsigned short integer (16-bit)
%f	Floating-point (64-bit)
%e	Floating-point printed using exponential (scientific) notation (64-bit)
%c	Unsigned char (8-bit)
%C	Unicode char (16-bit)
%s	Null-terminated char array (string, 8-bit)
%S	Null-terminated Unicode char array (16-bit)
%p	Pointer address using lowercase hex output, with a leading 0x
%x	Lowercase unsigned hex (32-bit)
%X	Uppercase unsigned hex (32-bit)

From the Library of Eric Berna

Notice that NSLog does not require a hard-coded return character. It automatically appends a new line when used. What's more, it adds a timestamp to every log, so the results of the NSLog invocation shown previously look something like this:

```
2011-05-07 14:19:08.792 HelloWorld[11197:20b] Make: Ford
```

Nearly every object converts itself into a string via the description message. NSLog uses description to show the contents of objects formatted with %@. This returns an NSString with a textual description of the receiver object. You can describe objects outside of NSLog by sending them the same description method. This is particularly handy for use with printf and fprintf, which cannot otherwise print objects:

```
fprintf(stderr, "%s\n", [[myCar description] UTF8String]);
```

You can combine NSLog's object formatting features with printf's no-prefix presentation by implementing a simple NS-styled print utility such as the following function. This function automatically adds a carriage return, as NSLog does but printf does not. It also handles printing objects, which printf natively does not. Adjust this accordingly for your own needs. As with NSLog, the data this function sends to standard out can be viewed in the Xcode debugging console.

```
void nsprintf(NSString *formatString,...)
{
    va_list arglist;
    if (formatString)
    {
        va_start(arglist, formatString);
        {
            NSString *outstring = [[NSString alloc]
                initWithFormat:formatString arguments:arglist];
            fprintf(stderr, "%s\n", [outstring UTF8String]);
            // [outstring release]; // uncomment if not using ARC
        }
        va_end(arglist);
    }
}
```

Cocoa touch offers an NSStringFrom family of functions. These provide a simple way to convert structures to printable strings suitable for logging. For example, NSStringFromCGRect() creates a string from a CGRect structure and NSStringFromCGAffineTransform() builds its string formatted to contain the data from a Core Graphics affine transform.

Basic Memory Management

Memory management comes down to two simple rules. At creation, every object has a retain count of 1. At release, every object has (or, more accurately, is about to have) a retain count of 0. When using manual retain/release (MRR), it is up to you as a developer to

From the Library of Eric Berna

manage an object's retention over its lifetime. With automated reference counting (ARC), the compiler does most of the work for you. Here are two quick-and-dirty guides that explain how each approach works.

Managing Memory with MRR

Under MRR, your role as a coder is to ensure that every object you allocate moves from the start to the finish of its life cycle without being prematurely released and to guarantee that it does finally get released when it is time to do so. Complicating matters is Objective-C's MRR autorelease pool. If some objects are autoreleased and others must be released manually, how do you best control your objects? Here's a quick-and-dirty guide to getting your memory management right.

Creating Objects

Any time you create an object using the `alloc/init` pattern with the MRR compiler, you build it with a retain count of 1. It doesn't matter which class you use or what object you build, `alloc/init` produces a +1 count.

```
id myObject = [[SomeClass alloc] init];
```

For locally scoped variables, if you do not release the object before the end of a method, the object leaks. Your reference to that memory goes away, but the memory itself remains allocated. The retain count remains at +1.

```
- (void) leakyMRRMethod
{
   // This is leaky in MRR
    NSArray *array = [[NSArray alloc] init];
}
```

The proper way to use an `alloc/init` pattern is to create, use, and then release. Releasing brings the retain count down to 0. When the method ends, the object is deallocated.

```
- (void) properMRRMethod
{
    NSArray *array = [[NSArray alloc] init];
    // use the array here
    [array release];
}
```

Autoreleasing objects do not require an explicit `release` statement for locally scoped variables. (In fact, avoid doing so to prevent double-free errors that will crash your program.) Sending the autorelease message to an object marks it for autorelease. When the autorelease pool drains at the end of each event loop, it sends `release` to all the objects it owns.

```
- (void) anotherProperMRRMethod
{
    NSArray *array = [[[NSArray alloc] init] autorelease];
```

From the Library of Eric Berna

```
    // This won't crash the way release would
    printf("Retain count is %d\n",
        [array retainCount]); // don't use retainCount in real code!
    // use the array here
}
```

By convention, object-creation class methods, both MRR and ARC, return an autoreleased object. The `NSArray` class method array returns a newly initialized array that is already set for autorelease. The object can be used throughout the method, and its release is handled when the autorelease pool drains.

```
- (void) yetAnotherProperMethod
{
    NSArray *array = [NSArray array];
    // use the array here
}
```

At the end of this method, the autoreleased array can return to the general memory pool.

Creating Autoreleased Objects

As a rule, whenever you ask another method to create an object, it's good programming practice to return that object autoreleased. Doing so consistently lets you follow a simple rule: "If I didn't allocate it, then it was built and returned to me as an autoreleasing object." You must do this manually in MRR. In ARC, the compiler is smart enough to autorelease the object for you.

```
- (Car *) fetchACarInMRR
{
    Car *myCar = [[Car alloc] init];
    return [myCar autorelease];
}
```

This holds especially true for class methods. By convention all class methods that create new objects return autoreleasing objects. These are generally referred to as "convenience methods." Any object that you yourself allocate is not set as autorelease unless you specify it yourself.

```
// This is not autoreleased in MRR
Car *car1 = [[Car alloc] init];

// This is autoreleased in MRR
Car *car2 = [[[Car alloc] init] autorelease];

// By convention, this *should* be an autoreleased object in MRR
Car *car3 = [Car car];
```

To create a convenience method at the class level, make sure to define the class with the + prefix instead of - and return the object after sending `autorelease` to it.

```
+ (Car *) car
{
```

From the Library of Eric Berna

```
    // MRR requires an explicit autorelease call
    return [[[Car alloc] init] autorelease];
}
```

Autoreleasing Object Lifetime

So how long can you use an autoreleasing object? What guarantees do you have? The hard-and-fast rule is that the object is yours until the next item in the event loop gets processed. The event loop is triggered by user touches, by button presses, by "time passed" events, and so forth. In human reckoning these times are impossibly short; in the iOS SDK device's processor frame of reference, they're quite large. As a more general rule, you can assume that an autoreleased object should persist throughout the duration of your method call.

Once you return from a method, guarantees go out the window. When you need to use an array beyond the scope of a single method or for extended periods of time (for example, you might start a custom run-loop within a method, prolonging how long that method endures), the rules change. In MRR, you must retain autoreleasing objects to increase their count and prevent them from getting deallocated when the pool drains; when the autorelease pool calls release on their memory, they'll maintain a count of at least +1. In ARC, assigning an autoreleasing object to any default (`strong`) variable or property retains that object for the lifetime of that scoping.

Never rely on an object's `retainCount` to keep track of how often it has already been retained. If you want to make absolutely sure you own an object, then retain it, use it, and release it when you're done. If you're looking at anything other than your own object's relative retain counts and matching releases, you're going to run into systemic development errors.

> **Note**
>
> Avoid assigning properties to themselves, especially in MRR (for example, `myCar.colors = myCar.colors`). The release-then-retain behavior of properties may cause the object to deallocate before it can be reassigned and re-retained.

Retaining Autoreleasing Objects

In MRR, you can send `retain` to autoreleasing objects just like any other object. Retaining autoreleasing objects allows them to persist beyond a single method. Once retained, an autoreleased object is just as subject to memory leaks as one that you created using `alloc/init`. For example, retaining an object that's scoped to a local variable might leak, as shown here:

```
- (void)anotherLeakyMRRMethod
{
    // After returning, you lose the local reference to
    // array and cannot release.
    NSArray *array = [NSArray array];
    [array retain];
}
```

From the Library of Eric Berna

Upon creation, `array` has a retain count of +1. Sending `retain` to the object brings that retain count up to +2. When the method ends and the autorelease pool drains, the object receives a single release message; the count returns to +1. From there, the object is stuck. It cannot be deallocated with a +1 count and with no reference left to point to the object, it cannot be sent the final release message it needs to finish its life cycle. This is why it's critical to build references to retained objects.

By creating a reference, you can both use a retained object through its lifetime and be able to release it when you're done. Set references via an instance variable (preferred) or a static variable defined within your class implementation. If you want to keep things simple and reliable in MRR, use retained properties built from those instance variables. A discussion of how retained properties work and why they provide a solution of choice for developers follows later in this chapter.

Managing Memory with ARC

Automatic reference counting (ARC) simplifies memory management by automating the way objects are retained and released. By handling your memory for you, ARC allows you to focus your effort on code semantics, reducing the time and overhead you devote to thinking about memory management issues.

To create an object in ARC, either allocate and initialize it or use a class convenience method. You do not have to retain your object, autorelease your object, or otherwise deal directly with any memory issues. The following two lines are functionally equivalent in ARC:

```
NSArray *array = [[NSArray alloc] init];
```

and

```
NSArray *array = [NSArray array];
```

That's because the compiler takes control of all memory management for you. In each case, assigning the new array to the local variable retains the array. This behavior is called `strong`. By default, object variables are "strong," which means they retain their contents throughout their scope.

Important things are happening on the right-hand side of the assignment as well. In the first instance, the compiler balances the +1 retain count of the newly created object with the `array` variable's desire to retain it. It performs a simple assignment. In the second instance, the compiler retains the autoreleased object returned by the `NSArray` class's convenience method, again assigning a +1 retained object to the variable.

In ARC, you do not implement or invoke `retain`, `release`, `retainCount`, or `autorelease`. In the preceding examples, the compiler releases the array when the variable's scope ends.

As with MRR, class convenience methods—methods that create new objects for you—return autoreleased objects. In ARC, you do not need to perform the autorelease yourself. Here's what a `Car` convenience class method might look like. It's not a very interesting method, and has no side effects, but it demonstrates how ARC adds memory

From the Library of Eric Berna

management for you. With ARC, you almost never have to worry about memory leaking when you follow good programming practices.

```
+ (Car *) car
{
    // ARC implicitly adds autoreleasing
    return [[Car alloc] init];
}
```

A more in-depth discussion of ARC follows later in this chapter for experienced developers making the move from MRR to ARC coding.

Properties

Properties expose class variables and methods to outside use through what are called "accessor methods"—that is, methods that access information. Using properties might sound redundant. After all, the class definition shown in Listing 2-1 already announces public methods. So why use properties? It turns out that there are advantages to using properties over publicly declared methods, not the least of which are encapsulation, dot notation, and memory management.

Encapsulation

Encapsulation allows you to hide implementation details away from the rest of your application, including any clients that use your objects. The object's internal representation and its method mechanisms are kept separate from the way the object declares itself to the outside. Properties provide ways to expose object state and other information in a well-structured and circumscribed manner.

Properties are not, however, limited to use as public variables. They play an important role within class definitions as well. Properties allow you to add smart proactive development techniques, including lazy loading and caching orthogonally, to the rest of a class implementation. That's why classes can be property clients as well as providers.

Dot Notation

Dot notation allows you to access object information without using brackets. Instead of calling `[myCar year]` to recover the year instance variable, you use `myCar.year`. Although this may look as if you're directly accessing the year instance variable, you're not. Properties always invoke methods. These, in turn, can access an object's data. So you're not breaking an object's encapsulation because properties rely on these methods to bring data outside the object.

Due to method hiding, properties simplify the look and layout of your code. For example, you can access properties to set a table's cell text via

```
myTableViewCell.textLabel.text = @"Hello World";
```

From the Library of Eric Berna

rather than the more cumbersome

```
[[myTableViewCell textLabel] setText:@"Hello World"];
```

The property version of the code is more readable and ultimately easier to maintain. Admittedly, Objective-C 2.0's dot notation may initially confuse programmers who are used to using dots for structures instead.

Properties and Memory Management

Under manual retain/release compilation (MRR), properties can help simplify memory management. You can create properties that automatically retain the objects they're assigned to for the lifetime of your objects and release them when you set those variables to `nil`. In MRR compilation, retained properties are qualified as `retain`. (In ARC, they are `strong`.) Setting a retained property ensures that memory will not be released until you say so. Of course, properties are not a magic bullet. They must be defined and used properly.

Assigning an object to a retained property means you're guaranteed that the objects they point to will be available throughout the tenure of your ownership. As already mentioned, that guarantee extends to instance variables in ARC. By default, ARC instance variables are qualified as `strong`, holding onto any object until the instance variable is reassigned or the object finishes its lifetime.

MRR and Retained Properties

Until ARC made its debut, a retained property offered a convenient way to hold onto objects until they were no longer needed. Want to retain an object for a long time? Create a property. Now, with ARC, retained properties are needed *only* for objects that must be accessible outside your class implementation.

Listing 2-2 did not retain its `make` and `model`. Under MRR, if those objects were released somewhere else in an application, the pointers to the memory that stored those objects would become invalid. At some point, the application might try to access that memory and crash. By retaining objects, you ensure that the memory pointers remain valid and meaningful.

The `arrayWithObjects:` method normally returns an autoreleased object, whose memory is deallocated at the end of the event loop cycle. Assigning the array to a retained property means that the array sticks around indefinitely. You retain the object, preventing its memory from being released until you are done using it.

```
self.colors = [NSArray arrayWithObjects:
    @"Gray", @"Silver", @"Black", nil];
```

When you're done using the array and want to release its memory, set the property to `nil`. This approach works because Objective-C knows how to synthesize accessor methods, creating properly managed ways to change the value of an instance variable. You're not really setting a variable to `nil`. You're actually telling Objective-C to run a method that releases any previously set object and then sets the instance variable to `nil`. All this

From the Library of Eric Berna

happens behind the scenes. From a coding point of view, it simply looks as if you're assigning a variable to `nil`.

```
self.colors = nil;
```

Do not send `release` directly to retained properties—for example, `[self.colors release]`. Doing so does not affect the `colors` instance variable assignment, which now points to memory that is likely deallocated. When you next assign an object to the retained property, the memory pointed to by `self.colors` will receive an additional release message, likely causing a double-free exception.

Declaring Properties

There are two basic styles of properties: `readwrite` and `readonly`. Read-write properties, which are the default, let you modify the values you access; read-only properties do not. Use `readonly` properties for instance variables that you want to expose, without providing a way to change their values. They are also handy for properties that are generated by algorithm, such as via a cache or lazy load.

The two kinds of accessor methods you must provide are called setters and getters. Setters set information; getters retrieve information. You can define these with arbitrary method names or you can use the standard Objective-C conventions: The name of the instance variable retrieves the object, while the name prefixed with `set`, sets it. Objective-C can even synthesize these methods for you. For example, if you declare a property such as the `Car` class's `year` in your class interface, as such

```
@property (assign) int year;
```

and then synthesize it in your class implementation with

```
@synthesize year;
```

you can read and set the instance variable with no further coding. Objective-C builds two methods that get the current value (that is, `[myCar year]`) and set the current value (that is, `[myCar setYear:1962]`) and add the two dot notation shortcuts:

```
myCar.year = 1962;
NSLog(@"%d", myCar.year);
```

To build a read-only property, declare it in your interface using the `readonly` qualifier. Read-only properties use getters without setters. For example, here's a property that returns a formatted text string with car information:

```
@property (readonly) NSString *carInfo;
```

Although Objective-C can synthesize read-only properties, you can also build the getter method by hand and add it to your Class implementation. This method returns a description of the car via `stringWithFormat:`, which uses a format string a la `sprintf` to create a new string:

From the Library of Eric Berna

```
- (NSString *) carInfo
{
    return [NSString stringWithFormat:
        @"Car Info\n    Make: %@\n    Model: %@\n    Year: %d",
        self.make ? self.make : @"Unknown Make",
        self.model ? self.model : @"Unknown Model",
        self.year];
}
```

This method now becomes available for use via dot notation. Here is an example:

```
NSLog(@"%@", myCar.carInfo);
```

If you choose to synthesize a getter for a read-only property, you should use care in your code. Inside your implementation file, make sure you assign the instance variable for that property without dot notation. Imagine that you declared `model` as a read-only property. You could assign `model` with

```
model = @"Prefect";
```

but not with

```
self.model = @"Prefect";
```

The latter use attempts to call `setModel:`, which is not defined for a read-only property.

> **Note**
> The "?:" ternary operator used in this section defines a simple if-then-else conditional expression. (a ? b : c) returns b if a is true, and c otherwise.

Creating Custom Getters and Setters

Although Objective-C automatically builds methods when you `@synthesize` properties, you may skip the synthesis by creating those methods yourself. For example, you could build methods as simple as these. Notice the capitalization of the second word in the set method. Objective-C expects setters to use a method named `setInstance:`, where the first letter of the instance variable name is capitalized:

```
-(int) year
{
    return year;
}

- (void) setYear: (int) aYear
{
    year = aYear;
}
```

In MRR Objective-C, you would want to add memory management when building your own setters and getters, as demonstrated in the following code snippet. You do not need to add memory management in ARC. The compiler handles those matters for you, so the ARC `setModel:` method only has to make the `newModel` to `model` assignment:

From the Library of Eric Berna

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

- (void) setModel: (NSString *) newModel
{
    if (newModel != model) {
        [newModel retain];
        [model release];
        model = newModel;
    }
}
```

Property accessor methods go even further by building more complicated routines that generate side effects upon assignment and retrieval. For example, you might keep a count of the number of times the value has been retrieved or changed, or send in-app notifications to other objects. You might build a flushable cache to hold objects until you experience memory warnings or use lazy loading to delay creating assets until they're requested by a getter.

The Objective-C compiler remains happy so long as it finds, for any property, a getter (typically named the same as the property name) and a setter (usually `setName:`, where name is the name of the property). What's more, you can bypass any Objective-C naming conventions by specifying setter and getter names in the property declaration. This declaration creates a new Boolean property called `forSale` and declares a custom getter/setter pair. As always, you add any property declarations to the class interface.

```
@property (getter=isForSale, setter=setSalable:) BOOL forSale;
```

Now you can synthesize the methods as normal in the class implementation. The implementation is typically stored in the .m file that accompanies the .h header file:

```
@synthesize forSale;
```

If you have more than one item to synthesize, you can add them in separate `@synthesize` statements or combine them onto a single line, separating each by a comma:

```
@synthesize forSale, anotherProperty, yetAnotherProperty;
```

Using this approach creates both the normal setter and getter via dot notation plus the two custom methods, `isForSale` and `setSalable:`.

You can also use `@synthesize` statements to declare local variables that are tied to property names. For example, you can tie a property's methods to its local backing variable by using the same name. Here, the `myString` variable backs the `myString` property:

```
@interface MyClass : NSObject
{
   NSString *myString;
}
```

From the Library of Eric Berna

```
@property (strong) NSString *myString;
@end

@implementation MyClass
@synthesize myString;
@end
```

The instance variable declaration is not necessary. Synthesizing a property automatically creates its backing variable. The following snippet is functionally equivalent to the preceding one:

```
@interface MyClass : NSObject
@property (strong) NSString *myString;
@end

@implementation MyClass
@synthesize myString;
@end
```

Some Objective-C programmers like to distinguish the instance variable from the property. They do this by prefixing instance variables with an underscore. The equivalent variable can be specified in the synthesize declaration. This approach helps emphasize encapsulation, in that instance variables are private to classes:

```
@interface MyClass : NSObject
@property (strong) NSString *myString;
@end

@implementation MyClass
@synthesize myString = _myString;
@end
```

Property Qualifiers

In addition to `readwrite` and `readonly`, you can specify whether a property is retained and/or atomic. The default behavior for properties in ARC is `strong`; in MRR the default behavior is `assign`. `Strong` and `retain` are synonymous; "strong" emphasizes the object relationship while "retain" places its emphasis on the underlying mechanics.

Assigned properties are not retained. ARC uses two styles of unretained property assignment. A `weak` property uses self-nullifying pointers; you never have to worry about dangling pointers. An `unsafe_unretained` property simply points to memory and, as its name indicates, may point to an unsafe address. Under ARC, `assign` properties are used to point to non-object value types such as integers and structures.

The following sections discuss property qualifiers for MRR and ARC compilation.

From the Library of Eric Berna

MRR Qualifiers

With `assign`, there's no special retain/release behavior associated with the property, but by making it a property you expose the variable outside the class via dot notation. In MRR, a property that's declared

```
@property NSString *make;
```

uses the `assign` behavior.

Setting the property's attribute to MRR's `retain` does two things. First, it retains the passed object upon assignment. Second, it releases the previous value before a new assignment is made. You can clear up any current memory usage by assigning the retained property to `nil`. To create a retained property, add the attribute between parentheses in the declaration:

```
@property (retain) NSString *make;
```

A third attribute called `copy` copies the passed object and releases any previous value. With MRR, copies are always created with a retain count of 1.

```
@property (copy) NSString *make;
```

ARC Qualifiers

When you're declaring properties, strong properties automatically retain the objects assigned to them, releasing them only when the object whose property this is gets released or the property is set to `nil`. Use strong properties to hold onto any items that may be referenced through the lifetime of your object. By setting a property to be `strong`, you're assured that the instance the property points to will not be released back into the general memory pool until you're done with it, as shown here:

```
@property (nonatomic, strong) NSDate *date;
```

Like MRR's `assign` properties, weak properties do not retain objects or otherwise extend their lifetime. However, weak properties do something that `assign` properties never did. They ensure that if the object being pointed to gets deallocated, the property returns `nil`, not a reference to reclaimed memory. ARC eradicates dangling pointers, creating safe `nil` values instead. This is known as "zeroing" weak references.

Use `assign` properties in ARC to point to items that aren't objects, such as floats and Booleans as well as structs such as `CGRect`:

```
@property (assign) CGRect bounds;
```

Atomic Qualifiers

When you develop in a multithreaded environment, you want to use atomic properties. Xcode synthesizes atomic properties to automatically lock objects before they are accessed or modified and unlock them after. This ensures that setting or retrieving an object's value

From the Library of Eric Berna

is performed fully regardless of concurrent threads. There is no atomic keyword. All methods are synthesized atomically by default. You can, however, state the opposite, allowing Objective-C to create accessors that are nonatomic:

```
@property (nonatomic, strong) NSString *make;
```

Marking your properties nonatomic does speed up access, but you might run into problems should two competing threads attempt to modify the same property at once. Atomic properties, with their lock/unlock behavior, ensure that an object update completes from start to finish before that property is released to a subsequent read or change.

Some will argue that accessors are not usually the best place for locks and cannot ensure thread safety. An object might be set to an invalid state, even with all atomic properties. As Jay Spenser, one of my early readers, pointed out, "If you had a trade-in thread and an inventory thread, you could end up thinking you had a 1946 Tesla Prefect on your lot."

Key-Value Coding

Key-value coding allows you to access properties by using strings that store property names. Just as you can access a property by `myObject.propertyName`, you can also retrieve the same value by issuing `[myObject valueForKey:@"propertyName"]`. Key-value coding is particularly valuable when working with numerous objects that share property names, allowing you to extract properties by name in a collection.

For example, this code snippet takes an average of all the `myNumber` properties in an array. It uses the `@avg` collection operator combined with the name of the key-value-compliant `myNumber` property:

```
NSNumber *averageNumber = [myArray valueForKeyPath:@"@avg.myNumber"];
NSLog(@"Average number is : %0.2f", averageNumber.floatValue);
```

You might use key-value coding to extract all items from an array whose property exceeds 50, as shown here:

```
// collect the indices of each object whose myNumber is over 50
NSIndexSet *indexSet = [myArray indexesOfObjectsPassingTest:
    ^BOOL(id obj, NSUInteger idx, BOOL *stop) {
        NSNumber *num = [(MyClass *)obj valueForKey:@"myNumber"];
        return (num.intValue > 50);
}];
NSArray *subArray = [myArray objectsAtIndexes:indexSet];
NSLog(@"Numbers larger than 50 include %@", subArray);
```

Key-value coding offers a highly flexible way to get at information structured within an object. As these simple examples demonstrate, the key-value coding API provides powerful routines that support that access, especially when applying key-value across numerous object instances.

From the Library of Eric Berna

Key-Value Observing

Key-value observing (better known as KVO) introduces a way to trigger notifications when object properties change. Just like you can add observers for notifications, targets for controls, and delegates for other objects, KVO lets you create callbacks when an object's state updates.

Add observers to objects by specifying which key path they are interested in. Here, the main view controller (`self`) will receive a message whenever `myObject`'s `myString` property changes. You can add many observers to an object, or add the same observer for many key paths.

```
[myObject addObserver:self forKeyPath:@"myString"
    options:NSKeyValueObservingOptionNew context:NULL];
```

The callback method for key-value observing is fixed. Implement `observeValueForKeyPath:ofObject:change:context:` to participate in observing. Once the observer is added, this method is called each time the `myString` property changes to a new value. If you observe many items, you may want to add some kind of tag to your objects to distinguish which object triggered the observer callback:

```
- (void) observeValueForKeyPath:(NSString *)keyPath
    ofObject:(id)object change:(NSDictionary *)change
    context:(void *)context
{
    NSLog(@"Key path %@ has changed to %@",
        keyPath, [object valueForKey:keyPath]);
}
```

MRR and High Retain Counts

In MRR, retain counts that go and stay above +1 do not necessarily mean you've done anything wrong. Consider the following code segment. It creates a view and starts adding it to arrays. This raises the retain count from +1 up to +4.

```
// On creation, view has a retain count of +1
UIView *view = [[[UIView alloc] init] autorelease];
printf("Count: %d\n", [view retainCount]); // MRR only

// Adding it to an array increases that retain count to +2
NSArray *array1 = [NSArray arrayWithObject:view];
printf("Count: %d\n", [view retainCount]);

// Another array, retain count goes to +3
NSArray *array2 = [NSArray arrayWithObject:view];
printf("Count: %d\n", [view retainCount]);

// And another +4
NSArray *array3 = [NSArray arrayWithObject:view];
printf("Count: %d\n", [view retainCount]);
```

From the Library of Eric Berna

Notice that each array was created using a class convenience method and returns an autoreleased object. The view is set as autorelease, too. Some collection classes such as `NSArray` automatically retain objects when you add them into an array and release them when either the objects are removed (mutable objects only) or when the collection is released. This code has no leaks because every one of the four objects is set to properly release itself and its children when the autorelease pool drains.

When `release` is sent to the three arrays, each one releases the view, bringing the count down from +4 to +1. The final `release`, when the object is at +1, allows the view to deallocate when this method finishes: no leaks, no further retains, no problems.

Other Ways to Create Objects

You've seen how to use `alloc` to allocate memory. Objective-C offers other ways to build new objects. You can discover these by browsing class documentation, as the methods vary by class and framework. As a rule of thumb, if you build an MRR object using any method whose name includes `alloc`, `new`, `create`, or `copy`, you maintain responsibility for releasing the object. Unlike class convenience methods, methods that include these words generally do not return autoreleased objects.

In ARC, the same rules hold. These methods return +1 new objects; however, ARC's memory management means you don't have to worry about this in your code. The retain counts are managed for you and you do not explicitly have to release the object.

Sending a copy message to an object in MRR, for example, duplicates it. `copy` returns an object with a retain count of +1 and no assignment to the autorelease pool. Use `copy` when you want to duplicate and make changes to an object while preserving the original. Note that for the most part, Objective-C produces shallow copies of collections such as arrays and dictionaries. It copies the structure of the collection, and maintains the addresses for each pointer, but does not perform a deep copy of the items stored within.

C-Style Object Allocations

As a superset of C, Objective-C programs for the iOS SDK often use APIs with C-style object creation and management. Core Foundation (CF) is a Cocoa Touch framework with C-based function calls. When working with CF objects in Objective-C, you build objects with `CFAllocator`s and often use the `CFRelease()` function to release object memory.

There are, however, no simple rules. You may end up using `free()`, `CFRelease()`, and custom methods such as `CGContextRelease()` all in the same scope, side by side with standard Objective-C class convenience methods such as `imageWithCGImage:`. The function used to create the context object used in the following snippet is `CGBitmapContextCreate()`, and like most Core Foundation function calls it does not return an autoreleased object. This code snippet is admittedly a bit of a forced example to show off lots of different approaches in action; it builds a `UIImage`, the iOS class that stores image data:

From the Library of Eric Berna

```
UIImage *buildImage(int imgsize)
{
    // Create context with allocated bits
    CGContextRef context  =
        MyCreateBitmapContext(imgsize, imgsize);
    CGImageRef myRef =
        CGBitmapContextCreateImage(context);
    free(CGBitmapContextGetData(context)); // Standard C free()
    CGContextRelease(context); // Core Graphics Release
    UIImage *img = [UIImage imageWithCGImage:myRef];
    CFRelease(myRef); // Core Foundation Release
    return img;
}
```

ARC does *not* perform memory management for Core Foundation. A discussion about using ARC with Core Foundation follows later in this chapter.

Carbon and Core Foundation

Working with Core Foundation comes up often enough that you should be aware of its existence and be prepared to encounter its constructs, specifically in regard to its frameworks. Frameworks are libraries of classes you can utilize in your application.

Table 2-2 explains the key terms involved. To summarize the issue, early OS X used a C-based framework called Core Foundation to provide a transitional system for developing applications that could run on both Classic Mac systems as well as Mac OS X. Although Core Foundation uses object-oriented extensions to C, its functions and constructs are all based on C, not Objective-C.

Table 2-2 **Key OS X Development Terms**

Term	Definition
Foundation	The core classes for Objective-C programming, offering all the fundamental data types and services needed for Cocoa and Cocoa Touch. A section at the end of this chapter introduces some of the most important Foundation classes you'll use in your applications.
Core Foundation	A library of C-based classes that are based on Foundation APIs but that are implemented in C. Core Foundation uses object-oriented data but is not built using the Objective-C classes.
Carbon	An early set of libraries provided by Apple that use a procedural API. Carbon offered event handling support, a graphics library, and many more frameworks. Some Carbon APIs live on through Core Foundation. Carbon was introduced for the Classic Mac OS, first appearing in Mac OS 8.1.
Cocoa	Apple's collection of frameworks, APIs, and runtimes that make up the modern Mac OS X runtime system. Frameworks are primarily written in Objective-C, although some continue to use C/C++.

From the Library of Eric Berna

Table 2-2 **Key OS X Development Terms**

Term	Definition
Cocoa Touch	Cocoa's equivalent for the iOS SDK, where the frameworks are tuned for the touch-based mobile iOS user experience. Some iOS frameworks such as Core Audio and Open GL are considered to reside outside Cocoa Touch.
Toll Free Bridging	A method of Cocoa/Carbon integration. Toll Free Bridging refers to sets of interchangeable data types. For example, Cocoa's Foundation (`NSString *`) object can be used almost interchangeably with Carbon's Core Foundation's `CFStringRef`. Bridging connects the C-based Core Foundation with the Objective-C Foundation world. See the ARC discussion later in this chapter to learn how to use Toll Free Bridging with the ARC compiler.

Core Foundation technology lives on through Cocoa. You can and will encounter C-style Core Foundation when programming iOS applications using Objective-C. The specifics of Core Foundation programming fall outside the scope of this chapter, however, and are best explored separately from learning how to program in Objective-C.

Deallocating Objects

There's no garbage collection in iOS and little likelihood there ever will be. With ARC, there's also little reason for it. Objects automatically clean up after themselves. So what does that mean in practical terms? When using MRR, you are responsible for bringing objects to their natural end. With ARC, you remain responsible for tidying up any loose ends that may linger at the end of the object's life.

MRR Deallocation

Under manual memory management, instance variables must release any retained objects before deallocation. You as the developer must ensure that those objects return to a retain count of 0 before the parent object is itself released. You perform these releases in `dealloc`, a method automatically called by the runtime system when an object is about to be released. If you use an MRR class with object instance variables (that is, not just floats, ints, and Bools), you probably need to implement a deallocation method. The basic dealloc method structure looks like this:

```
- (void) dealloc
{
    // Class-based clean-up
    clean up my own properties and instance variables here

    // Clean up superclass
    [super dealloc]
}
```

From the Library of Eric Berna

The method you write should work in two stages. First, clean up any retained memory from your class. Then ask your superclass to perform its cleanup routine. The special `super` keyword refers to the superclass of the object that is running the `dealloc` method. How you clean up depends on whether your instance variables are automatically retained.

You've read about creating objects, building references to those objects, and ensuring that the objects' retain counts stay at +1 after creation. Now, you see the final step of the object's lifetime—namely releasing those +1 objects so they can be deallocated.

In the case of retained properties, set them to `nil` using dot notation assignment. This calls the custom setter method synthesized by Objective-C and releases any prior object the property has been set to. Assuming that prior object had a retain count of +1, this release allows that object to deallocate:

```
self.make = nil;
```

When using plain (nonproperty) instance variables or `assign`-style properties, send `release` at deallocation time. Say, for example, you've defined an instance variable called `salesman`. It might be set at any time during the lifetime of your object. The assignment of `salesman` might look like this:

```
// release any previous value
[salesman release];

// make the new assignment. Retain count is +1
salesman = [[SomeClass alloc] init];
```

This assignment style means that `salesman` could point to an object with a +1 retain count at any time during the object's lifetime. Therefore, in your `dealloc` method, you must release any object currently assigned to `salesman`. You can guard this with a check if `salesman` is not `nil`, but practically you're free to send `release` to `nil` without consequence, so feel free to skip the check.

```
[salesman release];
```

A Sample MRR Deallocation Method

Keeping with an expanded `Car` class that uses retained properties for `make`, `model`, and `colors`, and that has a simple instance variable for `salesman`, the final deallocation method would look like this. The integer `year` and the Boolean `forSale` instance variables are not objects and do not need to be managed this way.

```
- (void) dealloc
{
    self.make = nil;
    self.model = nil;
    self.colors = nil;
    [salesman release];
    [super dealloc];
}
```

From the Library of Eric Berna

Managing an object's retain count proves key to making Objective-C memory management work. Few objects should continue to have a retain count greater than +1 after their creation and assignment. By guaranteeing a limit, your final releases in `dealloc` are ensured to produce the memory deallocation you desire.

ARC Deallocation

You can create your own `dealloc` methods in ARC classes, just like you do in MRR ones. There are, however, differences. You never call `[super dealloc]` the way you do with MRR. And, you do not worry about any memory management. If you look at the sample MRR deallocation method, that pretty much eliminates every single line in that code.

So what's an ARC `dealloc` method supposed to do? It's where you clean up all associated issues such as freeing `malloc`'ed memory, disposing of system sounds, or calling Core Foundation `release` as needed. The compiler automatically calls `[super dealloc]` for you, regardless of whether or not you override `dealloc` in your class.

Cleaning Up Other Matters

The `dealloc` method offers a perfect place to clean up shop. For example, you might need to dispose of an Audio Toolbox sound (as shown here) or perform other maintenance tasks before the class is released. These tasks almost always relate to Core Foundation, Core Graphics, Core Audio, Address Book, Core Text, or similar C-style frameworks.

```
if (snd) AudioServicesDisposeSystemSoundID(snd);
```

Think of `dealloc` as your last chance to tidy up loose ends before your object goes away forever. Whether this involves shutting down open sockets, closing file pointers, or releasing resources, use this method to make sure your code returns state as close to pristine as possible.

Using Blocks

In Objective-C 2.0, **blocks** refer to a language construct that supports "closures," a way of treating code behavior as objects. First introduced to iOS in the 4.*x* SDK, Apple's language extension makes it possible to create "anonymous" functionality, a small coding element that works like a method without having to be defined or named as a method.

This allows you to pass that code as parameters, providing an alternative to traditional callbacks. Instead of creating a separate "doStuffAfterTaskHasCompleted" method and using the method selector as a parameter, blocks allow you to pass that same behavior directly into your calls. This has two important benefits.

First, blocks localize code to the place where that code is used. This increases maintainability and readability, moving code to the point of invocation. This also helps minimize or eliminate the creation of single-purpose methods.

Second, blocks allow you to share lexical scope. Instead of explicitly passing local variable context in the form of callback parameters, blocks can implicitly read locally declared variables and parameters from the calling method or function. This context-sharing

From the Library of Eric Berna

provides a simple and elegant way to specify the ways you need to clean up or otherwise finish a task without having to re-create that context elsewhere.

Defining Blocks in Your Code

Closures have been used for decades. They were introduced in Scheme (although they were discussed in computer science books, papers, and classes since the 1960s), and popularized in Lisp, Ruby, and Python. The Apple Objective-C version is defined using a caret symbol, followed by a parameter list, followed by a standard block of code, delimited with braces. Here is a simple use of a block. It is used to show the length of each string in an array of strings.

```
NSArray *words = [@"This is an array of various words"
    componentsSeparatedByString:@" "];
[words enumerateObjectsUsingBlock:
    ^(id obj, NSUInteger idx, BOOL *stop){
        NSString *word = (NSString *) obj;
        NSLog(@"The length of '%@' is %d", word, word.length);
}];
```

This code enumerates the "words" array, applying the block code to each object in that array. The block uses standard Objective-C calls to log each word and its length. Enumerating objects is a common way to use blocks in your applications. This example does not highlight the use of the `idx` and `stop` parameters. The `idx` parameter is an unsigned integer, indicating the current index of the array. The `stop` pointer references a Boolean value. When the block sets this to `YES`, the enumeration stops, allowing you to short-circuit your progress through the array.

Block parameters are defined within parentheses after the initial caret. They are typed and named just as you would in a function. Using parameters allows you to map block variables to values from a calling context, again as you would with functions.

Using blocks for simple enumeration works similarly to existing Objective-C 2.0 "for in" loops. What blocks give you further in the iOS APIs are two things. First is the ability to perform concurrent and/or reversed enumeration using the `enumerateObjectsAtIndexes:options:usingBlock:` method. This method extends array and set iteration into new and powerful areas. Second is dictionary support. Two methods (`enumerateKeysAndObjectsUsingBlock:` and `enumerateKeysAndObjectsWithOptions:usingBlock:`) provide dictionary-savvy block operations with direct access to the current dictionary key and object, making dictionary iteration cleaner to read and more efficient to process.

Assigning Block References

Because blocks are objects, you can use local variables to point to them. For example, you might declare a block as shown in this example. This code creates a simple `maximum` function, which is immediately used and then discarded:

From the Library of Eric Berna

```
// Creating a maximum value block
float (^maximum)(float, float) = ^(float num1, float num2) {
    return (num1 > num2) ? num1 : num2;
};

// Applying the maximum value block
NSLog(@"Max number: %0.1f", maximum(17.0f, 23.0f));
```

Declaring the block reference for the `maximum` function requires that you define the kinds of parameters used in the block. These are specified within parentheses but without names (that is, a pair of floats). Actual names are only used within the block itself.

The compiler automatically infers block return types, allowing you to skip specification. For example, the return type of the block in the preceding example is `float`. To explicitly type blocks, add the type after the caret and before any parameter list, like this.

```
// typing a block
float (^maximum)(float, float) = ^float(float num1, float num2) {
    return (num1 > num2) ? num1 : num2;
};
```

Because the compiler generally takes care of return types, you need only worry about typing in those cases where the inferred type does not match the way you'll need to use the returned value.

Blocks provide a good way to perform expensive initialization tasks in the background. The following example loads an image from an Internet-based URL using an asynchronous queue. When the image has finished loading (normally a slow and blocking function), a new block is added to the main thread's operation queue to update an image view with the downloaded picture. It's important to perform all GUI updates on the main thread, and this example makes sure to do that.

```
// Create an asynchronous background queue
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
[queue addOperationWithBlock:
^{
    // Load the weather data
    NSURL *weatherURL = [NSURL URLWithString:
        @"http://image.weather.com/images\
        /maps/current/curwx_600x405.jpg"];
    NSData *imageData = [NSData dataWithContentsOfURL:weatherURL];

    // Update the image on the main thread using the main queue
    [[NSOperationQueue mainQueue] addOperationWithBlock:^{
        UIImage *weatherImage = [UIImage imageWithData:imageData];
        imageView.image = weatherImage;}];}];
```

This code demonstrates how blocks can be nested as well as used at different stages of a task. Take note that any pending blocks submitted to a queue hold a reference to that queue, so the queue is not deallocated until all pending blocks have completed. That

From the Library of Eric Berna

allows me to create an autoreleased queue in this example without worries that the queue will disappear during the operation of the block.

Blocks and Local Variables

Blocks are offered read access to local parameters and variables declared in the calling method. To allow the block to change data, the variables must be assigned to storage that can survive the destruction of the calling context. A special kind of variable, the `__block` variable, does this by allowing itself to be copied to the application heap when needed. This ensures that the variable remains available and modifiable outside the lifetime of the calling method or function. It also means that the variable's address can possibly change over time, so `__block` variables must be used with care in that regard.

> **Note**
>
> The `_block` qualifier has changed semantics in ARC, now acting as a `__strong` reference, so it retains and releases accordingly. Under MRR, it uses `__unsafe_unretained` (`assign`) behavior, where it did not implicitly retain.

The following example shows how to use locally scoped mutable variables in your blocks. It enumerates through a list of numbers, selecting the maximum and minimum values for those numbers:

```
// Using mutable local variables
NSArray *array = [@"5 7 3 9 11 13 1 2 15 8 6"
    componentsSeparatedByString:@" "];

// assign min and min to block storage using __block
__block uint min = UINT_MAX;
__block uint max = 0;

[array enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL *stop)
{
    // update the max and min values while enumerating
    min = MIN([obj intValue], min);
    max = MAX([obj intValue], max);

    // display the current max and min
    NSLog(@"max: %d min: %d\n", max, min);
 }];
```

Blocks and `typedef`

Using `typedef` simplifies the way you declare block properties in your classes and use blocks as arguments to method calls. The following snippet defines a `DrawingBlock` type and uses it to create a `copy` property in the `MyView` class:

From the Library of Eric Berna

```
typedef void (^DrawingBlock)(UIColor *);

@interface MyView : UIView
@property (copy) DrawingBlock drawingBlock;
@end
```

This particular block type takes one argument, a `UIColor` object. You can assign a block to a variable of this type like this:

```
theView.drawingBlock = ^(UIColor *foregroundColor) {
    [theView.superview.backgroundColor set];
    CGContextFillRect(UIGraphicsGetCurrentContext(),
        theView.bounds);
    [foregroundColor set];
    [[UIBezierPath bezierPathWithRoundedRect:theView.bounds
        cornerRadius:32.0f] fill];
};
```

It might be called from within the class implementation like this:

```
drawingBlock([UIColor greenColor]);
```

This is, admittedly, a contrived example. It might suffer if `theView` changed after the block is created and assigned. What this example demonstrates is how to move behavior out of a class definition, allowing a client to customize, in this case, a drawing block, whose creation is decoupled from the original class's implementation.

Blocks and Memory Management with MRR

When using blocks with standard Apple APIs, you will not need to retain or copy blocks. When creating your own blocks-based APIs in MRR, where your block will outlive the current scope, you may need to do so. In such a case, you will generally want to copy (rather than retain) the block to ensure that the block is allocated on the application heap. You may then autorelease the copy to ensure proper memory management is followed.

Using blocks with ARC is discussed later in this chapter.

Other Uses for Blocks

In this section, you have seen how to define and apply blocks. In addition to the examples shown here, blocks play other roles in iOS development. Blocks can be used with notifications, animations, and as completion handlers. Many iOS APIs use blocks to simplify calls and coding. You can easily find block-based APIs by searching the Xcode Document set for method selectors containing "block," "completion," and "handler."

Getting Up to Speed with ARC

Automated reference counting (ARC) offers a new compiler feature for your iOS projects. Introduced by way of LLVM (llvm.org), ARC simplifies memory management by

From the Library of Eric Berna

automating the way objects are retained and released. By handling your memory for you, ARC allows you to focus your effort on code semantics, reducing the time and overhead you devote to thinking about memory management issues.

The problems with this in practice are as follows. First, most experienced iOS developers already use memory management almost reflexively. They don't have to plan when to retain an object, and when to autorelease it—it's something that comes out of years of patterned responses. Second, ARC has yet to fully stabilize. Apple is still working on adding features as this chapter is being written, even though initial technical specifications have been published (http://clang.llvm.org/docs/AutomaticReferenceCounting.html).

This quick overview, therefore, introduces ARC with a single goal: to get you up to speed as fast as possible, moving past your previous coding patterns into new ones. It will help you learn the steps to translate your code from the old form to the new so you can get going right away. It also shows how to preserve old-style compilation for those who need to avoid that transition as long as possible.

Property and Variable Qualifiers

ARC means approaching memory management in new ways. The first and easiest of these rules is this: With ARC, you move from `assign` and `retain` attributes for objects to `weak` and `strong`. Each of these (`assign`, `retain`, `weak`, and `strong`) is a *qualifier*. They qualify the kind of property attributes used. These same qualifiers can be used to set the attributes of variables, in the same way you can use `const` and `volatile` as type qualifiers in standard C.

> **Note**
>
> ARC continues to use `assign` qualifiers for non-object values such as `int` and `CGRect`.

Strong and Weak Properties

When you're declaring properties, `strong` usurps the role of retained properties. In versions previous to Objective-C 2.0, retained properties automatically retained items, releasing them only when your object was released or the property set to `nil`.

Use strong properties to hold onto any items that may be referenced through the lifetime of your object. By setting a property to be `strong`, you're assured that the instance the property points to will not be released back into the general memory pool until you're done with it.

Thus, instead of

```
@property (nonatomic, retain) NSDate *date;
```

you declare

```
@property (nonatomic, strong) NSDate *date;
```

Like `strong` properties, `weak` properties replace a previous Objective-C style—in this case `assign`. Like `assign` properties, `weak` properties do not retain objects or otherwise extend their lifetime. However, `weak` properties do something that `assign` properties never did. They ensure that if the object being pointed to gets deallocated, the property

From the Library of Eric Berna

returns nil, not a reference to reclaimed memory. ARC eradicates dangling pointers, creating safe nil values instead. This is known as "zeroing weak references."

You are limited to deployment targets of iOS 5.0 or later (also OS X 10.7 or later) when using weak qualifiers. Prior to iOS 5.0, the operating system runtime cannot guarantee that dangling pointers get reassigned to nil.

If the compiler reports errors, as shown in Figure 2-3, check your deployment target. Use the Project Navigator and select the project file. In the Build Settings, ensure that the iOS Deployment Target is set to at least iOS 5.0. If you must deploy to iOS 4.*x*, use __unsafe_unretained qualifiers instead of weak. This is functionally equivalent to old-style assign properties and will work on old-style OS runtimes. When working with _unsafe_unretained qualifiers, make sure you don't dereference those pointers after the object disappears, just as you had to make sure pre-ARC.

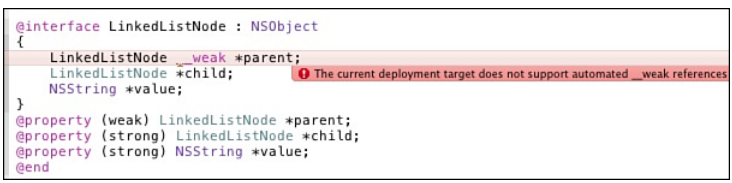


Figure 2-3 Weak references are supported starting with iOS 5.0. ARC will not compile to earlier deployment targets.

> **Note**
>
> Use copy properties to create a copy of an object and then retain it. Copy properties are like strong properties that first copy the object in question and then use that copy instead of the original object. The ARC migratory converts assign properties to unsafe_unretained for deployment targets prior to 5.0. When you are using LLVM 3 with non-ARC code, strong and unsafe_unretained are synonyms for retain and assign.

Using Variable Qualifiers

ARC's weak and strong qualifiers are not limited to property attributes. You can also qualify variable declarations in your code. All variables, such as

```
NSDate *lastRevision;
```

default to strong handling. They retain any object assigned to them throughout their scope. Consider the code in Listing 2-3. In old-style Objective-C, this code might have been problematic. When the retained property is set to the new date, the old property was sent a release message.

Listing 2-3 **Contrasting Old and New Style Coding**

```
NSDate *lastRevision = self.date; // store the most recently used date
self.date = [NSDate date]; // update the date property to the current time
NSTimeInterval timeElapsed = [date timeIntervalSinceDate:lastRevision];
NSLog(@"Time between revisions: %f", timeElapsed);
```

From the Library of Eric Berna

With ARC, the compiler automatically ensures that `lastRevision` is not deallocated before it's used to calculate the elapsed time interval. Variable qualification persists throughout their scoping. Listing 2-3 works because the `lastRevision` date is qualified as strong by default, thus retaining its value.

You can explicitly declare the qualifier in code as well. The following snippet is equivalent to the qualifier-free declaration:

```
NSDate __strong *lastRevision;
```

Available memory management qualifiers include the following. Notice the double underscores that begin each qualifier:

- **`__strong`**—The default, a strong qualifier holds onto an object throughout a variable's scope.
- **`__weak`**—Weak qualifiers use autozeroing weak references. If the assigned object gets deallocated, the variable is reset to `nil`. Weak references are available only for deployed targets of iOS 5.0 and later.
- **`__unsafe_unretained`**—An unsafe reference works like an old assignment. You may encounter invalid pointers should the assigned object be deallocated.
- **`__autoreleasing`**—These variables are used with object pointer arguments (`id *`), which are autoreleased on return. The `NSError` class offers the most common use case for these variables—for example, `int result = doSomething(arg, arg, arg, &error);`. Convenience methods also return autoreleased objects.

Apple recommends avoiding `weak` qualifiers when you intend to use objects that have no other references to them. Here's an example:

```
NSDate __weak *weakDate = [NSDate date];
NSLog(@"The date is %@", weakDate);
```

Theoretically, the date that prints out *could* be `nil`; that's because no other references exist to the date other than the weak one. In practice, however, the date *does* display correctly. You just can't depend on it. Avoid this pattern where possible.

Autoreleased Qualifiers

ARC automatically rewrites strong declarations to autoreleasing ones for pointers to objects. You may want to declare autoreleased qualifiers in your code directly to better match reality. For example, consider this file manager call. If it fails, it assigns an error to the `NSError` object that was passed to it as an `(id *)` reference:

```
NSError __autoreleasing *error

// Retrieve the icon's file attributes
NSString *iconLocation =
    [[NSBundle mainBundle] pathForResource:@"icon" ofType:@"png"];
```

From the Library of Eric Berna

```
NSDictionary *dictionary = [[NSFileManager defaultManager]
    attributesOfItemAtPath:iconLocation error:&error];

if (!dictionary)
    NSLog(@"Could not get attribute information: %@", error);
```

> **Note**
>
> This code does not set `error` to `nil` when declaring the variable. With ARC, all strong, weak, and autoreleasing stack variables are automatically initialized with `nil`.

If you do not declare `error` explicitly as autoreleasing, the compiler performs the following tweaks for you behind the scenes. It creates an autoreleasing temporary variable (probably not named "tmp," though) and uses that in the method call. It then performs an assignment back to your original strong variable. During optimization, the compiler removes the extra variable. As with previous versions of Objective-C, autoreleased objects persist until the next item in the event loop is processed.

```
NSError __strong *error;
NSError __autoreleasing *tmp = error;

NSString *iconLocation =
    [[NSBundle mainBundle] pathForResource:@"icon" ofType:@"png"];
NSDictionary *dictionary = [[NSFileManager defaultManager]
    attributesOfItemAtPath:iconLocation error:&error];
error = tmp;

if (!dictionary)
    NSLog(@"Could not get attribute information: %@", error);
```

Reference Cycles

ARC does not eliminate reference cycles. For example, consider the following linked list class declaration:

```
@interface LinkedListNode : NSObject
{
    LinkedListNode *parent;
    LinkedListNode *child;
    NSString *value;
}
@property (strong) LinkedListNode *parent;
@property (strong) LinkedListNode *child;
@property (strong) NSString *value;
@end
```

When an object uses strong links to its parent *and* its child, it creates two way linking. This causes a reference cycle. The linked list cannot be deallocated unless its members are

From the Library of Eric Berna

manually torn down after use—that is, the links are removed for each object along the chain. Otherwise, the child's references hold onto the parent, and the parent's references hold onto the child.

Avoid this situation by applying strong references from parent to child and weak references from child to parent. The weak link is applied up the chain (from child to parent) rather than down the chain (parent to child) to ensure that removing a child does not collapse your data structure and to allow the entire list to be deallocated when its top-most parent is no longer referenced.

Here's how the class interface declaration looks when you rewrite it using weak child-to-parent linking. As mentioned, you must use a deployment target of at least iOS 5.0 for the following to compile:

```
@interface LinkedListNode : NSObject
{
    LinkedListNode __weak *parent;
    LinkedListNode *child;
    NSString *value;
}
@property (weak) LinkedListNode *parent;
@property (strong) LinkedListNode *child;
@property (strong) NSString *value;
@end
```

A similar scenario arises when using completion handlers in ARC, as shown in Figure 2-4. If the completion block refers to its owner, you run into a reference cycle. The owner is retained by the handler, just as the owner retains the handler.

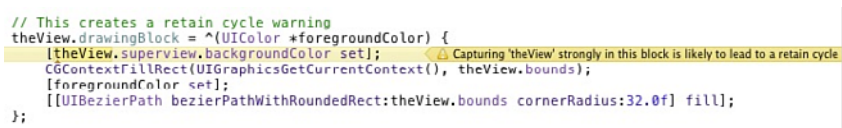


Figure 2-4 ARC detects possible strong retain cycles. Here, the completion block is defined with `copy`, a strong style.

The easiest way to work around this is to use a temporary weak variable to point to any object that would otherwise be strongly retained Here's an example:

```
MyView __weak *weakView = theView;
theView.drawingBlock = ^(UIColor *foregroundColor) {
    [weakView.superview.backgroundColor set];
    CGContextFillRect(UIGraphicsGetCurrentContext(), weakView.bounds);
    [foregroundColor set];
    [[UIBezierPath bezierPathWithRoundedRect:weakView.bounds
        cornerRadius:32.0f] fill];
};
```

From the Library of Eric Berna

To be more thorough, Apple recommends using strong references within blocks to retain weak references for the duration of the block. Notice the post-assignment check that ensures that the object still exists:

```
MyView __weak *weakView = theView;
theView.drawingBlock = ^(UIColor *foregroundColor) {
    MyView *strongView = weakView;
    if (strongView)
    {
        // view still exists
        [strongView.superview.backgroundColor set];
        CGContextFillRect(UIGraphicsGetCurrentContext(),
            strongView.bounds);
        [foregroundColor set];
        [[UIBezierPath bezierPathWithRoundedRect:strongView.bounds
            cornerRadius:32.0f] fill];
    }
    else
    {
        // view was prematurely released
    }
};
```

> **Note**
>
> The `__block` qualifier has changed semantics in ARC, now acting as a `__strong` reference, so it retains and releases accordingly. Formerly it used `__unsafe_unretained` (`assign`) behavior, where it did not implicitly retain.

Autorelease Pools

Create autorelease pools in ARC by using the `@autoreleasepool` construct shown in Listing 2-4. This new construct provides performance that's many times more efficient than the original `NSAutoreleasePool`. Apple says it can be up to five or six times faster.

Listing 2-4 **main.m, ARC Style**

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

From the Library of Eric Berna

As I discovered to my dismay (when executing Quartz image creation), the ARC migratory cannot yet handle variables created inside autorelease pools, retained, and then used outside them. Therefore, beware. Declare a `strong` variable outside the pool (local or instance) and use it to hold onto any object needed later. Here is an example from Chapter 15's XML Parser example that addresses that need to transfer objects created inside a pool for later use:

```
- (TreeNode *)parseXMLFromURL: (NSURL *) url
{
    TreeNode *results = nil;
    @autoreleasepool {
        NSXMLParser *parser =
            [[NSXMLParser alloc] initWithContentsOfURL:url];
        results = [self parse:parser];
    }
    return results;
}
```

Opting into and out of ARC

In the newest Xcode, all new projects are created ARC compliant. Old-style projects imported into the new Xcode do not transition to ARC until you migrate your code. Migration is offered in Edit > Convert > Convert to Objective C Automatic Reference Counting. Migration helps you move from memory managed code (manual retain/release, or MRR) to ARC.

Thoroughly back up your projects first. No matter how much Xcode offers to help you with that, it's best to protect yourself. Create your own revision point that you can move back to outside of Xcode auspices.

Migrating to ARC

Xcode migration assists you through the process of changing both settings and code. Your project updates to use the LLVM 3.0 compiler and is prescanned for lexical issues. Lexical issues will include any in-code autorelease pools as well as any other potential conflicts. The migration tool tags these, allowing you to manually alter items as needed before it begins the in-code conversion.

Figure 2-5 shows a typical conversion analysis result. The project that was scanned included `NSAutoreleasePool` objects as well as Core Foundation objects cast to `id`. The Issue Navigator allows you to address each complaint, one by one.

You can jumpstart your conversion by pre-changing the contents of your `main()` function in the main.m file to the code shown in Listing 2-4. Use your actual app delegate class name, not the placeholder shown here. This book is being written fairly early in the ARC introduction process. As Xcode's migration tool continues to improve, this step may no longer be necessary.

From the Library of Eric Berna

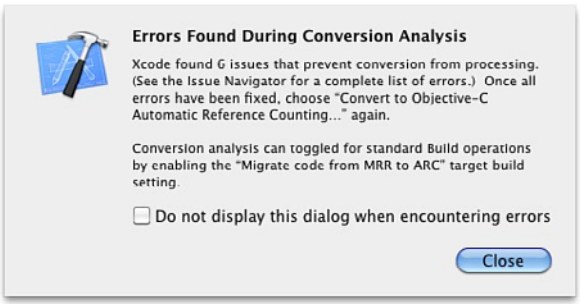


Figure 2-5 Xcode scans your project for lexical issues before proceeding with the conversion.

Once your project passes the lexical scan, it moves on to the actual ARC conversion. Xcode introduces the process and asks permission to begin. Once you click Next, Xcode allows you to enable or disable automatic snapshots. Make your choice and continue on. From there on, let Xcode do its work.

In early beta releases, this could take a bit of time and involve crash-and-burn. Here's where you need to let Apple do its magic and hope that everything gets finished and released expeditiously. For whatever reason, everything seems to convert better when using a device scheme rather than a simulator scheme, so if you run into i386 errors, look at the current active scheme.

Will My Code Still Work after Conversion?

I have had significant issues of production code not working after ARC conversion. Keep in mind that ARC is a beta release at the time this book is being written *and* that code that relies heavily on Core Foundation (as mine does) may need more love and attention than the automated conversion can offer. Your mileage can and will vary. Code written from scratch under ARC will probably be more reliable and maintainable than any that has been autoconverted.

Be aware that you can disable conversion on a file-by-file basis; read on to discover how.

Disabling ARC across a Target

You are not required to use ARC. That's handy if you'd rather continue development for iOS 5 using your current code base without making it ARC compatible yet. The switch for ARC is found in the Target > Build Settings > Apple LLVM Compiler 3.0 - Code Generation section, as shown in Figure 2-6. This translates to enabling or omitting the `-fobjc-arc` compiler flag.

Remaining in a pre-ARC compiler world limits you from taking advantage of the new ARC features in exchange for backward code stability.

From the Library of Eric Berna

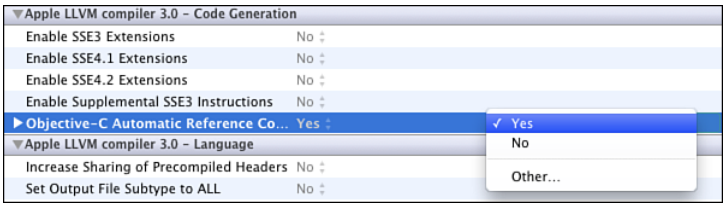


Figure 2-6 Target Build settings allow you to choose whether or not to use ARC.

Disabling ARC on a File-by-File Basis

To disable ARC for any given file, open Target > Build Phases > Compile Sources. Select a file and edit the Compiler Flags field, adding `-fno-objc-arc`. (The converse, `-fobjc-arc`, explicitly enables ARC.) This prevents ARC from being applied, as evaluated on a file-by-file basis. It also will affect the migrator in future Xcode builds, preventing the code from being updated.

You'll want to do this for any code that is excessively affected by ARC migration, particularly classes that rely heavily on Core Foundation calls. When non-ARC code interacts with ARC code, it need only obey normal Objective-C conventions. For example, ARC methods that use the phrases "copy" and "new" return objects with +1 retain counts. Convenience methods return autoreleased objects, and so forth.

Control ARC's return behavior by declaring `NS_RETURNS_RETAINED` and `NS_RETURNS_NOT_RETAINED`. For example,

```
- (NSString *) copyName NS_RETURNS_NOT_RETAINED;
```

returns an autoreleased object to non-ARC code, even though the name starts with the phrase "copy."

The `NS_RETURN` declarations work in all versions of LLVM, in non-ARC code as well as ARC code. These declarations help the static analyzer override implicit misunderstandings, such as when `+ (id) newObject` should return an autoreleased object as a convenience method rather than a +1 version.

Use special care when working with weak references. You can explicitly disallow weak references to your class instances in your implementation:

```
- (BOOL) supportsWeakPointers { return NO; }
```

This approach is built in to a number of `AppKit` classes on the Mac, including `NSWindow`. It is unclear which Cocoa Touch classes are affected.

Creating an ARC-less Project from Xcode Templates

Regardless of ARC's advantages, there are practical reasons you may want to bypass ARC while developing applications. Committing to ARC may mean refactoring a large codebase. You may not have time to make the move and still meet development

From the Library of Eric Berna

deadlines. Or, you may simply want to delay learning the new ARC concepts until a more convenient time.

Follow these steps to convert Xcode's Empty Application template back to pre-ARC development standards:

1. Create a new Xcode project using the Empty Application template.
2. Select the project file in the Project Navigator. Choose Target > Build Settings and search for Code Generation or Automated Reference Counting. As shown in Figure 2-6 and discussed previously, switch Objective-C Automatic Reference Counting to No.
3. Edit your `window` property in the AppDelegate.h file from `strong` to `retain`.
4. In the AppDelegate.m file, add `autorelease` to the `window` property assignment. This is optional because the `window` property will generally persist throughout your entire application life cycle, regardless of whether the retain count is +1 (with `autorelease` and the retained property) or +2 (without `autorelease`). Adding `autorelease` is really for form's sake here.

After following these steps, you'll have created a backward-compatible project that will use pre-ARC memory management while still taking advantage of all new iOS APIs and creation tools.

ARC Rules

ARC requires that you obey certain rules that do not apply when compiling without ARC. These rules include:

- Do not implement *or* invoke `retain`, `release`, `retainCount`, or `autorelease`.
- Do not explicitly invoke `dealloc`; even `[super dealloc]` is prohibited. You can freely override the `dealloc` method to perform any standard task cleanup such as freeing `malloc`'ed memory, disposing of system sounds, or calling Core Foundation release (`CFRelease`), as needed. The compiler automatically calls `[super dealloc]` for you, regardless of whether you override `dealloc` in your subclasses or not.
- Never use `NSAllocateObject` or `NSDeallocateObject`.
- Do not cast between `(id)` and `(void *)`. Instead, cast to Core Foundation object references and pass those as function arguments.
- Do not use `NSZone`—or memory zones at all.
- Do not use `NSAutoreleasePool`. Use `@autoreleasepool` blocks instead.
- You must assign the result of `[super init]`, typically as `self = [super init]`. You probably already do this, as well as checking to see whether `self` is `nil`. Just keep doing that.

From the Library of Eric Berna

- Do not use object pointers in C structures; use Objective-C classes to manage your objects instead.

In addition to these rules, ARC uses standard preexisting rules for memory management:

- Methods named with `new` and `copy` return values with +1 retain counts unless explicitly overridden by `NS_RETURN_NOT_RETAINED`. ARC checks for proper camel case for method names using the phrase `copy`; for example, `mutableCopy` triggers a +1 retain count but not `copyright` or `copycat`. For the most part, these retain counts are hidden from your code due to ARC's memory management.
- Convenience methods return autoreleased values.

Using ARC with Core Foundation and Toll Free Bridging

Core Foundation objects are a reality for anyone who develops iOS applications outside of the most limited `UIKit` situations. These classes use C-based APIs rather than Objective-C. Core Foundation can be used directly, using CF-style classes such as `CFStringRef`, or by using any iOS frameworks that adhere to Core Foundation conventions, such as Core Graphics, Core Text, and Address Book, among others.

ARC does not integrate into Core Foundation object management. That leaves the memory management in your hands, as old-style Objective-C did. If you allocate, you must release. `CFRetain()` and `CFRelease()` continue with the same roles they held prior to ARC's debut.

Toll Free Bridging allows you to work with Core Foundation objects in an Objective-C context, and vice versa. To cast an object from one to the other and back, use the qualifiers and function demonstrated in this section.

Casting between Objective-C and Core Foundation

ARC offers C-style casting with its standard parentheses form. You can cast to and from Objective-C and Core Foundation objects, specifying the destination type you wish to cast to. ARC provides three basic ways to cast, depending on how you want to handle retain counts and what kinds of objects are used as the source and destination.

Basic Casting

The simplest bridging cast lets you move between the Objective-C and Core Foundation worlds. Use `__bridge` to cast an operand, specifying what type to cast to within the parentheses. These examples demonstrate casting in both directions. In the first, an `NSData` object is cast to `CFDataRef`. In the second, a `CFDataRef` object is cast to `NSData`.

```
CFDataRef theData = (__bridge CFDataRef) data;
NSData *otherData = (__bridge NSData *) theData;
```

From the Library of Eric Berna

This bridging *only* casts to and from Objective-C and Core Foundation. Casting in this manner between objects that use the same kind of retainability (that is, Objective-C to Objective-C or CF to CF) produces ill-formed casts.

ARC conversion automates many basic bridging details for you. For example, a line of code that performs a simple `(id)` cast:

```
self.coreLayer = [CALayer layer];
self.coreLayer.contents = (id) myUIImage.CGImage;
```

would be translated to this bridge:

```
self.coreLayer.contents = (__bridge id) myUIImage.CGImage;
```

Adjusting Retain Counts

Two additional bridging casts adjust retain counts during assignment. Apple recommends against using retain-altering bridging casts directly. To improve code readability, use their equivalent CF macros instead. Each of the following examples uses the CF macros followed by the alternative bridging cast.

The CF bridging macros used in these examples respectively produce `id` and `CFTypeRef` results. Cast their results for assignment to other types (in the same CF or Objective-C family), as demonstrated in these examples.

Transfers

The `bridge_transfer` cast releases an object as it is assigned, allowing its recipient to retain it. Its macro equivalent is `CFBridgingRelease()`. Use this cast to assign Core Foundation objects to Objective-C. It transfers the responsibility of a +1 object to ARC to manage.

The most common use for transfer bridging assigns the results of a CF creation function that returns a +1 object. Prior to ARC, the results of `CFUUIDCreateString()` might have been cast directly to an `NSString` and then autoreleased:

```
CFUUIDRef theUUID = CFUUIDCreate(kCFAllocatorDefault);
NSString *uuidString =
    [(NSString *)CFUUIDCreateString(NULL, theUUID) autorelease];
CFRelease(theUUID);
NSLog(@"UUID is %@", uuidString);
```

Bridging handles the memory management issues for you. It transfers ownership from a retained Core Foundation creation function to a local ARC Objective-C recipient. Here, the `strong` variable `uuidString` retains the resulting string. It will continue to do so throughout its scope, releasing it when it is finished.

```
CFUUIDRef theUUID = CFUUIDCreate(kCFAllocatorDefault);
NSString *uuidString =
    (NSString *) CFBridgingRelease(CFUUIDCreateString(NULL, theUUID)).
```

From the Library of Eric Berna

```
CFRelease(theUUID);
NSLog(@"UUID is %@", uuidString);
```

You can use a direct bridging cast instead of the macro, as follows. Although it is lexically correct, Apple considers this code as slightly less readable than its alternative:

```
NSString *uuidString = (__bridge_transfer NSString *)
    CFUUIDCreateString(NULL, theUUID));
```

Retains

The `__bridge_retained` cast retains an object as it is assigned. Use this cast to assign Objective-C objects to Core Foundation. On transfer, ARC retains the object and the recipient takes responsibility for balancing that +1 retain. Use retained bridging to hold onto objects that might otherwise be released, such as those returned from class convenience methods, until your CF code finishes using them. Use `CFRelease()` to release bridge-retained objects:

```
// theDate receives retained object
CFDateRef theDate = (CFDateRef) CFBridgingRetain([NSDate date]);

// ...use theDate here...

CFRelease(theDate);
```

As with transfers, you can invoke the retain cast directly:

```
CFDateRef theDate = (__bridge_retained CFDate) [NSDate date];
```

Choosing a Bridging Approach

Here's a quick way to determine which bridging approach to use when casting between Objective-C and Core Foundation objects:

- If you're assigning the result of a Core Foundation function that returns an object with a +1 retain count (for example, any `copy`, `new`, or `create` function) that you would normally balance with `CFRelease()`, use `CFBridgingRelease()` or a transfer bridge cast, ARC takes control of its object management and you do not have to release it yourself.
- If you're using an autoreleased Objective-C object in a Core Foundation context, use `CFBridgingRetain()` or a retained bridge cast to hold onto it until you're ready to `CFRelease()` it.
- If you're moving between an Objective-C variable to a CF one, or vice versa, and you do not need to retain or release in that assignment, use a simple `__bridge` cast.

In summary, unless you're using CF creation/Objective-C convenience methods, just use `__bridge`. Otherwise, CF to Objective-C is `__bridge_transfer` and Objective-C to CF is `__bridge_retain` with a `CFRelease()` to follow.

From the Library of Eric Berna

Runtime Workarounds

The following casting calls are part of the Objective-C runtime and were used in early beta releases of ARC, before Apple implemented bridging casts. These functions *may* continue to work moving forward, even as Apple introduces new approaches. Or, as Apple's documentation suggests, they may be removed as valid ways to cast between CF and Objective-C `id`. Developer beware.

As with the standard bridge cast, `objc_unretainedObject()` returns an `id`, leaving the CF object's retain count unchanged.

```
CFUUIDRef theUUID = CFUUIDCreate(kCFAllocatorDefault);
NSString *uuidString = objc_unretainedObject(
    CFUUIDCreateString(NULL, theUUID));
CFRelease(theUUID);
NSLog(@"UUID is %@", uuidString);
```

The retain-adjusted bridging casts are also represented through Objective-C runtime functions. Use `objc_retainedObject()` to transfer retain management to ARC, while consuming a retain on the object in question, avoiding a secondary `CFRelease()`. The `objc_unretainedPointer()` function converts from `id` to a CF object, without transferring ownership of that object:

```
- (NSDictionary *) propertyListFromData: (NSData *) data
{
    CFStringRef errorString;
    CFPropertyListRef plist =
        CFPropertyListCreateFromXMLData(kCFAllocatorDefault,
           (CFDataRef) objc_unretainedPointer(data),
           kCFPropertyListMutableContainers, &errorString);
    if (!plist) {NSLog(@"%@", errorString); return nil;}
    NSDictionary *dict = objc_retainedObject(plist);
    return dict;
}
```

Conversion Issues

Apple warns that conversion to CF is particularly dangerous if the passed object is the only reference to that object. It recommends explicitly retaining and then releasing the object around its use. Here's an example:

```
- (NSDictionary *) propertyListFromData: (NSData *) data
{
    CFStringRef errorString;
    CFDataRef theData = CFRetain(objc_unretainedPointer(data));
    CFPropertyListRef plist = CFPropertyListCreateFromXMLData(
        kCFAllocatorDefault, theData, kCFPropertyListMutableContainers,
        &errorString);
    CFRelease(theData);
    if (!plist) {NSLog(@"%@", errorString); return nil;}
```

From the Library of Eric Berna

```
    NSDictionary *dict = objc_retainedObject(plist);
    return dict;
}
```

Tips and Tricks for Working with ARC

Here are a few final tips to keep in mind when working with ARC.

- If you're not using a storyboard entry point, use the name of your primary application delegate class in your `main()` function in main.m rather than simply calling `UIApplicationMain(argc, argv, nil, nil)`. The fourth argument should be a string with the name of your class—for example, `@"MyAppDelegate"`.
- Declare `UIWindow` in your application delegate and set the window's `rootViewController` property to your primary view controller. (It can be a navigation controller as well as a `UIViewController` subclass.) The window variable defaults to `strong` and will retain your `window`.
- For the most part, ARC has become far more clever in dealing with `CGImages` returned from functions and other non-Objective-C items in terms of assigning them to objects, but this remains a feature that continues to evolve.
- You can use `CFAutorelease` tricks with ARC CF objects so long as you compile that code using MRR and then call the function from your ARC-compiled methods.

```
// Compile with -fno-objc-arc
#define _CFAutorelease(obj) ([(id)obj autorelease])
CFTypeRef CFAutorelease(CFTypeRef anObject)
{
    return _CFAutorelease(anObject);
}
```

Crafting Singletons

The `UIApplication` and `UIDevice` classes let you access information about the currently running application and the device hardware it is running on. They do so by offering singletons—that is, sole instances of classes in the current process. For example, `[UIApplication sharedApplication]` returns a singleton that can report information about the delegate it uses, whether the application supports shake-to-edit features, what windows are defined by the program, and so forth.

Most singleton objects act as control centers. They coordinate services, provide key information, and direct external access, among other functionality. If you have a need for centralized functionality, such as a manager that accesses a web service, a singleton approach ensures that all parts of your application coordinate with the same central manager.

From the Library of Eric Berna

Building a singleton takes very little code. You define a static shared instance inside the class implementation and add a class method pointing to that instance. In this snippet, the instance is built the first time it is requested:

```
@implementation MyClass
static MyClass __strong *sharedInstance = nil;

+( MyClass *) sharedInstance {
    if(!sharedInstance)
        sharedInstance = [[self alloc] init];
    return sharedInstance;
}

// Class behavior defined here

@end
```

To use this singleton, call `[ViewIndexer sharedInstance]`. This returns the shared object and lets you access any behavior that the singleton provides. For thread-safe use, you may want to use a more guarded approach to singleton creation, such as this one:

```
+ (MyClass *)sharedInstance
{
    static dispatch_once_t pred;
    dispatch_once(&pred, ^{
        sharedInstance = [[self alloc] init];
    });
    return sharedInstance;
}
```

Categories (Extending Classes)

Objective-C's built-in capability to expand already-existing classes is one of its most powerful features. This behavioral expansion is called a "category." Categories extend class functionality without subclassing. You choose a descriptive expansion name, build a header, and then implement the functionality in a method file. Categories add methods to existing classes even if you did not define that class in the first place and do not have the source code for that class.

To build a category, you declare a new interface. Specify the category name (it's arbitrary) within parentheses, as you see here. List any new public methods and properties and save the header file. This `Orientation` category expands the `UIDevice` class, which is the SDK class responsible for reporting device characteristics, including orientation, battery level, and the proximity sensor state. This interface adds a single property to `UIDevice`, returning a read-only Boolean value. The new `isLandscape` property reports back whether the device is currently using a landscape orientation.

From the Library of Eric Berna

```
@interface UIDevice (Orientation)
@property (nonatomic, readonly) BOOL isLandscape;
@end
```

You cannot add new instance variables to a category interface as you could when subclassing. You are instead expanding a class's behavior, as shown in the source code of Listing 2-5. The code implements the landscape check by looking at the standard `UIDevice` orientation property.

You might use the new property like this:

```
NSLog(@"The device orientation is %@"
    [UIDevice currentDevice].isLandscape ? @"Landscape" : @"Portrait");
```

Here, the landscape orientation check integrates seamlessly into the SDK-provided `UIDevice` class via a property that did not exist prior to expanding the class. Just FYI, `UIKit` does offer a pair of device orientation macros (`UIDeviceOrientationIsPortrait` and `UIDeviceOrientationIsLandscape`), but you must pass these an orientation value, which you have to poll from the device.

When working with actual production code, do not use any method or property name that Apple may eventually add to its classes. Instead of `isLandscape`, you might use `esIsLandscape`, for example, to guard against such potential overlaps. In this example, "es" is my initials. You might use the initials of your company or some other similar prefix.

> **Note**
>
> In addition to adding new behavior to existing classes, categories also let you group related methods into separate files for classes you build yourself. For large, complex classes, this helps increase maintainability and simplifies the management of individual source files. Please note that when you add a category method that duplicates an existing method signature, the Objective-C runtime uses your implementation and overrides the original.

Listing 2-5 **Building an Orientation Category for the `UIDevice` Class**

```
@interface UIDevice (Orientation)
@property (nonatomic, readonly) BOOL isLandscape;
@end

@implementation UIDevice (Orientation)
- (BOOL) isLandscape
{
    return
        (self.orientation == UIDeviceOrientationLandscapeLeft) ||
        (self.orientation == UIDeviceOrientationLandscapeRight);
}
@end
```

From the Library of Eric Berna

Protocols

Chapter 1, "Introducing the iOS SDK," introduced the notion of delegates. Delegates implement details that cannot be determined when a class is first defined. For example, a table knows how to display rows of cells, but it can't know what to do when a cell is tapped. The meaning of a tapped row changes with whatever application implements that table. A tap might open another screen, send a message to a web server, or perform any other imaginable result. Delegation lets the table communicate with a smart object that is responsible for handling those taps but whose behavior is written at a completely separate time from when the table class itself is created.

Delegation basically provides a language that mediates contact between an object and its handler. A table tells its delegate "I have been tapped," "I have scrolled," and other status messages. The delegate then decides how to respond to these messages, producing updates based on its particular application semantics.

Data sources operate the same way, but instead of mediating action responses, data sources provide data on demand. A table asks its data source, "What information should I put into cell 1 and cell 2?" The data source responds with the requested information. Like delegation, data sourcing lets the table place requests to an object that is built to understand those demands.

In Objective-C, both delegation and data sourcing are produced by a system called protocols. **Protocols** define *a priori* how one class can communicate with another. They contain a list of methods that are defined outside any class. Some of these methods are required. Others are optional. Any class that implements the required methods is said to "conform to the protocol."

Defining a Protocol

Imagine, if you would, a jack-in-the box toy. This is a small box with a handle. When you turn the crank, music plays. Sometimes a puppet (called the "jack") jumps out of the box. Now imagine implementing that toy (or a rough approximation) in Objective-C. The toy provides one action, turning the crank, and there are two possible outcomes: the music or the jack.

Now consider designing a programmatic client for that toy. It could respond to the outcomes, perhaps, by gradually increasing a boredom count when more music plays or reacting with surprise when the jack finally bounces out. From an Objective-C point of view, your client needs to implement two responses: one for music, another for the jack. Here's a client protocol you might build:

```
@protocol JackClient <NSObject>
- (void) musicDidPlay;
- (void) jackDidAppear;
@end
```

This protocol declares that to be a client of the toy, you must respond to music playing and the jack jumping out of the box. Listing these methods inside an `@protocol`

From the Library of Eric Berna

container defines the protocol. All the methods listed here are required unless you specifically declare them as `@optional`, as you read about in the next sections.

Incorporating a Protocol

Next, imagine designing a class for the toy itself. It offers one action, turning the crank, and requires a second object that implements the protocol, in this case called `client`. This class interface specifies that the client needs to be some kind of object (`id`) that conforms to the `JackClient` protocol (`<JackClient>`). Beyond that, the class does not know at design time what kind of object will provide these services.

```
@interface JackInTheBox : NSObject
- (void) turnTheCrank;
@property (strong) id <JackClient> client;
@end
```

Adding Callbacks

Callbacks connect the `toy` class to its client. Because the client must conform to the `JackClient` protocol, you can send `jackDidAppear` and `musicDidPlay` messages to the object and they will compile without error. The protocol ensures that the client implements these methods. In this code, the callback method is selected randomly. The music plays approximately nine out of every ten calls, sending `musicDidPlay` to the client.

```
- (void) turnTheCrank
{
    // You need a client to respond to the crank
    if (!client) return;

    // Randomly respond to the crank turn
    int action = random() % 10;
    if (action < 1)
        [client jackDidAppear];
    else
        [client musicDidPlay];
}
```

Declaring Optional Callbacks

Protocols include two kinds of callbacks: required and optional. By default, callbacks are required. A class that conforms to the protocol must implement those methods or they produce a compiler warning. You can use the `@required` and `@optional` keywords to declare a protocol method to be of one form or the other. Any methods listed after an `@required` keyword are required; after an `@optional` keyword, they are optional. Your protocol can grow complex accordingly, as shown here:

From the Library of Eric Berna

```
@protocol JackClient <NSObject>
- (void) musicDidPlay; // required
@required
- (void) jackDidAppear; // also required
@optional
- (void) nothingDidHappen; // optional
@end
```

In practice, using more than a single `@optional` keyword is overkill. The same protocol can be declared more simply. When you don't use any optional items, skip the keyword entirely. Notice the `<NSObject>` declaration. It's required to effectively implement optional protocols. It says that a `JackClient` object conforms to and will be a kind of `NSObject`.

```
@protocol JackClient <NSObject>
- (void) musicDidPlay;
- (void) jackDidAppear;
@optional
- (void) nothingDidHappen;
@end
```

Implementing Optional Callbacks

Optional methods let the client choose whether to implement a given protocol method. They reduce the implementation burden on whoever writes that client but add a little extra work to the class that hosts the protocol definition. When you are unsure whether a class does or does not implement a method, you must test before you send a message. Fortunately, Objective-C and the `NSObject` class make it easy to do so:

```
// optional client method
if ([client  respondsToSelector: @selector(nothingDidHappen)])
    [client nothingDidHappen];
```

`NSObject` provides a `respondsToSelector:` method, which returns a Boolean `YES` if the object implements the method or `NO` otherwise. By declaring the client with `<NSObject>`, you tell the compiler that the client can handle this method, allowing you to test the client for conformance before sending the message.

Conforming to a Protocol

Classes include protocol conformance in interface declarations. A view controller that implements the `JackClient` protocol declares it between angle brackets. A class might conform to several protocols. Combine these within the brackets, separating protocol names with commas.

```
@interface TestBedViewController :
    UIViewController <JackClient>
@property (strong) JackInTheBox *jack;
@end
```

From the Library of Eric Berna

Declaring the `JackClient` protocol lets you assign the host's `client` property. The following code compiles without error because the class for `self` was declared in conformance with `JackClient`:

```
self.jack = [JackInTheBox jack];
jack.client = self;
```

Had you omitted the protocol declaration in your interface, this assignment would produce an error at compile time.

Once you include that protocol between the angle brackets, you must implement all required methods in your class. Omitting any of them produces the kind of compile-time warnings shown in Figure 2-7. Exploring each complaint in the Issues Navigator lets you discover which method is missing and what protocol that method belongs to.

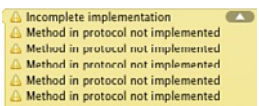


Figure 2-7 You must implement all required methods to conform to a protocol. Objective-C warns about incomplete implementations.

The majority of protocol methods in the iOS SDK are optional. Both required and optional methods are detailed exhaustively in the developer documentation. Note that protocols are documented separately from the classes they support. For example, Xcode documentation provides three distinct `UITableView` reference pages: one for the `UITableView` class, one for the `UITableViewDelegate` protocol, and another for the `UITableViewDataSource` protocol.

Foundation Classes

If you're new to Objective-C, there are a few key classes you absolutely need to be familiar with before moving forward. These include strings, numbers, and collections, and they provide critical application building blocks. The `NSString` class, for example, provides the workhorse for nearly all text manipulation in Objective-C. However, like other fundamental classes, it is not defined in Objective-C itself. It is part of the Foundation framework, which offers nearly all the core utility classes you use on a day-to-day basis.

Foundation provides over a dozen kinds of object families and hundreds of object classes. These range from value objects that store numbers and dates, to strings that store character data; from collections that store other objects, to classes that access the file system and retrieve data from URLs. Foundation is often referred to (slightly inaccurately) as Cocoa. (Cocoa and its iPhone-device-family equivalent Cocoa Touch actually include all the frameworks for OS X programming.) To master Foundation is to master Objective-C programming, and thorough coverage of the subject demands an entire book of its own.

From the Library of Eric Berna

Because this section cannot offer an exhaustive introduction to Foundation classes, you're about to be introduced to a quick-and-dirty survival overview. Here are the classes you need to know about and the absolutely rock-core ways to get started using them. You find extensive code snippets that showcase each of the classes to give you a jumping-off point if, admittedly, not a mastery of the classes involved.

Strings

Cocoa strings store character data, just as their cousins the (`char *`) C strings do. They are, however, objects and not byte arrays. Unlike C, the core `NSString` class is immutable in Cocoa. That is, you can use strings to build other strings, but you can't edit the strings you already own. String constants are delineated by quote marks and the `@` character. Here is a typical string constant, which is assigned to a string variable:

```
NSString *myString = @"A string constant";
```

Building Strings

You can build strings using formats, much as you would using `sprintf`. If you're comfortable creating `printf` statements, your knowledge transfers directly to string formats. Use the `%@` format specifier to include objects in your strings. String format specifiers are thoroughly documented in the Cocoa String Programming Guide, available via Xcode's documentation window (Command-Option-?). The most common formats are listed in Table 2-1.

```
NSString *myString = [NSString stringWithFormat:
    @"The number is %d", 5];
```

You can append strings together to create new strings. The following call outputs "The number is 522" and creates a new instance built from other strings.

```
NSLog(@"%@", [myString stringByAppendingString@"22"]);
```

Appending formats provides even more flexibility. You specify the format string and the components that build up the result:

```
NSLog(@"%@", [myString stringByAppendingFormat:@"%d, 22]);
```

> **Note**
>
> Allow custom objects to respond to `%@` delimiters by implementing the `description` method. Return an `NSString` object that describes your object's state. `NSLog()` automatically calls `description` for you on objects passed as its format parameters.

Length and Indexed Characters

Every string can report its length (via `length`) and produce an indexed character on demand (via `characterAtIndex:`). The two calls shown here output `15` and `e`,

From the Library of Eric Berna

respectively, based on the previous `@"The number is 5"` string. Cocoa characters use the unichar type, which store Unicode-style characters.

```
NSLog(@"%d", myString.length);
printf("%c", [myString characterAtIndex:2]);
```

Converting to and from C Strings

The realities of normal C programming often crop up despite working in Objective-C. Being able to move back and forth between C strings and Cocoa strings is an important skill. Convert an `NSString` to a C string either by sending `UTF8String` or `cStringUsingEncoding:`. These methods are equivalent, producing the same C-based bytes:

```
printf("%s\n", [myString UTF8String]);
printf("%s\n", [myString cStringUsingEncoding: NSUTF8StringEncoding]);
```

You can also go the other way and transform a C string into an `NSString` by using `stringWithCString:encoding:`. The examples here use UTF-8 encoding, but Objective-C supports a large range of options, including ASCII, Japanese, Latin, Windows-CP1251, and so forth. UTF-8 encoding can be used safely with ASCII text; UTF-8 was explicitly created to provide backward compatibility with ASCII, avoiding endianness and byte order marks.

```
NSLog(@"%@", [NSString stringWithCString: "Hello World"
    encoding: NSUTF8StringEncoding]);
```

Writing Strings to and Reading Strings from Files

Writing to and reading strings from the local file system offers a handy way to save and retrieve data. This snippet shows how to write a string to a file:

```
NSString *myString = @"Hello World";
NSError __autoreleasing *error;
NSString *path = [NSHomeDirectory()
    stringByAppendingPathComponent:@"Documents/file.txt"];
if (![myString writeToFile:path atomically:YES
    encoding:NSUTF8StringEncoding error:&error])
{
    NSLog(@"Error writing to file: %@", [error localizedFailureReason]);
    return;
}
NSLog(@"String successfully written to file");
```

The path for the file is `NSHomeDirectory()`, a function that returns a string with a path pointing to the application sandbox. Notice the special `append` method that properly appends the Documents/file.txt subpath.

From the Library of Eric Berna

In Cocoa, most file access routines offer an atomic option. When you set the atomically parameter to YES, the iOS SDK writes the file to a temporary auxiliary and then renames it into place. Using an atomic write ensures that the file avoids corruption.

The request shown here returns a Boolean YES if the string was written, or NO if it was not. Should the write request fail, this snippet logs the error using a language-localized description. It uses an instance of the NSError class to store that error information and sends the localizedFailureReason selector to convert the information into a human-readable form using the current (localized) language settings. Whenever iOS methods return errors, use this approach to determine which error was generated.

Reading a string from a file follows a similar form but does not return the same Boolean result. Instead, check to see whether the returned string is nil, and if so display the error that was returned.

```
NSString *inString = [NSString stringWithContentsOfFile:path
    encoding:NSUTF8StringEncoding error:&error];
if (!inString)
{
    NSLog(@"Error reading from file %@", [path lastPathComponent],
        [error localizedFailureReason]);
    return;
}
NSLog(@"String successfully read from file");
NSLog(@"%@", inString);
```

Accessing Substrings

Cocoa offers a number of ways to extract substrings from strings. Here's a quick review of some typical approaches. As you'd expect, string manipulation is a large part of any flexible API, and Cocoa offers many more routines and classes to parse and interpret strings than the few listed here. This quick NSString summary skips any discussion of NSScanner, NSXMLParser, and so forth.

Converting Strings to Arrays

You can convert a string into an array by separating its components across some repeated boundary. This example chops the string into individual words by splitting around spaces. The spaces are discarded, leaving an array that contains each number word.

```
NSString *myString = @"One Two Three Four Five Six Seven";
NSArray *wordArray = [myString componentsSeparatedByString: @" "];
NSLog(@"%@", wordArray);
```

Requesting Indexed Substrings

You can request a substring from the start of a string to a particular index, or from an index to the end of the string. These two examples return @"One Two" and @"Two Three Four Five Six Seven", respectively, using the To and From versions of the indexed substring request. As with standard C, array and string indexes start at 0.

From the Library of Eric Berna

```
NSString *sub1 = [myString substringToIndex:7];
NSLog(@"%@", sub1);

NSString *sub2 = [myString substringFromIndex:4];
NSLog(@"%@", sub2);
```

Generating Substrings from Ranges

Ranges let you specify exactly where your substring should start and stop. This snippet returns `@"Tw"`, starting at character 4 and extending two characters in length. `NSRange` provides a structure that defines a section within a series. You use ranges with indexed items such as strings and arrays.

```
NSRange r;
r.location = 4;
r.length = 2;
NSString *sub3 = [myString substringWithRange:r];
NSLog(@"%@", sub3);
```

The `NSMakeRange()` function takes two arguments: a location and a length returning a range. `NSMakeRange(4,2)` is equivalent to the range used in this example.

Search and Replace with Strings

With Cocoa, you can easily search a string for a substring. Searches return a range, which contain both a location and a length. Always check the range location. The location `NSNotFound` means the search failed. This returns a range location of 18, with a length of 4:

```
NSRange searchRange = [myString rangeOfString:@"Five"];
if (searchRange.location != NSNotFound)
    NSLog(@"Range location: %d, length: %d", searchRange.location,
searchRange.length);
```

Once you've found a range, you can replace a subrange with a new string. The replacement string does not need to be the same length as the original, thus the result string may be longer or shorter than the string you started with.

```
NSLog(@"%@", [myString stringByReplacingCharactersInRange:
    searchRange withString: @"New String"]);
```

A more general approach lets you replace all occurrences of a given string. This snippet produces `@"One * Two * Three * Four * Five * Six * Seven"` by swapping out each space for a space-asterisk-space pattern:

```
NSString *replaced = [myString stringByReplacingOccurrencesOfString:
    @" " withString: @" * "];
NSLog(@"%@", replaced);
```

From the Library of Eric Berna

Changing Case

Cocoa provides three simple methods that change a string's case. Here, these three examples produce a string all in uppercase, all in lowercase, and one where every word is capitalized (`"Hello World. How Do You Do?"`). Because Cocoa supports case-insensitive comparisons (`caseInsensitiveCompare:`), you rarely need to apply case conversions when testing strings against each other.

```
NSString *myString = @"Hello world. How do you do?";
NSLog(@"%@", [myString uppercaseString]);
NSLog(@"%@", [myString lowercaseString]);
NSLog(@"%@", [myString capitalizedString]);
```

Testing Strings

The iOS SDK offers many ways to compare and test strings. The three simplest check for string equality and match against the string prefix (the characters that start the string) and suffix (those that end it). More complex comparisons use `NSComparisonResult` constants to indicate how items are ordered compared with each other.

```
NSString *s1 = @"Hello World";
NSString *s2 = @"Hello Mom";
NSLog(@"%@ %@ %@", s1, [s1 isEqualToString:s2] ?
    @"equals" : @"differs from", s2);
NSLog(@"%@ %@ %@", s1, [s1 hasPrefix:@"Hello"] ?
    @"starts with" : @"does not start with", @"Hello");
NSLog(@"%@ %@ %@", s1, [s1 hasSuffix:@"Hello"] ?
    @"ends with" : @"does not end with", @"Hello");
```

Extracting Numbers from Strings

Convert strings into numbers by using a `Value` method. These examples return 3, 1, 3.141592, and 3.141592, respectively:

```
NSString *s1 = @"3.141592";
NSLog(@"%d", [s1 intValue]);
NSLog(@"%d", [s1 boolValue]);
NSLog(@"%f", [s1 floatValue]);
NSLog(@"%f", [s1 doubleValue]);
```

Mutable Strings

The `NSMutableString` class is a subclass of `NSString`. It offers you a way to work with strings whose contents can be modified. Once instantiated, you can append new contents to the string, which allows you to grow results before returning from a method. This example displays `"Hello World. The results are in now."`

```
NSMutableString *myString = [NSMutableString stringWithString:
    @"Hello World. "];
[myString appendFormat:@"The results are %@ now.", @"in"];
NSLog(@"%@", myString);
```

From the Library of Eric Berna

Numbers and Dates

Foundation offers a large family of value classes. Among these are numbers and dates. Unlike standard C floats, integers, and so forth, these elements are all objects. They can be allocated and released, and used in collections such as arrays, dictionaries, and sets. The following examples show numbers and dates in action, providing a basic overview of these classes.

Working with Numbers

The `NSNumber` class lets you treat numbers as objects. You can create new `NSNumber` instances using a variety of convenience methods—namely `numberWithInt:`, `numberWithFloat:`, `numberWithBool:`, and so forth. Once they are set, you extract those values via `intValue`, `floatValue`, `boolValue`, and so on. Use normal C-based math to perform your calculations.

You are not limited to extracting the same data type an object was set with. You can set a float and extract the integer value, for example. Numbers can also convert themselves into strings.

```
NSNumber *number = [NSNumber numberWithFloat:3.141592];
NSLog(@"%d", [number intValue]);
NSLog(@"%@", [number stringValue]);
```

One of the biggest reasons for using `NSNumber` objects rather than `int`s, `NSInteger`s, `float`s, and so forth, is that they are objects. You can use them with Cocoa routines and classes. For example, you cannot set a user default (that is, a preference value) to, say, the integer 23, as in "You have used this program 23 times." You can, however, store an object `[NSNumber numberWithInt:23]` and later recover the integer value from that object to produce the same user message.

If, for some reason, you must stick with C-style variables or byte arrays, consider using `NSValue`. The `NSValue` class allows you to encapsulate C data, including `int`, `float`, `char`, structures, and pointers into an Objective-C wrapper.

For the most part, you'll want to use `NSInteger` and `NSUInteger` typedefs instead of `int` and `uint` in your Objective-C applications. These provide architecture-safe versions of the corresponding C types. Tied to the underlying architecture, the size of `NSInteger` and `NSUInteger` always matches any valid pointer that might be used on that platform.

> **Note**
>
> The `NSDecimalNumber` class provides an object-oriented wrapper for base-10 arithmetic.

Working with Dates

As with standard C and `time()`, `NSDate` objects use the number of seconds since an epoch (that is, a standardized universal time reference) to represent the current date. The iOS SDK epoch was at midnight on January 1, 2001. The standard UNIX epoch took place at midnight on January 1, 1970.

From the Library of Eric Berna

Each `NSTimeInterval` represents a span of time in seconds, stored with subsecond floating-point precision. The following code shows how to create a new date object using the current time and how to use an interval to reference sometime in the future (or past):

```
// current time
NSDate *date = [NSDate date];

// time 10 seconds from now
date = [NSDate dateWithTimeIntervalSinceNow:10.0f];
```

You can compare dates by setting or checking the time interval between them. This snippet forces the application to sleep until 5 seconds into the future and then compares the date to the one stored in date:

```
// Sleep 5 seconds and check the time interval
[NSThread sleepUntilDate:[NSDate dateWithTimeIntervalSinceNow:5.0f]];
NSLog(@"Slept %f seconds", [[NSDate date] timeIntervalSinceDate:date]);
```

The standard description method for dates returns a somewhat human-readable string, showing the current date and time:

```
// Show the date
NSLog(@"%@" [date description]);
```

To convert dates into fully formatted strings rather than just using the default description, use an instance of `NSDateFormatter`. You specify the format (for example, YY for two-digit years, and YYYY for four-digit years) using the object's date format property. A useful list of format specifiers is offered at the end of Chapter 11, "Creating and Managing Table Views." You can find a more complete specifier write-up at http://unicode.org/reports/tr35/tr35-10.html#Date_Format_Patterns.

In addition to producing formatted output, this class can also be used to read preformatted dates from strings, although that is left as an exercise for the reader.

```
// Produce a formatted string representing the current date
NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
formatter.dateFormat = @"MM/dd/YY HH:mm:ss";
NSString *timestamp = [formatter stringFromDate:[NSDate date]];
NSLog(@"%@", timestamp);
```

> **Note**
>
> See the `NSDate` category at http://github.com/erica for examples of using common date scenarios (such as yesterday, tomorrow, and so forth). Chapter 11 contains a handy table of date formatter codes.

Timers

When working with time, you may need to request that some action occur in the future. Cocoa provides an easy-to-use timer that triggers at an interval you specify; use the

From the Library of Eric Berna

NSTimer class. The timer shown here triggers after 1 second and repeats until the timer is disabled:

```
[NSTimer scheduledTimerWithTimeInterval: 1.0f target: self
    selector: @selector(handleTimer:) userInfo: nil repeats: YES];
```

Each time the timer activates, it calls its target sending the selector message it was initialized with. The callback method takes one argument (notice the single colon), which is the timer itself. To disable a timer, send it the invalidate message; this releases the timer object and removes it from the current runloop.

```
- (void) handleTimer: (NSTimer *) timer
{
    printf("Timer count: %d\n", count++);
    if (count > 3)
    {
        [timer invalidate];
        printf("Timer disabled\n");
    }
}
```

Recovering Information from Index Paths

The NSIndexPath class is used with iOS tables. It stores the section and row number for a user selection (that is, when a user taps on the table). When provided with index paths, you can recover these numbers via the row and section properties. Learn more about this class and its use in Chapter 11.

Collections

The iOS SDK primarily uses three kinds of collections: arrays, dictionaries, and sets. Arrays act like C arrays. They provide an indexed list of objects, which you can recover by specifying which index to look at. Dictionaries, in contrast, store values that you can look up by keys. For example, you might store a dictionary of ages, where Dad's age is the NSNumber 57, and a child's age is the NSNumber 15. Sets offer an unordered group of objects and are usually used in the iOS SDK in connection with recovering user touches from the screen. Each of these classes offers regular and mutable versions, just as the NSString class does.

Building and Accessing Arrays

Create arrays using the arrayWithObjects: convenience method, which returns an autoreleased array. When you're calling this method, list any objects you want added to the array and finish the list with nil. (If you do not include nil in your list, you'll experience a runtime crash.) You can add any kind of object to an array, including other arrays and dictionaries. This example showcases the creation of a three-item array:

```
NSArray *array = [NSArray arrayWithObjects:@"One", @"Two", @"Three", nil];
```

From the Library of Eric Berna

The count property returns the number of objects in an array. Arrays are indexed starting with 0, up to one less than the count. Attempting to access [array objectAtIndex: array.count] causes an "index beyond bounds" exception and crashes. So always use care when retrieving objects, making sure not to cross either the upper or lower boundary for the array.

```
NSLog(@"%d", array.count);
NSLog(@"%@", [array objectAtIndex:0]);
```

Mutable arrays are editable. The mutable form of NSArray is NSMutableArray. With mutable arrays, you can add and remove objects at will. This snippet copies the previous array into a new mutable one and then edits the array by adding one object and removing another one. This returns an array of [@"One", @"Two", @"Four]:

```
NSMutableArray *mutableArray = [NSMutableArray arrayWithArray:array];
[mutableArray addObject:@"Four"];
[mutableArray removeObjectAtIndex:2];
NSLog(@"%@", mutableArray);
```

Whether or not you're working with mutable arrays, you can always combine arrays to form a new version containing the components from each. No checks are done about duplicates. This code produces a six-item array, including one, two, and three from the original array, and one, two, and four, from the mutable array:

```
NSLog(@"%@", [array arrayByAddingObjectsFromArray: mutableArray]);
```

Checking Arrays

You can test whether an array contains an object and recover the index of a given object. This code searches for the first occurrence of "Four" and returns the index for that object. The test in the if-statement ensures that at least one occurrence exists.

```
if ([mutableArray containsObject:@"Four"])
      NSLog(@"The index is %d",
        [mutableArray indexOfObject:@"Four"]);
```

Converting Arrays into Strings

As with other objects, sending description to an array returns an NSString that describes an array. In addition, you can use componentsJoinedByString: to transform an NSArray into a string. The following code returns @"One Two Three":

```
NSArray *array = [NSArray arrayWithObjects:@"One", @"Two", @"Three", nil];
NSLog(@"%@", [array componentsJoinedByString:@" "]);
```

Building and Accessing Dictionaries

NSDictionary objects store keys and values, enabling you to look up objects using strings. The mutable version of dictionaries, NSMutableDictionary, lets you modify these dictionaries by adding and removing elements on demand. In iOS programming,

From the Library of Eric Berna

you use the mutable class more often than the static one, so these examples showcase mutable versions.

Creating Dictionaries

Use the dictionary convenience method to create a new mutable dictionary, as shown here. This returns a new initialized dictionary that you can start to edit. Populate the dictionary using `setObject:forKey:`.

```
NSMutableDictionary *dict = [NSMutableDictionary dictionary];
[dict setObject:@"1" forKey:@"A"];
[dict setObject:@"2" forKey:@"B"];
[dict setObject:@"3" forKey:@"C"];
NSLog(@"%@", [dict description]);
```

Searching Dictionaries

Searching the dictionary means querying the dictionary by key name. Use `objectForKey:` to find the object that matches a given key. When a key is not found, the dictionary returns `nil`. This snippet returns `@"1"` and `nil`:

```
NSLog(@"%@", [dict objectForKey: @"A"]);
NSLog(@"%@", [dict objectForKey: @"F"]);
```

Replacing Objects

When you set a new object for the same key, Cocoa replaces the original object in the dictionary. This code replaces `"3"` with `"foo"` for the key `"C"`:

```
[dict setObject:@"foo" forKey:@"C"];
NSLog(@"%@", [dict objectForKey:@"C"]);
```

Removing Objects

You can also remove objects from dictionaries. This snippet removes the object associated with the `"B"` key. Once removed, both the key and the object no longer appear in the dictionary.

```
[dict removeObjectForKey:@"B"];
```

Listing Keys

Dictionaries can report the number of entries they store plus they can provide an array of all the keys currently in use. This key list lets you know what keys have already been used. It lets you test against the list before adding an item to the dictionary, avoiding overwriting an existing key/object pair.

```
NSLog(@"The dictionary has %d objects", [dict count]);
NSLog(@"%@", [dict allKeys]);
```

From the Library of Eric Berna

Accessing Set Objects

Sets store unordered collections of objects. You encounter sets when working with the iPhone device family's multitouch screen. The `UIView` class receives finger movement updates that deliver touches as an `NSSet`. To work with touches, you typically issue `allObjects` to the set and work with the array that's returned. After the set is converted to an array, use standard array calls to list, query, and iterate through the touches. You can also use fast enumeration directly with sets.

Memory Management with Collections

Arrays, sets, and dictionaries automatically retain objects when they are added and release those objects when they are removed from the collection. Releases are also sent when the collection is deallocated. Collections do not copy objects. Instead, they rely on retain counts to hold onto objects and use them as needed.

Writing Out Collections to File

Both arrays and dictionaries can store themselves into files using `writeToFile:atomically:` methods so long as the types within the collections belong to the set of `NSData`, `NSDate`, `NSNumber`, `NSString`, `NSArray`, and `NSDictionary`. Pass the path as the first argument, and a `Boolean` as the second. As when saving strings, the second argument determines whether the file is first stored to a temporary auxiliary and then renamed into place. The method returns a Boolean value: `YES` if the file was saved, `NO` if not. Storing arrays and dictionaries creates standard property lists files.

```
NSString *path = [NSHomeDirectory()
    stringByAppendingPathComponent:@"Documents/ArraySample.txt"];
if ([array writeToFile:path atomically:YES])
    NSLog(@"File was written successfully");
```

To recover an array or dictionary from file, use the convenience methods `arrayWithContentsOfFile:` and `dictionaryWithContentsOfFile:`. If the methods return `nil`, the file could not be read.

```
NSArray *newArray = [NSArray arrayWithContentsOfFile:path];
NSLog(@"%@", newArray);
```

Building URLs

`NSURL` objects point to resources. These resources can refer to both local files and to URLs on the Web. Create `url` objects by passing a string to class convenience functions. Separate functions have been set up to interpret each kind of URL. Once built, however, `NSURL` objects are interchangeable. Cocoa does not care if the resource is local or points to an object only available via the Net. This code demonstrates building URLs of each type, path and Web:

From the Library of Eric Berna

```
NSString *path = [NSHomeDirectory()
    stringByAppendingPathComponent:@"Documents/foo.txt"];
NSURL *url1 = [NSURL fileURLWithPath:path];
NSLog(@"%@", url1);

NSString *urlpath = @"http://ericasadun.com";
NSURL *url2 = [NSURL URLWithString:urlpath];
NSLog(@"%d characters read",
    [[NSString stringWithContentsOfURL:url2] length]);
```

Working with `NSData`

If `NSString` objects are analogous to zero-terminated C strings, then `NSData` objects correspond to buffers. `NSData` provides data objects that store and manage bytes. Often, you fill `NSData` with the contents of a file or URL. The data returned can report its length, letting you know how many bytes were retrieved. This snippet retrieves the contents of a URL and prints the number of bytes that were read:

```
NSData *data = [NSData dataWithContentsOfURL:url2];
NSLog(@"%d", [data length]);
```

To access the core byte buffer that underlies an `NSData` object, use `bytes`. This returns a `(const void *)` pointer to the actual data stored by the `NSData` object.

As with many other Cocoa objects, you can use the standard `NSData` version of the class or its mutable child, `NSMutableData`. Most Cocoa programs that access the Web, particularly those that perform asynchronous downloads, pull in a bit of data at a time. For those cases, `NSMutableData` objects prove useful. You can keep growing mutable data by issuing `appendData:` to add the new information as it is received.

File Management

The iOS SDK's file manager is a singleton provided by the `NSFileManager` class. It can list the contents of folders to determine what files are found and perform basic file system tasks. The following snippet retrieves a file list from two folders. First, it looks in the sandbox's Documents folder and then inside the application bundle itself.

```
NSFileManager *fm = [NSFileManager defaultManager];

// List the files in the sandbox Documents folder
NSString *path = [NSHomeDirectory() stringByAppendingPathComponent:@"Documents"];
NSLog(@"%@", [fm directoryContentsAtPath:path]);

// List the files in the application bundle
path = [[NSBundle mainBundle] bundlePath];
NSLog(@"%@", [fm directoryContentsAtPath:path]);
```

From the Library of Eric Berna

Note the use here of `NSBundle`. It lets you find the application bundle and pass its path to the file manager. You can also use `NSBundle` to retrieve the path for any item included in your app bundle. (You cannot, however, write to the application bundle at any time.) This code returns the path to the application's Default.png image. Note that the file and extension names are separated and that each is case sensitive.

```
NSBundle *mainBundle = [NSBundle mainBundle];
NSLog(@"%@", [mainBundle pathForResource:@"Default" ofType:@"png"]);
```

The file manager offers a full suite of file-specific management. It can move, copy, and remove files as well as query the system for file traits and ownership. Here are some examples of the simpler routines you may use in your applications:

```
// Create a file
NSString *docspath = [NSHomeDirectory()
    stringByAppendingPathComponent:@"Documents"];
NSString *filepath = [NSHomeDirectory()
    stringByAppendingPathComponent:@"Documents/testfile"];
NSArray *array = [@"One Two Three" componentsSeparatedByString:@" "];
[array writeToFile:filepath atomically:YES];
NSLog(@"%@", [fm directoryContentsAtPath:docspath]);

// Copy the file
NSString *copypath = [NSHomeDirectory()
    stringByAppendingPathComponent:@"Documents/copied"];
if (![fm copyItemAtPath:filepath toPath:copypath error:&error])
{
    NSLog(@"Copy Error: %@", [error localizedFailureReason]);
    return;
}

NSLog(@"%@", [fm directoryContentsAtPathdocspath]);
// Move the file
NSString *newpath = [NSHomeDirectory()
    stringByAppendingPathComponent:@"Documents/renamed"];
if (![fm moveItemAtPath:filepath toPath:newpath error:&error])
{
    NSLog(@"Move Error: %@", [error localizedFailureReason]);
    return;
}
NSLog(@"%@", [fm directoryContentsAtPath:docspath]);

// Remove a file
if (![fm removeItemAtPath:copypath error:&error])
{
    NSLog(@"Remove Error: %@", [error localizedFailureReason]);
    return;
}
NSLog(@"%@", [fm directoryContentsAtPath:docspath]);
```

From the Library of Eric Berna

> **Note**
> As another convenient file trick, use tildes in path names (for example, "~/Library/Preferences/foo.plist") and apply the `NSString` method `stringByExpandingTildeInPath`.

One More Thing: Message Forwarding

Although Objective-C does not provide true multiple-inheritance, it offers a workaround that lets objects respond to messages that are implemented in other classes. If you want your object to respond to another class's messages, you can add message forwarding to your applications and gain access to that object's methods.

Normally, sending an unrecognized message produces a runtime error, causing an application to crash. But before the crash happens, iOS's runtime system gives each object a second chance to handle a message. Catching that message lets you redirect it to an object that understands and can respond to that message.

Consider the `Car` example used throughout this chapter. The `carInfo` property introduced midway through these examples returns a string that describes the car's make, model, and year. Now imagine if a `Car` instance could respond to `NSString` messages by passing them to that property. Send `length` to a `Car` object and instead of crashing, the object would return the length of the `carInfo` string. Send `stringByAppendingString:` and the object adds that string to the property string. It would be as if the `Car` class inherited (or at least borrowed) the complete suite of string behavior.

Objective-C provides this functionality through a process called "message forwarding." When you send a message to an object that cannot handle that selector, the selector gets forwarded to a `forwardInvocation:` method. The object sent with this message (namely an `NSInvocation` instance) stores the original selector and arguments that were requested. You can override `forwardInvocation:` and send that message on to another object.

Implementing Message Forwarding

To add message forwarding to your program, you must override two methods: `methodSignatureForSelector:` and `forwardInvocation:`. The former creates a valid method signature for messages implemented by another class. The latter forwards the selector to an object that actually implements that message.

Building a Method Signature

This first method returns a method signature for the requested selector. For this example, a `Car` instance cannot properly create a signature for a selector implemented by another class (in this case, `NSString`). Adding a check for a malformed signature (that is, returning `nil`) gives this method the opportunity to iterate through each pseudo-inheritance and attempts to build a valid result. This example draws methods from just one other class via its `carInfo` instance variable:

From the Library of Eric Berna

```
- (NSMethodSignature*) methodSignatureForSelector:(SEL)selector
{
    // Check if car can handle the message
    NSMethodSignature* signature = [super
        methodSignatureForSelector:selector];

    // If not, can the car info string handle the message?
    if (!signature)
        signature = [carInfo methodSignatureForSelector:selector];

    return signature;
}
```

Forwarding

The second method you need to override is `forwardInvocation:`. This method only gets called when an object has been unable to handle a message. This method gives the object a second chance, allowing it to redirect that message. The method checks to see whether the `carInfo` string responds to the selector. If it does respond, it tells the invocation to invoke itself using that object as its receiver.

```
- (void)forwardInvocation:(NSInvocation *)invocation
{
    SEL selector = [invocation selector];

    if ([carInfo respondsToSelector:selector])
    {
        printf("[forwarding from %s to %s] ", [[[self class] description]
            UTF8String], [[NSString description] UTF8String]);
        [invocation invokeWithTarget:self.carInfo];
    }
}
```

Under ARC, you can call non-class messages such as `UTF8String` and `length` via `performSelector:`-style calls. ARC raises errors when you send undeclared methods to objects, even if forwarding allows objects to respond properly to those messages. You can work around this by declaring a class category that specifies the forwardable methods, allowing them to compile without warnings:

```
printf("Sending string methods to the myCar Instance:\n");


// These will compile with minor warnings about not casting the results
// of the methods. Ignore the warnings or cast to (char *) and (int)
// respectively
printf("UTF8String: %s\n",
    [myCar performSelector:@selector(UTF8String)]);
printf("String Length: %d\n",
    [myCar performSelector:@selector(length)]);
```

From the Library of Eric Berna

```
// This will not compile at all
// printf("String Length: %d\n", [myCar length]);

// This will compile cleanly and work at runtime
printf("String Length: %d\n", [(NSString *)myCar length]);
```

House Cleaning

Although invocation forwarding mimics multiple inheritance, `NSObject` never confuses the two. Methods such as `respondsToSelector:` and `isKindOfClass:` only look at the inheritance hierarchy and not at the forwarding change.

A couple of optional methods allow your class to better express its message compliance to other classes. Reimplementing `respondsToSelector:` and `isKindOfClass:` lets other classes query your class. In return, the class announces that it responds to all string methods (in addition to its own) and that it is a "kind of" string, further emphasizing the pseudo-multiple inheritance approach.

```
// Extend selector compliance
- (BOOL)respondsToSelector:(SEL)aSelector
{
    // Car class can handle the message
    if ( [super respondsToSelector:aSelector] )
        return YES;

    // CarInfo string can handle the message
    if ([carInfo respondsToSelector:aSelector])
        return YES;

    // Otherwise...
    return NO;
}

// Allow posing as class
- (BOOL)isKindOfClass:(Class)aClass
{
    // Check for Car
    if (aClass == [Car class]) return YES;
    if ([super isKindOfClass:aClass]) return YES;

    // Check for NSString
    if ([carInfo isKindOfClass:aClass]) return YES;

    return NO;
}
```

From the Library of Eric Berna

Super-easy Forwarding

The method signature/forward invocation pair of methods provides a robust and approved way to add forwarding to your classes. A simpler approach is also available on iOS 4.0 and later devices. You can replace both those methods with this single one, which does all the same work with less coding and less operational expense. According to Apple, this approach is "an order of magnitude faster than regular forwarding."

```
- (id)forwardingTargetForSelector:(SEL)sel
{
    if ([self.carInfo respondsToSelector:sel])
        return self.carInfo;
    return nil;
}
```

Summary

This chapter provided an abridged, high-octane introduction to Objective-C and Foundation. In it, you read about the way that Objective-C extends C and provides support for object-oriented programming. You were introduced to MRR and ARC memory management. You were subjected to a speedy review of the most important Foundation classes. So what can you take away from this chapter? Here are a few final thoughts:

- Try testing all the material discussed in this chapter directly in Xcode. Mess around with the examples. Hands-on experience offers the best way to gain critical skills you need for iOS development.
- Learning Objective-C and Cocoa takes more than just a chapter. If you're serious about learning iOS programming, and these concepts are new to you, consider seeking out single-topic books that are dedicated to introducing these technologies to developers new to the platform. Consider Aaron Hillegass's *Cocoa Programming for Mac OS X, 3rd Edition*, or Stephen Kochan's *Programming in Objective-C 2.0, 2nd Edition*, or Fritz Anderson's *Xcode 3 Unleashed*. Apple has an excellent Objective-C 2.0 overview at http://developer.apple.com/Mac/library/documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html.
- This chapter mentioned Core Foundation and Carbon but did not delve into these technologies in any depth. You can and will experience C-based APIs in the iOS SDK, particularly when you work with the address book, with Quartz 2D graphics, and with Core Audio, among other frameworks. Each of these specific topic areas is documented exhaustively at Apple's developer website, complete with sample code. A strong grounding in C (and sometimes C++) programming will help you work through the specific implementation details.

From the Library of Eric Berna

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

From the Library of Eric Berna

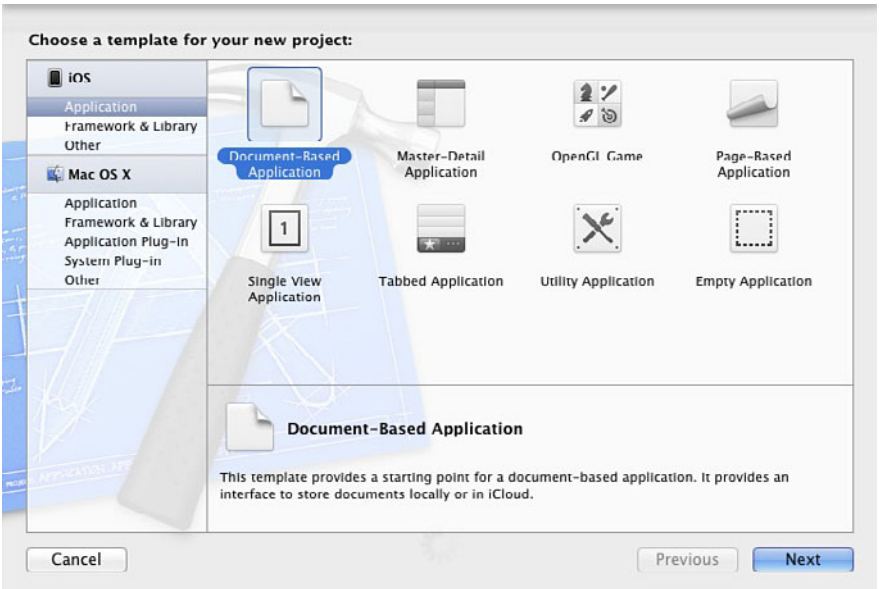


Figure 3-1 The Xcode New Project template selection window. The Xcode interface will change as Apple continues evolving its design tools.

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

From the Library of Eric Berna

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

From the Library of Eric Berna

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.

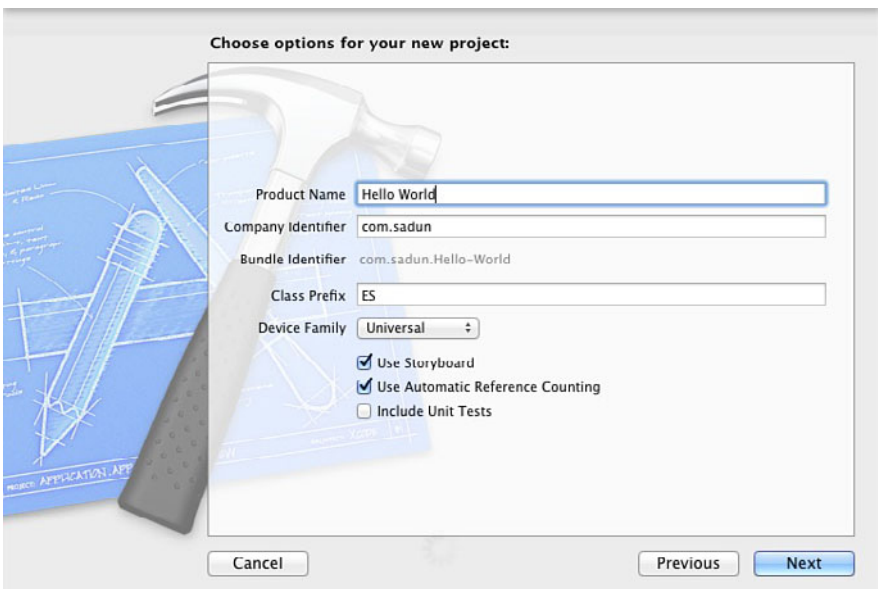


Figure 3-2 Once you set your company identifier, Xcode applies your settings between successive runs. This helps you avoid what was one of the most irritating features of previous Xcode releases—having to remember to edit your settings from "com.yourcompany" for each project.

From the Library of Eric Berna

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.

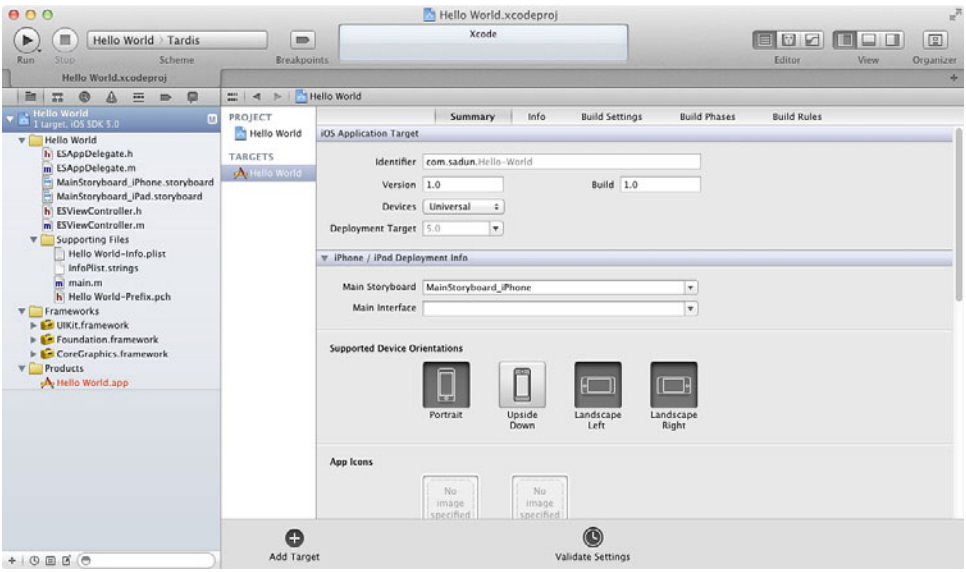


Figure 3-3 This brand-new Hello World project was created by choosing one of the available templates. On older projects, you may see a "modernize" button on this screen as well.

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,

From the Library of Eric Berna

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.

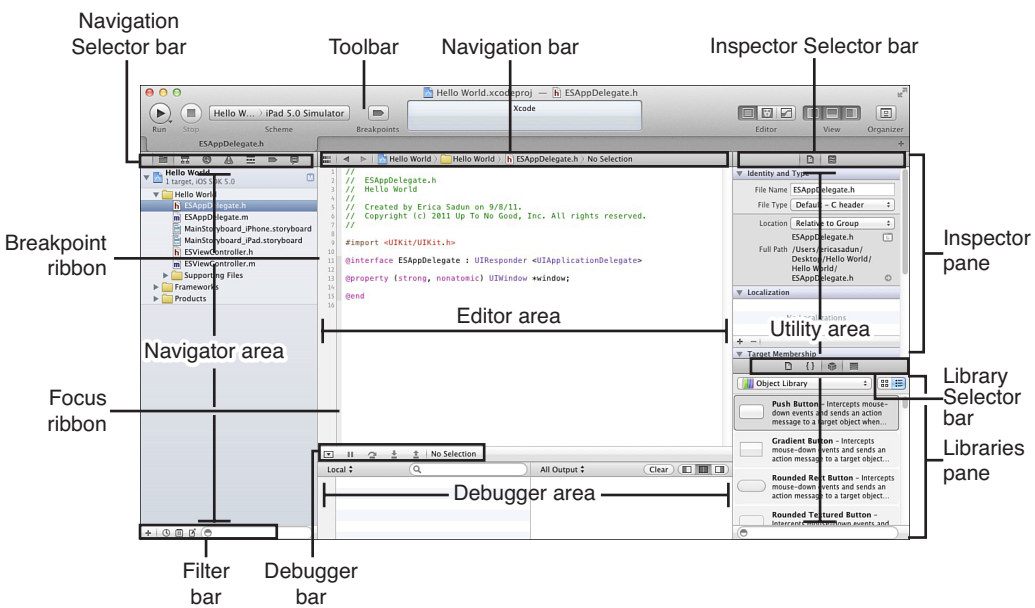


Figure 3-4 The Xcode workspace consists of several areas, as shown here.

From the Library of Eric Berna

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.

From the Library of Eric Berna

> **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

From the Library of Eric Berna

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.

From the Library of Eric Berna

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.

From the Library of Eric Berna

> **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).

From the Library of Eric Berna

- **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.

From the Library of Eric Berna

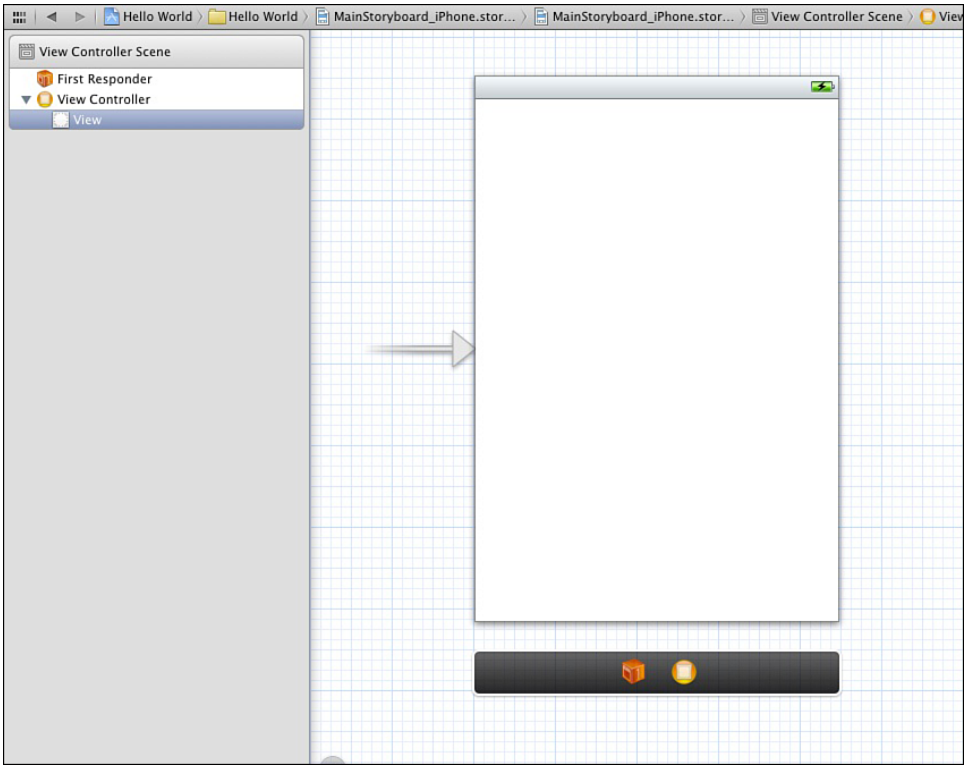


Figure 3-5 The Interface Builder window for an iPhone storyboard. A small dock floats below each storyboard scene, offering access to objects associated with its view.

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.

From the Library of Eric Berna

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.)

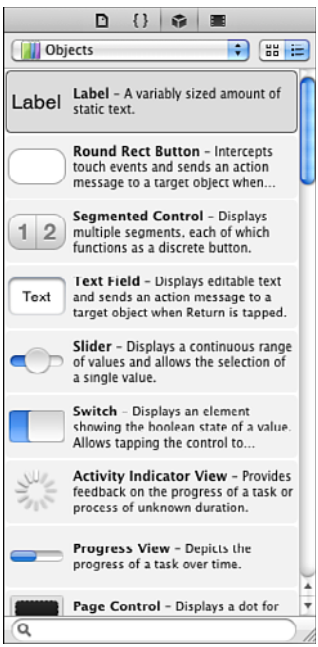


Figure 3-6 The Interface Builder object library.

From the Library of Eric Berna

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.

From the Library of Eric Berna

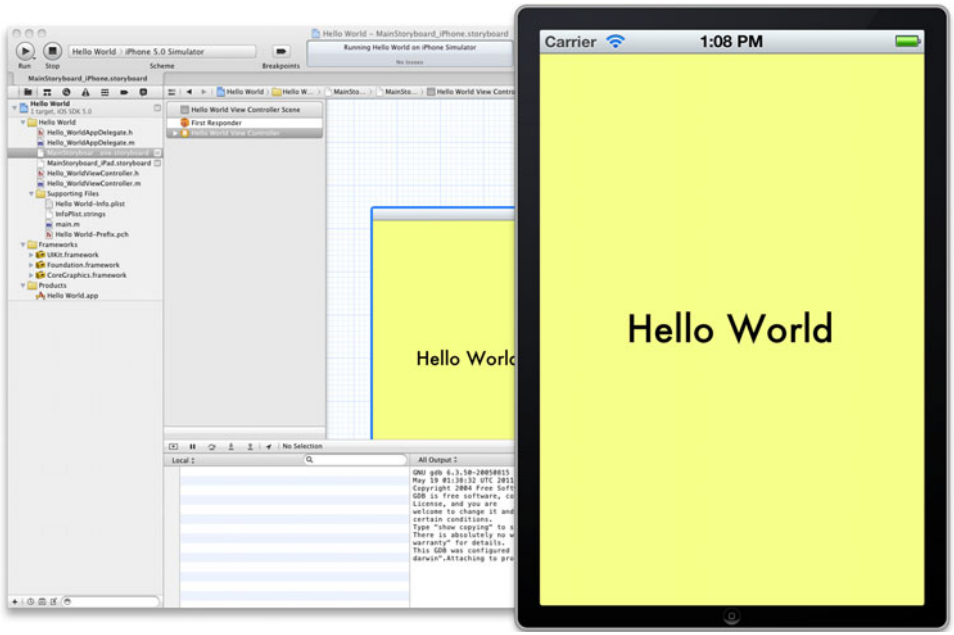


Figure 3-7 The customized Hello World application runs on the simulator.

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.

From the Library of Eric Berna

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).

From the Library of Eric Berna

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.

From the Library of Eric Berna

- **/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

From the Library of Eric Berna

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 walkthrough 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

From the Library of Eric Berna

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
{
```

From the Library of Eric Berna

```
    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.

From the Library of Eric Berna

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.

From the Library of Eric Berna

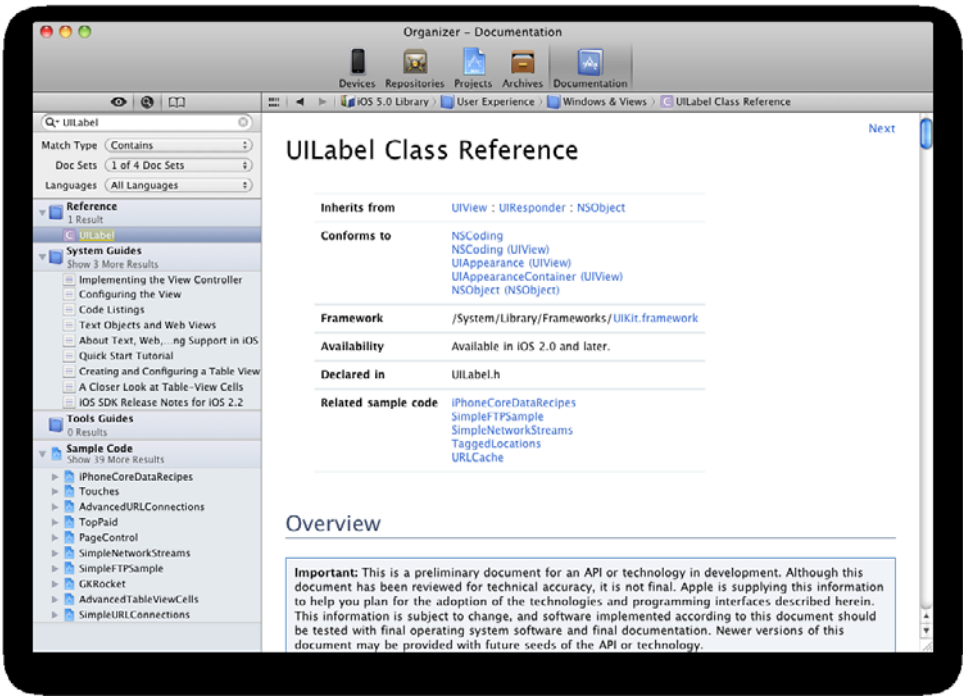


Figure 3-8 Apple offers complete developer documentation from within Xcode itself.

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.

From the Library of Eric Berna

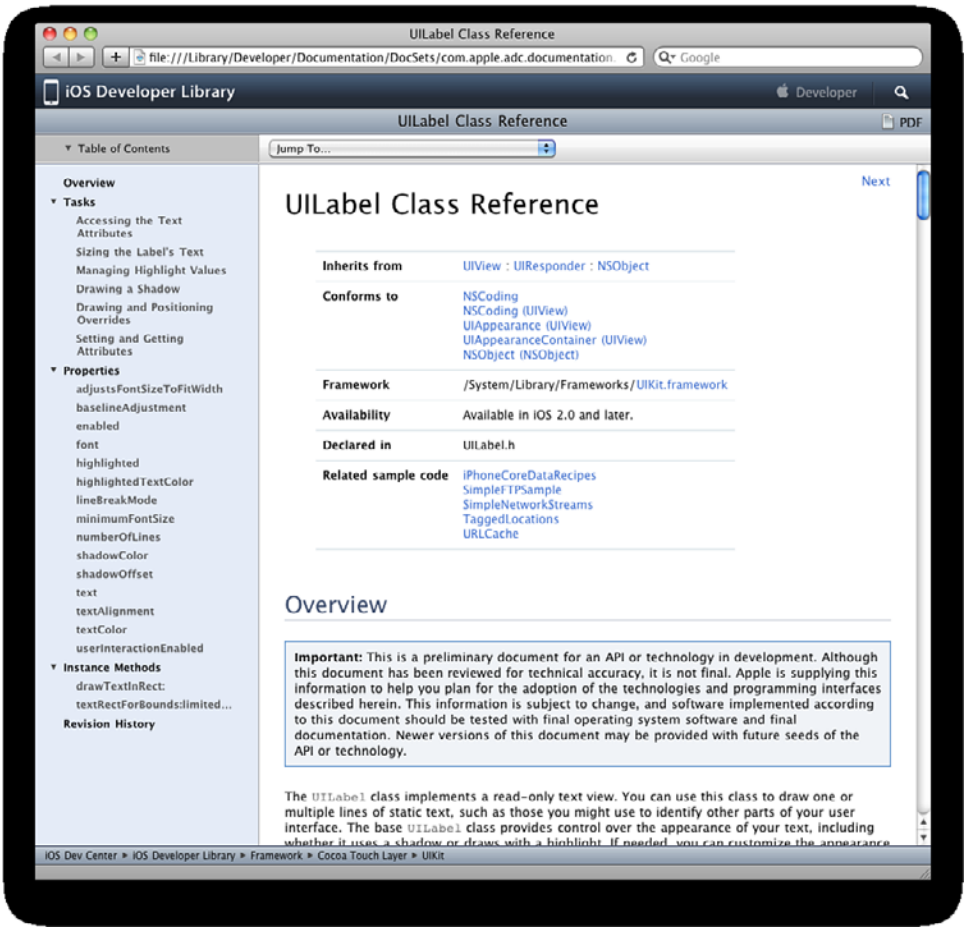


Figure 3-9 The "Table of Contents" view is no longer available from within Xcode itself but can be accessed via Open Page in Browser.

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.

From the Library of Eric Berna

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;
```

From the Library of Eric Berna

```
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.

From the Library of Eric Berna

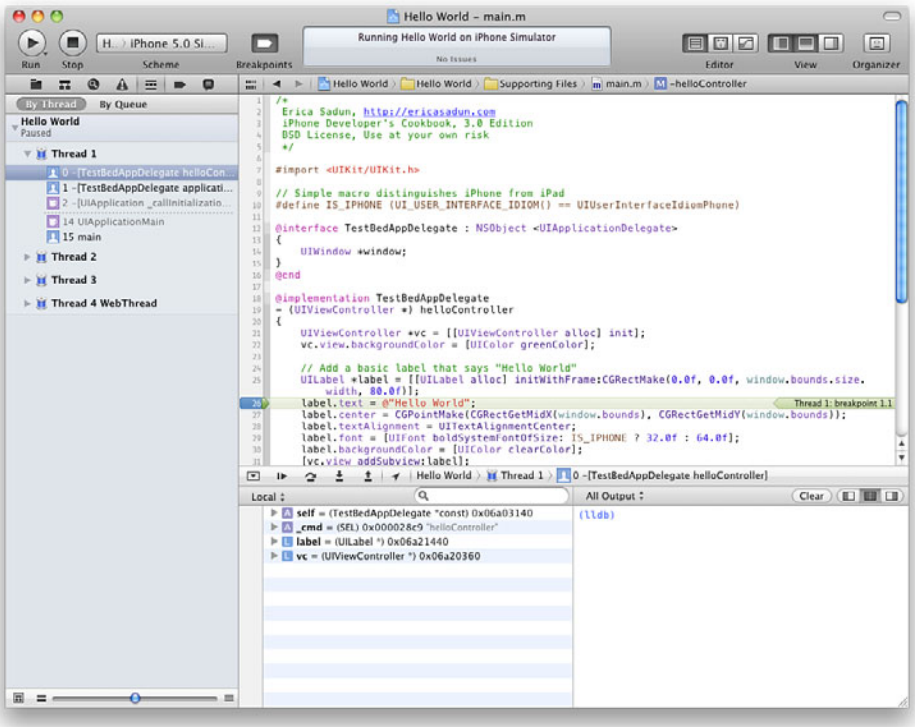


Figure 3-10 Blue breakpoint indicators appear in the gutter to the left of the editor area. You can reveal the debugger at the bottom of the workspace by clicking the Debugger disclosure button while running an application or by clicking the center of the three View buttons at the top-right of the workspace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

From the Library of Eric Berna

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.

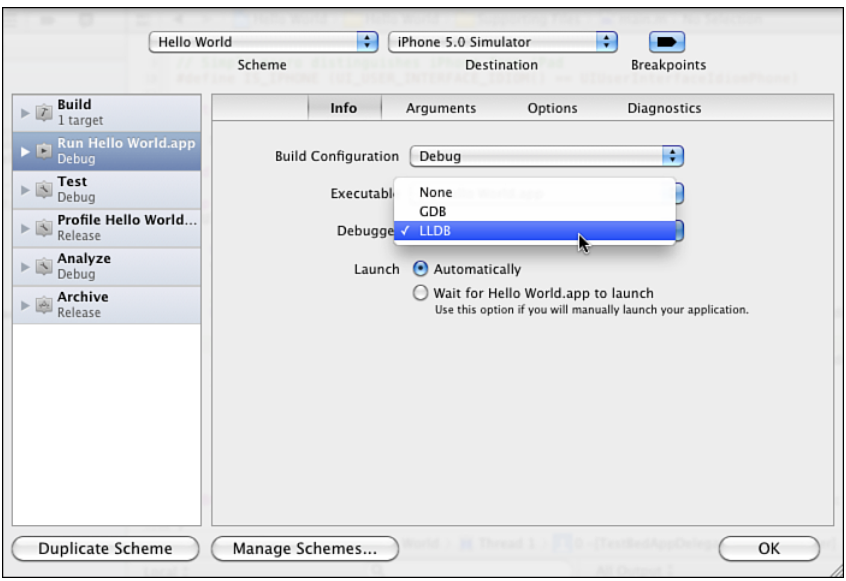


Figure 3-11 The project scheme editor allows you to select which debugger you prefer to use.

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

From the Library of Eric Berna

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.

From the Library of Eric Berna

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.

From the Library of Eric Berna

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:

From the Library of Eric Berna

- 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

From the Library of Eric Berna

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.

From the Library of Eric Berna

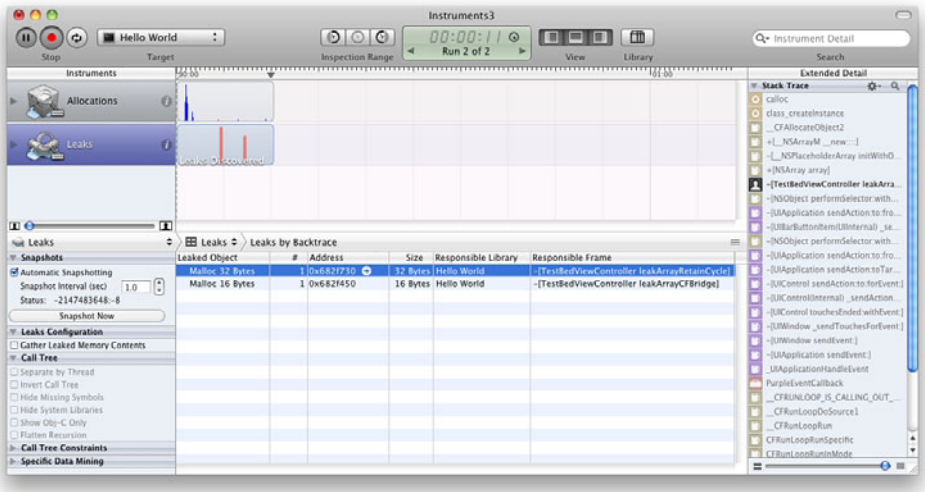


Figure 3-13 Instruments tracks leaks created by memory blocks that cannot be addressed or recovered by your code.

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];
}
```

From the Library of Eric Berna

```
- (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.

From the Library of Eric Berna

```
- (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.

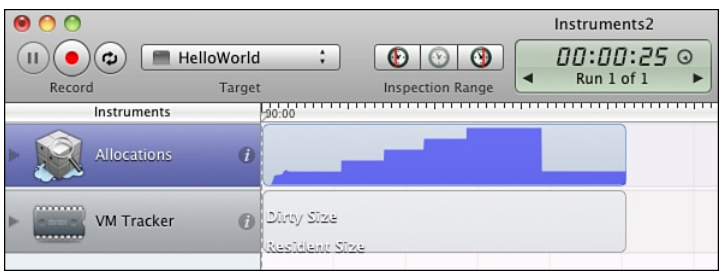


Figure 3-14 Instruments helps monitor object allocations, letting you test your release strategies during memory warnings.

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.

From the Library of Eric Berna

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
```

From the Library of Eric Berna

```
{
    [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.

From the Library of Eric Berna

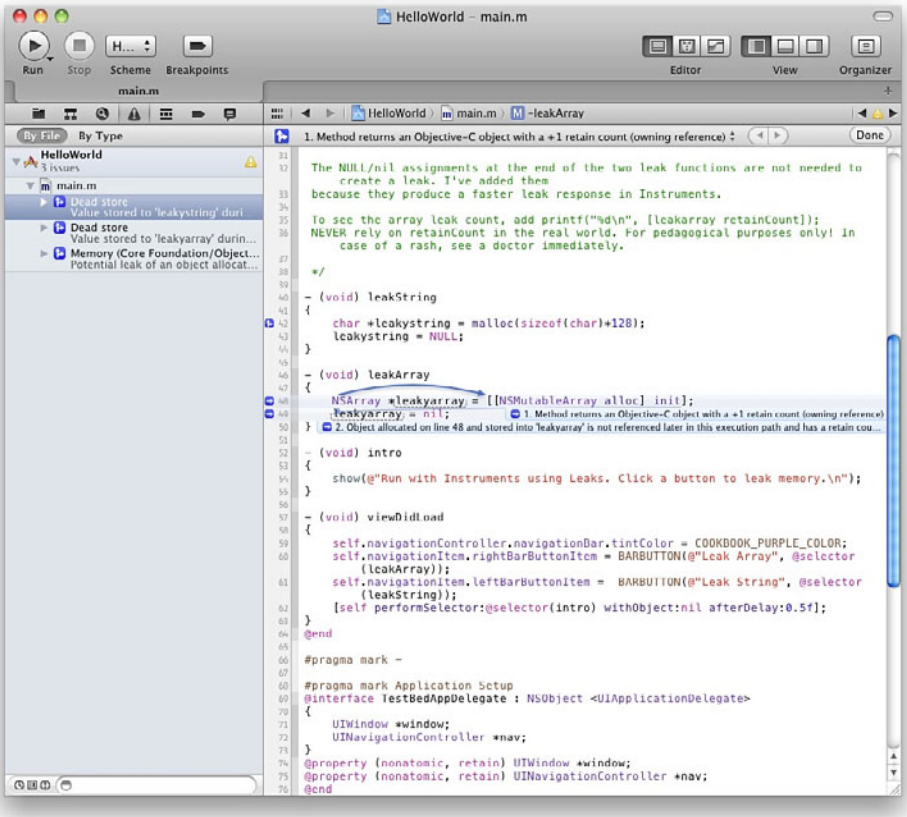


Figure 3-15 The Clang static analyzer creates bug reports for source code and embeds them into your Xcode editor window. Clang is useful for both MRR (as shown here) and ARC and provides reports specific to each compilation style.

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.

From the Library of Eric Berna

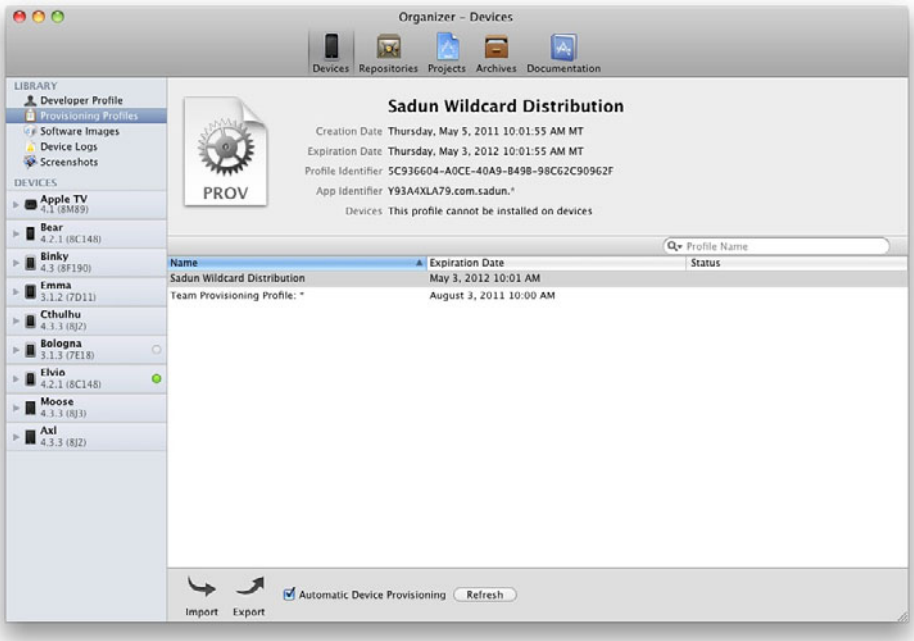


Figure 3-16 Xcode's Organizer window provides a central hub for managing your devices, certificates, and provisions.

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.

From the Library of Eric Berna

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.

From the Library of Eric Berna

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.

From the Library of Eric Berna

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.

From the Library of Eric Berna

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).

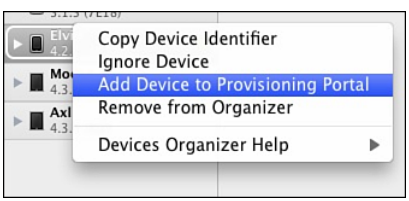


Figure 3-17 Use the Device organizer to add devices to the iOS provisioning portal.

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.

From the Library of Eric Berna

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.

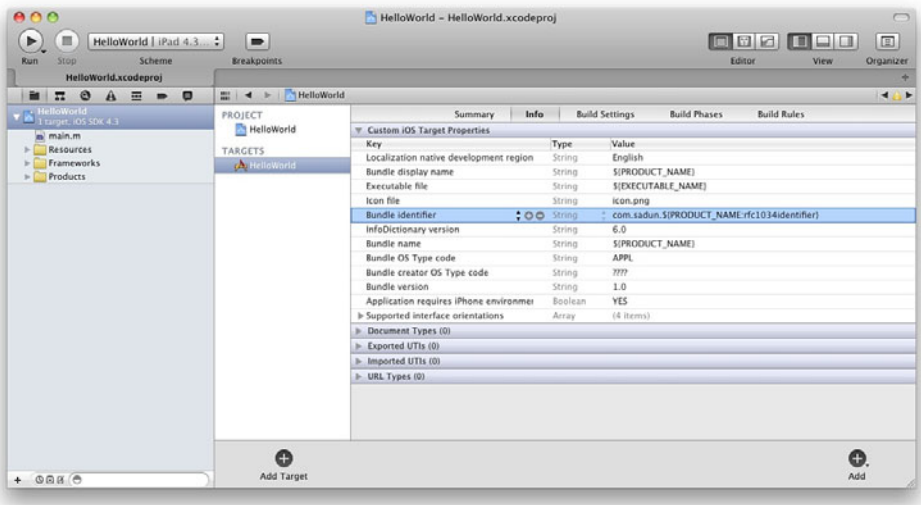


Figure 3-18 The Info tab allows you to edit the Bundle identifier.

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.

From the Library of Eric Berna

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).

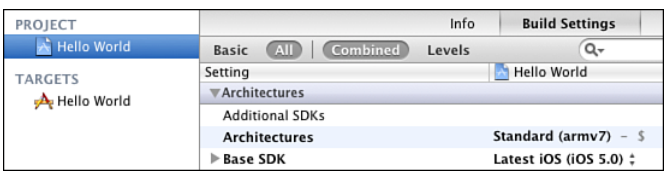


Figure 3-19 The Base SDK sets the iOS version used to compile your applications. The armv7 architecture is currently available on the iPhone 3GS and newer, the iPod touch 3G and newer, and all iPads.

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.

From the Library of Eric Berna

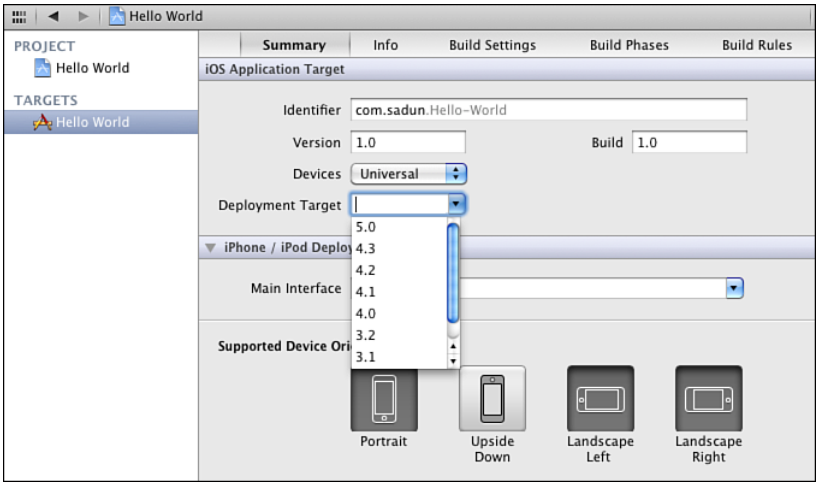


Figure 3-20 Deployment Target sets the earliest iOS version that is permitted to run your application.

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.

From the Library of Eric Berna

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

From the Library of Eric Berna

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")) . . .
  ```

From the Library of Eric Berna

- **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:`).

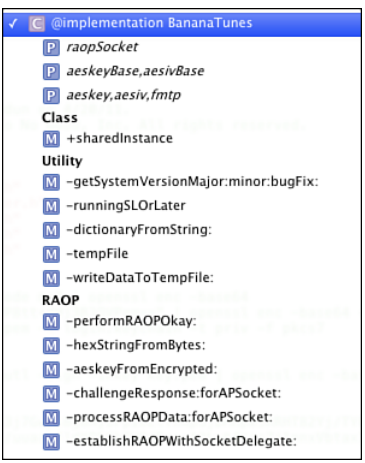


Figure 3-22 Use pragma marks to organize your method and function list.

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
```

From the Library of Eric Berna

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.

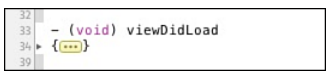


Figure 3-23 Xcode lets you collapse individual methods and functions. This allows you to see parts of your program that normally would not fit onscreen together.

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

From the Library of Eric Berna

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.

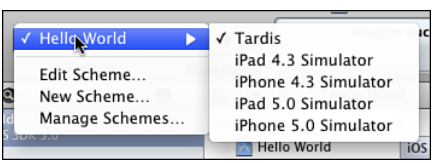


Figure 3-24 The scheme pop-up menu consists of schemes and their run destinations, followed by management options.

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.

From the Library of Eric Berna

> **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

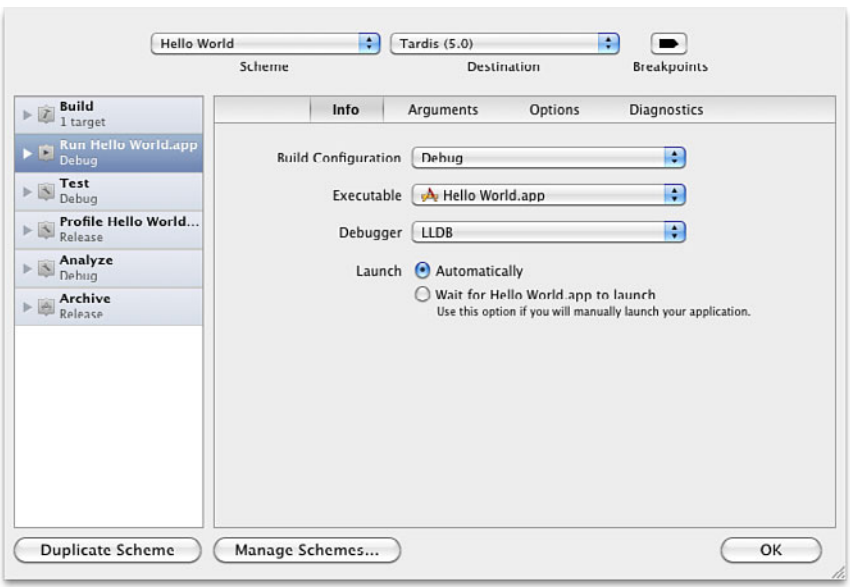


Figure 3-25 The actions list for a scheme offer a set of tasks that may need to be performed during the creation of an application.

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.

From the Library of Eric Berna

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.

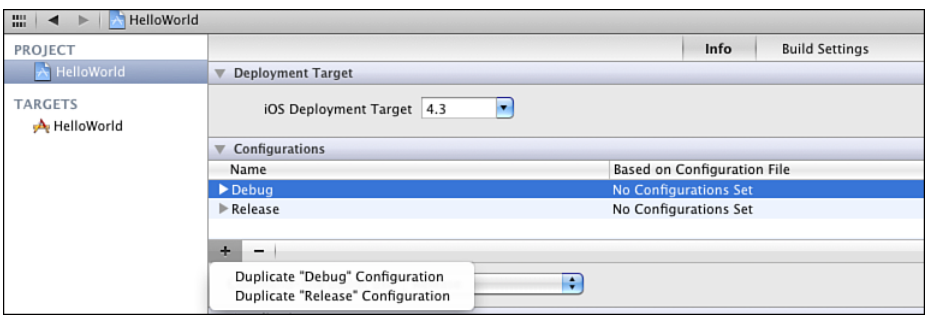


Figure 3-26 Use the Info > Configurations pane to create new configurations so you can build with preset options such as signing identities.

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.

From the Library of Eric Berna

> **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.

From the Library of Eric Berna

> **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.

From the Library of Eric Berna

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>
```

From the Library of Eric Berna

```
            <!-- 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>
```

From the Library of Eric Berna

```
                <!-- 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

From the Library of Eric Berna

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 reinstalling 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.

From the Library of Eric Berna

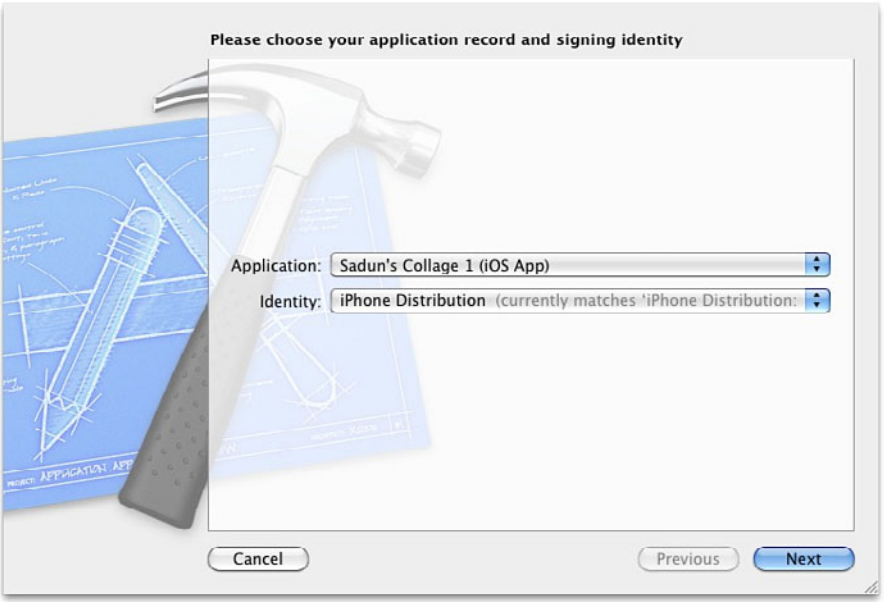


Figure 3-28 Only apps that are "Waiting for Upload" are listed in the Organizer's submission screen. Make sure your identity is not set to an ad hoc profile, also listed as iPhone Distribution. It's a common error that lots of developers make.

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.

From the Library of Eric Berna

- 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.

From the Library of Eric Berna

This page intentionally left blank

From the Library of Eric Berna

4

Designing Interfaces

The iOS SDK helps you craft user interfaces in a variety of ways. This chapter introduces the visual classes you'll work with and discusses their roles in the interface design process. You read about controllers, how they work with visual classes, and how they handle tasks such as device reorientation. Then you move on to solutions for laying out and customizing interfaces. You learn about hybrid solutions that rely both on Interface Builder–created interfaces and Objective-C-centered ones. By the time you finish this chapter, you'll have discovered many approaches that you can apply to your own application design.

UIView and UIWindow

Nearly everything that appears on an iOS device's screen represents a child of the `UIView` class. Views act like little canvases that you can draw on with colors, pictures, and buttons. You can move them around the screen. You can resize them. You can layer them. Views provide the basic component of user interfaces.

The typical iOS design pattern goes like this: one window, many views. If you keep that idea in mind, the iOS interface design scenario simplifies. Metaphorically speaking, `UIWindow` is the TV set, and `UIViews` are the actors on your favorite show. They can move around the screen, appear, and disappear, and may change the way they look and behave over time.

The TV set, on the other hand, normally stays still. It has a set screen size that doesn't change even if the virtual world you see through it is practically unlimited. You may even own several TVs in the same household (just like you can create several `UIWindow` instances in the same application), but you can watch just one at a time. With newer versions of the SDK, you can also create windows for external screens. External screens are accessed through a Video Out cable attached to an iOS device.

> **Note**
>
> AirPlay, an over-the-air technology that streams media from your device to Apple TV, speaker docks, and other enabled receivers, allows you to create and write to windows wirelessly.

From the Library of Eric Berna

UIViews are user interface building blocks. They provide visual elements that are shown onscreen and invite user interaction. Every iOS user interface is built from UIViews displayed within a parent UIWindow, which is a specialized kind of UIView. The window acts as a container; it is the root of a display hierarchy. It holds a collection of visible application components within itself.

Beyond UIView and UIWindow, you find a wealth of specialized views, such as UIImageView and UITextView, that allow you to build your interfaces from predesigned components. This section provides a rundown of those views. The Interface Builder library makes these views available to you, allowing you to place them in your application interfaces to build your GUIs.

> **Note**
> The "UI" at the beginning of certain classes (such as UIView) stands for User Interface.

Views That Display Data

One of the most important things a view can do is provide a visual representation of data. In Cocoa Touch, the following classes show information onscreen:

- The UITextView class presents passages of text to your users and/or allows them to type in their own text using the keyboard. You choose whether to set the view text as editable. Text views use a single font with a single text size throughout. Text views are explored in great detail in Chapter 10, "Working with Text."
- UILabel instances present short, read-only text views. As the name implies, this class is used to statically label items on your screen. You choose the color, font size, and font face for your labels by setting view properties. The words "Fahrenheit" and "Celsius," shown in figures later in the chapter, are created by UILabels.
- UIImageViews show pictures. You load them with UIImage objects, which are instances of an abstract image storing class. Once these are loaded, you specify the view's location and size. The UIImageView automatically scales its contents to fit those bounds. A special feature of this class allows you to load a sequence of images rather than a single picture and animate them on demand.
- When you want to display HTML, PDFs, or other advanced web content, the UIWebView class provides all the functionality you need. UIWebView instances offer a powerhouse of display capabilities, allowing you to present nearly any data type supported by the built-in Safari browser. These views offer simple web browsing with a built-in history, essentially giving you a canned, usable Safari-style object you can insert into your programs. Sometimes developers use UIWebView instances to present blocks of stylized text. As a bonus, these support zoom and scroll with no additional work.
- MKMapViews (MK stands for Map Kit) embed maps into your applications. Users can view map information and interact with the map contents, much as they would with the Maps application. This class, which was introduced in the 3.0 SDK, lets you

From the Library of Eric Berna

annotate the map with custom information using the `MKAnnotationView` and `MKPinAnnotationView` classes.

- `UIScrollView` instances allow you to present content that is larger than the normal size of an application window. Users can scroll through that content to view it all, using horizontal and/or vertical scrolling. Scroll views support zooming, so you can use standard iOS pinch and spread gestures to resize content.

Views for Making Choices

iOS offers two core classes that offer choices to users. The `UIAlertView` class produces those blue pop-up windows you see in many iPhone and iPod touch applications, and the more restrained popover presentations used on iPads. You choose the message and customize their buttons to ask users questions. For example, you might ask a user to confirm or cancel an action in your program. In addition to questions, you can present information. When just one button is offered (typically "Okay"), alert views provide a simple way to show text to users.

The second choice-based class is `UIActionSheet`, which offers menus that scroll up from the bottom of the screen on iPhones or pop over from menu bars on the iPad. Action sheets display a message and present buttons for the user to choose from. Although these sheets look different from alert views, functionally they perform in a similar manner and historically they used to be a lot closer in terms of class inheritance than they are now. As a rule, use action sheets when you have a number of options to choose from and use alert views when you are presenting just two or three choices at most.

Both these presentations are modal on iPhones and iPod touches, and partially modal on the iPad. They require users to tap a button before proceeding in the former case. On the iPad, users can also cancel out of a presentation by touching anywhere onscreen outside the popover.

For this reason, it's polite to offer a cancel option for any iPhone- or iPod touch-based alert or action sheet requiring a choice. (You can omit a "cancel" for an alert view that simply presents text; an Okay button suffices for dismissing the view.) On the iPad, users can cancel out of an action sheet by tapping outside the sheet's bounds; for that reason, iPad action sheets do not display the cancel button you specified when creating the sheet.

Controls

Controls are onscreen objects that transform user touches into callback triggers. They may also provide numeric or text values consumed by your application. Controls include buttons, switches, and sliders, among others. They correspond closely to the same kinds of control classes used in desktop programming, and are discussed at length in Chapter 9, "Building and Using Controls." Here's a quick rundown of the major classes provided by Cocoa Touch and what each control offers:

- `UIButton` instances provide onscreen buttons. Users can press them to trigger a callback via target/action programming. You specify how the button looks, the text it displays, and how the button triggers. The most typical trigger used is "touch up

From the Library of Eric Berna

inside," where the user's touch ends inside the button's bounds. If it seems strange to trigger with touch up rather than touch down, consider that the de facto standard on iOS allows users to cancel a button press by sliding their finger away from the button before lifting it.

In Interface Builder, buttons are called Round Rect Buttons. In the IB editor, you also encounter buttons that look like views and act like views but are not, in fact, views. Bar button items (`UIBarButtonItem`) store the properties of toolbar and navigation bar buttons but are not buttons themselves. The bars use these descriptions to build themselves; the actual button views are not generally accessible to you as a developer.

> **Note**
>
> In Interface Builder, you can search the view library by class name (for example, `UIButton`) or by IB's description (for example, `round` or `button`).

- The `UISegmentedControl` offers a row of equally sized buttons that act like the old-fashioned radio buttons in a car—namely, only one button can be selected at a time. You can present these buttons as images or text. An option (called "momentary") lets you replace the radio button behavior with a style that prevents the buttons from showing which button was last selected. A related class, the `UIStepper`, introduces a small bar with two segments: + and –, allowing users to raise or lower the control's value.
- In Cocoa Touch, the `UISwitch` class provides a simple binary control. This class presents on/off choices and looks a little like a standard light switch you'd see on a wall. The switch was redesigned in iOS 5 for a more compact presentation and a smaller round toggle area.
- The `UISlider` class lets users choose a value from a specified range by sliding an indicator along a horizontal bar. The indicator (called the "thumb") represents the current setting for the control. The value is set by the thumb's relative placement. iOS's onscreen volume slider in the iPod/Music application represents a typical slider instance. Starting in iOS 5, you can assign a minimum and maximum color tint to the slider to create a gradient effect.
- Page controls let users move between pages, usually as part of a `UIScrollView` implementation. The `UIPageControl` class offers a series of small dots (like the ones you see on iOS's home page) showing the current page and letting users navigate to the next or previous page.
- `UITextField`s are a kind of control that let you enter text. These fields offer just a single line for input and are meant to solicit short text items (such as usernames and passwords) from users.

From the Library of Eric Berna

Tables and Pickers

Tables present a scrolling list of choices. The `UITableView` class provides the most commonly used table style, which you see, for example, in the Contacts, YouTube, and iPod/Music applications. Tables offer rows of information (provided by the `UITableViewCell` class), which users scroll through and can select.

The `UIPickerView` class offers a kind of table, where users can select choices by scrolling individual wheels. A specialized version of this class is the `UIDatePicker`, which comes preloaded with date- and time-specific behavior and is used extensively in the Calendar and Clock applications.

Bars

The iOS offers four kinds of bar-style views. Bars are compact views (typically shorter than 50 points in height) that extend from one side of the screen to the other. Bar usage is device dependent. The structured view hierarchies common on iPhone and iPod touch do not play as important a role on the iPad.

The most commonly used bar view for iPhone and iPod touch is the `UINavigationBar`, which is presented on top of many interfaces to provide navigation state. As a developer, you almost never work directly with instances of these views. Instead, the view is generated and managed by `UINavigationController` instances, which you read about in chapter sections that follow this one. Navigation controllers help you collapse large interfaces into a small window presentation. Navigation bars can be used on the iPad as well but they are more typically used to present menu choices rather than to shrink interfaces.

Tab bars offer the kinds of choices you see at the bottom of the YouTube and iPod/Music applications, such as Featured, Most Viewed, Albums, and Podcasts. Like navigation bars, they are used to shrink an application's interface into manageable pieces and are used primarily on iPhone and iPod touch rather than on the iPad. Search bars (`UISearchBar`) add a text-based view meant to be shown on the top navigation bar of a table, as used in iOS Contacts application. As with navigation bars, you normally work through `UITabBarControllers` and `UISearchDisplayControllers` instead of building and managing the view directly.

Of all the iOS bars, only the `UIToolbar` class is meant for direct use. It provides a series of buttons similar to segmented controls but with a different look. Toolbars are limited to a momentary highlighting style. The role of toolbars is to provide a vocabulary of actions that act on the current view. The toolbar used in the Mail application allows you to delete messages or to reply to messages. Toolbars present monochrome images on their default buttons.

Although iPhone and iPod touch toolbars usually appear at the bottom of the screen, iPad toolbars should be placed toward the top of the interface. The iPad's greater screen real estate allows you to group all your toolbar buttons together in one bar interface element. This allows you to design past common iOS patterns such as undo/redo buttons at

From the Library of Eric Berna

the top of the screen and action choices at the bottom to bring all user manipulation into a single UI element.

If your design ideas include tab bars and toolbars, take the time to read Apple's Human Interface Guidelines, available as part of the standard iOS documentation library. Apple regularly rejects applications that use bars in a manner inconsistent with these guidelines.

> **Note**
>
> As with bar button items, navigation items appear in Interface Builder and can be placed in your projects as you would place views. Like their cousins, navigation items are not views themselves. They store information about what items go on navigation bars and are used to build the bar that does appear.

Progress and Activity

Cocoa Touch provides two classes meant to communicate an ongoing activity to the user. The `UIActivityIndicatorView` offers a spinning-style wheel, which is shown during an ongoing task. The wheel tells the user that the task may finish at some point, but it does not determine when that time will end. When you want to communicate progress to a user, use the `UIProgressView` class. Instances offer a bar that fills from left to right, indicating how far a task has progressed.

View Controllers

On iOS, view controllers centralize certain kinds of view management. They provide practical utility by linking views into the pragmatic reality of your device. View controllers, which are discussed in depth in Chapter 5, "Working with View Controllers," handle reorientation events such as when users tip the iOS device on its side to landscape mode, navigation issues such as when users need to move their attention from view to view, and memory issues about responding to low-memory warnings.

View controllers aren't views. They are abstract classes with no visual representation; only views offer visual canvases. Instead, they help your views live in a larger application design environment. Do not set a frame the way you would with a normal `UIView`. `UIViews` use `initWithFrame:`; `UIViewControllers` use `init`.

The iOS SDK offers many view controller classes. These classes range from the general to the specific. In a way, specialized controllers are both a blessing and a curse. On the positive side, they introduce enormous functionality, essentially with no additional programming burden. On the downside, they're so specialized that they often hide core features that developers might prefer to work with.

Apple is, however, responsive to feature requests from developers. In earlier releases of the SDK, developers could not directly access camera features and were forced to use the image picker controller class. In more modern SDK updates, developers are now able to directly pull camera data and use it in their applications. Chapter 7, "Working with Images," demonstrates how.

From the Library of Eric Berna

Here's a quick guide to a subset of the view controllers you'll encounter while building your iOS application interfaces.

UIViewController

`UIViewController` is the parent class for view controllers and the one you use to manage your primary views. It's the workhorse of view controllers. You may spend a large part of your time customizing this one class. The basic `UIViewController` class manages each primary view's lifetime from start to finish and takes into account the changes that the view must react to along the way.

For example, `UIViewControllers` handle reorientation tasks, letting you program for both landscape and portrait orientation. `UIViewControllers` decide whether to change their orientation when a user tilts the device, and specify how that orientation change occurs. They do this via instance methods such as `shouldAutorotateToInterfaceOrientation:`. Without a view controller, your interface won't support automatic orientation updates. Many developers have found it difficult trying to rotate `UIViews` directly without the help of a view controller class.

`UIViewController` instances are responsible for setting up how a view looks and what subviews it displays. Often they rely on loading that information from .xib files. A variety of instance methods such as `loadView` and `viewDidLoad` let you add behavior while or after a view sets up.

Reacting to views being displayed or dismissed is another job that view controllers handle. These are the realities of belonging to a larger application. Methods such as `viewDidAppear:` and `viewWillDisappear:` let you finish any bookkeeping associated with your view management. You might preload data in anticipation of being presented or clean up memory that won't be used when the view is not onscreen via `viewDidUnload:`.

Each of the tasks mentioned here specifies how a view fits into an enveloping application and works on a particular device. The `UIViewController` mediates between views and these external demands, allowing the view to change itself to meet these needs.

UINavigationController

As the name suggests, navigation controllers allow you to drill up and down through tree-based view hierarchies, which is an important primary interface design strategy on smaller members of the iOS device family and a vital supporting one on the iPad. Navigation controllers create the solid-colored navigation bars that appear at the top of many standard iOS applications. You see navigation controllers in use whenever you drill through some sort of hierarchy, whether using the Contacts application or the on-iOS App Store. Both of these applications are built using navigation controllers.

On the iPhone and iPod touch, navigation controllers let you push new views into place and automatically generate "back" buttons showing the title of the calling view controller. All navigation controllers use a "root" view controller to establish the top of their navigation tree, letting those back buttons lead you back to a primary view. Navigation controllers and their trees are discussed in greater detail later in this chapter. On the iPad,

From the Library of Eric Berna

you can use a navigation controller-based interface to work with bar-button-based menu items, to present popover presentations, or to integrate with `UISplitViewController` instances for a master-detail presentation experience. Split view controllers are described later in this section.

Handing off responsibility to a navigation controller lets you focus your design work on creating individual view controller screens. You don't have to worry about specific navigation details other than telling the navigation controller which view to move to next. The history stack and the navigation buttons are handled for you. Chapter 5 discusses navigation controllers in further detail and offers recipes for their use.

`UITabBarController`

Another small-device solution, `UITabBarController`, lets you control parallel views in your iPhone or iPod touch application. (Tab bar controllers are not recommended for iPad applications, except for use in occasional popover elements or as part of a split view pane.) These parallel views are like stations on a radio. A tab bar helps users select which `UIViewController` to "tune in to," without there being a specific navigation hierarchy. You see this best in applications such as YouTube and Music, where users choose whether to see a "Top 25" list or decide between viewing albums or playlists. Each parallel world operates independently, and each can have its own navigation hierarchy. You build the view controller or navigation controller that inhabits each tab, and Cocoa Touch handles the multiple-view details.

For example, when tab bar instances offer more than five view controller choices at a time, users can customize them through the More > Edit screen. The More > Edit screen lets users drag their favorite controllers down to the button bar at the bottom of the screen. No extra programming is involved. You gain editable tabs for free. All you have to do is request them via the `customizableViewControllers` property. See Chapter 5 to read more about implementing tab-bar-based applications and setting the images that adorn each button.

Split View Controllers

Meant for use on iPad applications, the `UISplitViewController` class offers a way to encapsulate a persistent set of data (typically a table) and associate that data with a detail presentation. You can see split views in action in the iPad's mail application. When used in landscape orientation, a list of messages appears on the left; individual message content appears on the right. The detail view (the message content in Mail) on the right is subordinate to the master view (Mail's message list) on the left. Tapping a message updates the right-hand view with its contents.

In portrait orientation, the master view is hidden. It is accessed by a popover, which is reached by tapping the left button of the split view's top bar.

From the Library of Eric Berna

Page View Controller

Page view controllers are containers for other view controllers. They create a book-like presentation where you set the book's "spine," typically along the left or top of the view. Build your "book" by adding individual content controllers. Each "page" transitions to the next using a style that's theoretically customizable but is limited to a page curl in the iOS 5.0 Beta SDK.

The `UIPageViewController` forms the core for one of Xcode's new project styles, the page-based application, which you see as an option whenever you create a new project.

Popover Controllers

Specific to the iPad, popover controllers create transient views that pop over other existing interface content. These controllers present information without taking over the entire screen, the way that modal views normally do. The popovers are usually invoked by tapping a bar button item in the interface (although they can be created using other interaction techniques) and are dismissed either by interacting with the content they present or by tapping outside their main view.

Popovers are populated with `UIViewController` instances. (The iPad's implementation of action sheets also uses popovers, but this approach is not exposed via iOS SDK.) Build the view controller and assign it as the popover's `contentViewController` property, before presenting the popover. This allows popovers to present any range of material that you can design into a standard view controller, offering exceptional programming flexibility.

Table Controllers

Table view controllers simplify using tables in your iOS projects. The `UITableViewController` class provides a standard already-connected `UITableView` instance and automatically sets delegation and data sources to point to itself. All you have to do is supply those delegate and data source methods to fill up the table with data and react to user taps. `UITableViewController` is discussed at length in Chapter 11, "Creating and Managing Table Views."

The search display controller is a kind of table view but one that offers a built-in search bar via `UISearchBar`. With it, you allow users to search data that is provided by another view controller, called its "contents controller." As users update the search information, the contents controller adjusts its data source to include only those items that match the search query.

It may seem odd to force another controller to perform that work, but in practice, it works out very neatly. The contents controller is almost always a table view controller, which displays the search controller on demand. The search then weeds through the original table's data and shows a subset of that information until the search is dismissed.

The `NSFetchedResultsController` also provides a kind of table-based controller. Although strictly speaking, not a view controller, this class helps populate a `UITableView` with objects fetched from a Core Data store. See Chapter 12, "A Taste of Core Data," for an example that shows this class in action.

From the Library of Eric Berna

Address Book Controllers

The Address Book user interface framework (`AddressBookUI.framework`) provides several view controllers that let you select a person from your address book, view his or her details, and add a new person or modify an existing person's entry. These view controllers tie into the C-based ABAddressBook framework, which provides functions that query and update iOS's built-in address book.

Image Picker

This utility controller allows users to select images from onboard albums or to snap a photo or shoot video using iOS camera. With it, you gain full access to most of the organizational features made available to users via the Camera and Photos applications. In truth, there are not two separate applications. There is just one application that poses as those two utilities, just as the single controller offers access to both camera and photo selection features.

For selecting pictures, Apple has added an advanced image-selection interface. Users can navigate up and down the photo album hierarchy until they find the image they want to use. The picker automatically handles access to the onboard photo album, leaving you little more to do than decide how to use the picture it picks.

The photo/video interface is equally impressive. The controller even lets the users optionally orient and zoom an image before finishing, providing user-defined "edits" on the picture they snap. Full discussions of this class, including how-to's for both the selection and camera versions, appear in Chapter 7.

Mail Composition

The `MFMailComposeViewController` lets you create mail messages that users can customize from directly in your program. Although iOS has long supported `mailto:` URLs to send mail messages, this class offers far more control over mail contents and attachments. What's more, users can continue working within your program without being forced to leave to access the Mail application.

The mail composition controller is simple to use and is used in Chapter 7 to mail photographs and in Chapter 10 to send text and data. It is part of the MessageUI framework; the MF prefix stands for Message Framework.

Document Interaction Controller

The `UIDocumentInteractionController` class allows developers to preview and open files. The controller presents a list of options that includes the preview option, a choice of copying the file to the system pasteboard, and the names of any application that can open and work with the file. Between this controller and SDK features for declaring supported file types in an application's Info.plist file, users are able to share files between third-party applications.

From the Library of Eric Berna

GameKit Peer Picker

The GameKit peer picker provides a standard GUI for discovering and connecting to other iOS devices. It offers a slick interface, listing other units that are available and can be linked to. Although this controller is part of GameKit, its technology is readily adaptable to nongame uses, including file transfer, messaging, and so forth.

You can configure the picker to select whether to use a Bluetooth or Internet (TCP/IP and UDP) connection. (As an alternative to Bluetooth, Wi-Fi connections on the same network are available from the 3.1 firmware and later.) When presented to the user, only the supported connections appear. Note that users cannot control that choice themselves using this interface.

Media Player Controllers

The Media Player framework offers several controllers that allow you to choose and play music and movies. The `MPMediaPickerController` provides a media-selection GUI that allows users to choose music, podcasts, and audio books. You choose which media to present, and you can play back that media via an `MPMusicPlayerController` instance.

When your user needs to watch a movie or listen to audio, an `MPMoviePlayerController` instance does the trick. Just supply it with a path to the media resource and push the controller into view. The controller provides a Done button for the user or automatically returns a delegate call when playback finishes.

View Design Geometry

The iOS hardware is not theoretically limited to a 320-by-480, 640-by-960, or 768-by-1024-pixel display. Always design your applications to be as resolution-independent as possible. That having been said, certain facts of geometry do play a role in the design of current generation iOS applications, particularly when you need to hand specs to a graphic designer to take to Photoshop.

Here is a rundown of the onscreen elements whose geometry can mostly be counted on to stay set when building your interfaces. Try not to rely on these sizes where possible, but rather design around them while keeping their proportions and aspect ratios in mind.

Keep in mind that future iPhone and iPad models and related iOS devices may not use the same screen size or shape. All the measurements in this section apply specifically to current members of the iOS family, all of which use either a 320×480 or 768×1024 geometry. Even those iPhone/iPod devices with Retina displays use a 320×480 geometry. That's because the geometry is defined in floating-point addresses, not in physical pixels.

> **Note**
>
> As this book is being written, rumors of the iPad 3 with a possible 2048-by-1536 Retina display are being floated, but there are no supporting facts yet. Always try to design your apps beyond existing unit specifications with scalable art and resolution-independent layout so you can easily adapt those resources when new models debut.

From the Library of Eric Berna

Status Bar

The status bar at the very top of the iOS screen shows the time, connectivity, battery status, and carrier (iPhones and iPad 3Gs) or model (iPods or iPads) of the unit. It offers quick access to the unit's notification center and built-in widgets. This bar is 20 points in height for normal use. It zooms to 40 points high during phone calls or when messages are displayed; note that double-height status bars appear to be a portrait-only feature. Unfortunately, the SDK does not offer any public hooks into the message display system, so you can't display your own messages. You can see these 40-point colorful status displays when you pause a Voice Memo recording, use Nike+, or tether on 3G or later units.

Figure 4-1 shows an iPhone status bar for portrait, landscape, and 40-point-high message modes. (The iPad bars are similar.) You can hide the status bar from your users, but doing so at a minimum eliminates their access to seeing the time and battery information unless you supply that information elsewhere in your application's user interface. You can set the status bar to display in gray, black, or translucent black. The latter allows the view behind it to bleed through to better coordinate colors with your application.

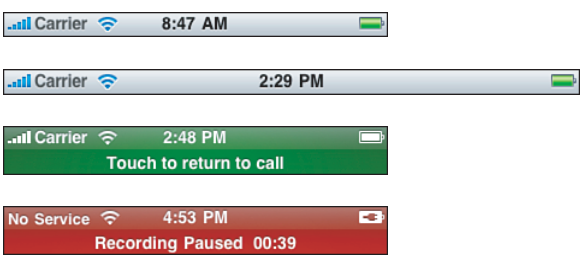


Figure 4-1 The status bar is normally 20 points high, regardless of whether the iOS device is using portrait or landscape orientation. At times, the status bar zooms to 40 points in height to indicate an ongoing system operation such as a phone call or a paused recording.

If you'd rather free up those 20 points of screen space for other use, you can hide the status bar entirely. Use this `UIApplication` call: `[UIApplication sharedApplication] setStatusBarHidden:YES animated:NO]`. Alternatively, set the `UIStatusBarHidden` key to `<true/>` in your application Info.plist file. Be aware that Apple frowns on gratuitous status-bar hiding; users generally like seeing the status bar in their applications. If you do opt to do so, consider displaying at least the time in some other fashion as an end-user option.

With the status bar displayed, your application has 320×460 or 768×1004 points to work with in portrait mode, and 480×300 or 1024×748 points in landscape mode. These numbers change depending on whatever other elements you add into the interface, such as navigation bars, tab bars, and so forth. And as already mentioned, the standard iPhone, iPod, and iPad geometries may change over time as Apple releases new models and new related touch-based products that run iOS.

From the Library of Eric Berna

The status bar plays a role in both landscape and portrait orientations, adjusting to fit as needed. To run your application in landscape-only mode, set the status bar orientation to landscape. Do this even if you plan to hide the status bar (that is, `[[UIApplication sharedApplication] setStatusBarOrientation: UIInterfaceOrientationLandscapeRight]`). Alternatively, set `UIInterfaceOrientation` in your Info.plist to the string `UIInterfaceOrientationLandscapeLeft` or `UIInterfaeOrientationLandscapeRight`. These options force windows to display side to side and produce a proper landscape keyboard.

A few further points:

- Although you can submit landscape-only or portrait-only iPad applications to the App Store, these apps must run in *both* landscape (left and right) modes or *both* portrait (standard and upside down) modes. Apple prefers applications that provide usable interfaces in all orientations.
- You can now create custom input views to replace standard keyboards, as described in Chapter 10. The geometry of these custom views must match the device orientation.
- You can hide and show the status bar during the life of your application. The `UIApplication` method `setStatusBarHidden:withAnimation:` allows you to animate the status bar in and out of view.

> **Note**
>
> Use Hardware > Toggle In Call Status Bar to test your interfaces in the simulator using the 40-point-high status bar.

Navigation Bars, Toolbars, and Tab Bars

By default, `UINavigationBar` objects (see Figure 4-2) are 44 points in height in iPhone and iPod portrait mode and for both portrait and landscape mode on the iPad. They are 32 points high in landscape mode on iPhone and iPod touch. They stretch from one side of the screen to the other, so their full dimensions are 320×44 or 768×44 points and 480×32 or 1024×44 points.

Navigation bars offer a seldom-used "prompt" mode that extends the height by 30 points. In portrait mode, the bar occupies 320×74 or 768×74 points, and in landscape, 480×74 or 1024×74, using a 44-point-high navigation bar for the landscape iPhone and iPod touch rather than the normal 32-point-high version.

> **Note**
>
> To add a prompt to a navigation bar, edit the view controller's navigation item—that is, `self.navigationItem.prompt = @"Please click a button now";`.

Tab bars are 48 points high in both orientations: 320×48 points and 480×48 points. I do not give the equivalent iPad dimensions (they are 768×48 points and 1024×48 points) because Apple guidelines suggest that you do not use tab bars directly in your iPad applications. You may feature them, however, in your popovers, using whatever widths you require. Sticking to iPhone/iPod dimensions for popovers provides a familiar geometry.

From the Library of Eric Berna

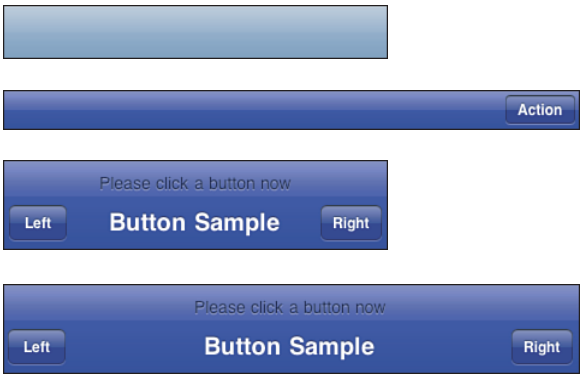


Figure 4-2 Navigation bars stretch from one side of the screen to the other. Their height is fixed at 44 points (32 points for landscape on iPhone and iPod touch). The rarely used prompt feature (shown in the bottom two images) zooms the bar to 74 points high.

According to Apple, the individual items on tab bars should be designed with a minimum 44×44-point hit region to provide sufficient space for users to tap. That corresponds to individual art of about 30×30 points.

Figure 4-3 shows a typical tab bar and its near-cousin class, the toolbar. Toolbars use the same 44-point spacing as navigation bars but, like tab bars, they're meant to be displayed at the bottom of the screen on iPhone and iPod units and incorporated into a bar at the top of the screen for iPad devices.

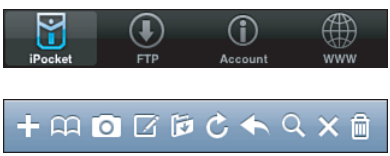


Figure 4-3 Tab bars are 48 points high for 320x480-point iPhone and iPod units (top). Toolbars use the same 44-point spacing as standard navigation bars.

These two UI elements aren't generally meant for landscape mode use. You can see this with both the iPod and YouTube applications on iPhone and iPod touch. These apps swap out a toolbar-based portrait view for a completely separate landscape presentation: Coverflow for iPod, movies for YouTube.

From the Library of Eric Berna

Between status bars, navigation bars, tab bars, and toolbars, you need to apply some basic math to calculate the remaining proportions available to background design. A typical application with a navigation bar and status bar leaves a central area of 320×416 or 768×960 for portrait display and 480×268 or 1024×704 for landscape. On the iPhone or iPod touch, using tab bars or toolbars effectively diminishes the available height by another 48 or 44 points and the resulting proportions change accordingly. On the iPad, toolbars are generally incorporated into the top navigation bar and the central area remains unaffected.

Note

Tab bars have a specifically intended use: to switch between application modalities. Tab bars are not meant to be used as a Safari-like bookmark bar.

Keyboards and Pickers

The standard iPhone and iPod touch keyboard uses 320×216 points for portrait presentation and 480×162 for landscape. The standard iPad keyboard is 768×264 points for portrait presentation and 1024×352 for landscape. You can also add custom input accessory views to add buttons (such as a Done button) and other controls on top of the keyboard whenever it is presented for certain text fields and views. The height of that accessory view adds to the space occupied by the keyboard. Adding a simple 44-point toolbar to a standard iPhone keyboard leaves almost no room whatsoever in landscape mode, so use accessory views cautiously.

Figure 4-4 shows a set of keyboards in their default configuration in both its orientations as well as an accessorized keyboard in a portrait orientation. When a text element becomes active in your application, the keyboard displays over any elements at the bottom of the screen, leaving a shortened space at the top for interaction. Complex keyboard layouts using custom accessory views or even a completely custom input view (see Chapter 9) may occupy even more onscreen room.

You may want to resize your main view when the keyboard displays. When you have several onscreen elements to edit, a shortened scrolling view works best. This lets your users access all possible areas by scrolling and won't leave text fields or text views hidden behind the keyboard. Solutions are demonstrated at length in several recipes in Chapter 10, along with ways to ensure that your user can dismiss the keyboard.

Note

`UISegmentedControl`s are typically 44 points high in their standard text-based form.

From the Library of Eric Berna



Figure 4-4 Both the portrait and landscape keyboards occupy a large part of the iOS screen. Design your applications accordingly. Here you see an iPhone/iPod touch keyboard in portrait (A) and landscape (B) orientation, an iPad keyboard in portrait (C) and landscape (D) orientation, and a portrait iPad keyboard using a custom accessory view (E).

From the Library of Eric Berna

Text Fields

When working with `UITextField` instances, allocate at least 30 points in height. This allows users enough room to enter text using the default font size without clipping.

The UIScreen Class

The `UIScreen` object acts as a stand-in for iOS's physical screen, which you can access via `[UIScreen mainScreen]`. This object maps standard window layout boundaries into pixel space. It takes into account any toolbars, status bars, and navigation bars in use.

To recover the size of the entire screen, use `[[UIScreen mainScreen] bounds]`. This returns a rectangle defining the full point size of iOS's screen. As mentioned earlier in this chapter, an iOS device screen may not always be 320×480 or 768×1024 points in size should Apple introduce new units.

Another method call returns the central application space. Call `[[UIScreen mainScreen] applicationFrame]` to query this value. On an iPhone or iPod touch, for an application that uses a status bar and a navigation bar, this might return a size of 320×416 points, taking into account the 20-point status bar and 44-point navigation bar. Use these numbers to calculate the available space on your iPhone screen and lay out your application views when not using Interface Builder.

iOS allows you to attach additional screens to your iOS device through the use of external cables. Retrieve an array of screen objects by sending the `screens` class method to the `UIScreen` class. The first item in that array is equivalent to the `mainScreen` screen. You can query each screen's `bounds` and `applicationFrame` independently.

Building Interfaces

There's more than one way to build an interface. With iOS SDK, you can build a GUI by hand using Objective-C, or you can lay it out visually using Xcode's Interface Builder. When coding, you programmatically specify where each element appears onscreen and how it behaves. With Interface Builder, you lay out those same elements using a visual editor. Both approaches offer benefits. As a developer, it's up to you to decide how to balance these benefits.

In the end, both technologies take you to the same place. The code used in Objective-C corresponds directly to the layout used in Interface Builder, and the callback behavior set up in Interface Builder produces identical results to those designed in Objective-C.

Yes, the implementation details differ. A hand-built version might use `viewDidLoad` to add its interface elements to the default view, or `loadView` to create a view from scratch. In contrast, a storyboard-based view controller often finishes setting itself up in `viewDidLoad` after loading a prebuilt interface from a .storyboard or .xib file. Cocoa Touch supports both these approaches, plus you can use a hybrid approach, loading .storyboard or .xib files via direct Objective-C commands.

The next few sections show you various ways to use these tools. You learn how to build a basic storyboard, walk through a full IB application approach, and then a full

From the Library of Eric Berna

Xcode one. After, you'll find a pair of further hybrid solutions. All these walkthroughs produce identical end products, offering identical functionality, and could be applied to both iPhone and iPad interfaces.

> **Note**
>
> Make sure you have worked through the Hello World examples in Chapter 3, "Building Your First Project," so you have a starting point for understanding Xcode and Interface Builder. The examples in this chapter go into greater depth but assume you've already learned some of the basic vocabulary for using these tools.

Walkthrough: Building Storyboard Interfaces

New to the iOS 5 SDK, storyboards allow you to lay out all the screens of your application using a single editor. Storyboards contain an entire UI in a single file, letting you design both the look of each screen and the way screens transition from one to another. By default, Interface Builder creates separate iPhone/iPod touch and iPad storyboards so you can customize your GUI for each deployment platform.

In this walkthrough you create a new storyboard application and learn how to create "scenes," the views that you see, and "segues," the process of moving from one scene to the next. For conciseness, these steps focus on the iPhone storyboard rather than creating a mirrored project on the iPad.

Create a New Project

Launch Xcode and choose File > New > New Project > iOS > Application > Single View Application. Click Next. Enter **Hello World** as the project name, and if you have not yet done so, set up your company identifier (for example, com.sadun). Set the Device Family to Universal (even though this walkthrough will not use the iPad storyboard) and leave Include Unit Tests unchecked. Set **Hello_World** as your prefix, with the underscore between the words. Click Next. Specify where to save your project (for example, the desktop) and click Create. Xcode opens a new browser containing your project.

The project consists of two storyboards—one for iPhone, one for iPad—as well as an app delegate class and a view controller class. Click MainStoryboard_iPhone.storyboard to open the storyboard in Interface Builder. With the default settings, the storyboard consists of a single scene made up of a basic view controller.

Add More View Controllers

Open the Utility pane by selecting View > Utilities > Show Utilities. This presents the inspector view at the top of the pane and the library view at the bottom. Use the search bar in the library to locate a navigation controller and drag it into your IB workspace, onto the grid-pattern background. Even while using a iPhone storyboard, as opposed to a much larger iPad one, you will instantly see why it helps to invest in a very large monitor to perform this kind of design work. Go ahead and hide the Navigator pane (View >

From the Library of Eric Berna

Navigators > Hide Navigator) to make a little more room. Figure 4-5 shows what your Xcode window will look like at this point.

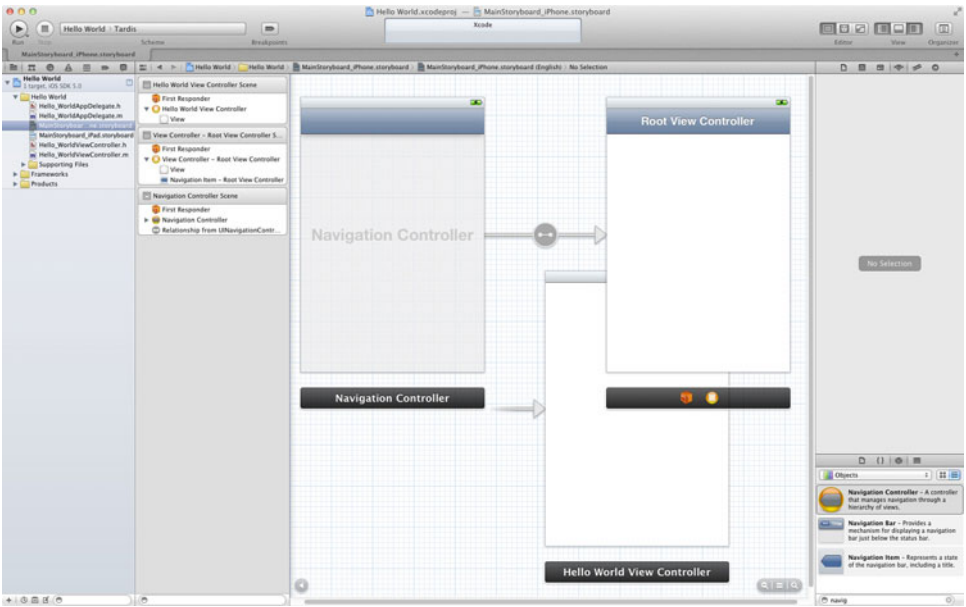


Figure 4-5 You'll need quite a large monitor to work with Interface Builder's new storyboard layout mode. This storyboard includes three scenes, as shown in the left column, consisting of a Navigation Controller and two view controllers.

Notice how the navigation controller is automatically attached to a view controller. A line with an arrow connects them. The Navigation Controller background is translucent; the Root View Controller is opaque white. This storyboard includes three scenes, as shown in the left column, consisting of a navigation controller and two view controllers. Now search for "view controller" in the library and add two more view controllers to the interface. This gives you a total of four view controllers and one navigation controller. Interface Builder lists all five scenes in the column at the left.

Organize Your Views

Set these elements up in the drawing space in the following way. Create a line of the Navigation Controller followed by the Root View Controller followed by the two new view controllers. Place the Hello World View Controller underneath the Root View Controller. To see all your elements at once, double-click the grid background. This shrinks down the elements to a more manageable size. You cannot edit your views in this presentation but it's easier to organize them. Figure 4-6 shows the layout you are aiming for. In the IB editor, notice that the Hello World View Controller at the bottom has an incoming arrow

From the Library of Eric Berna

connected to nothing else. This indicates that the view is currently set as the storyboard's entry point. You will update this later.

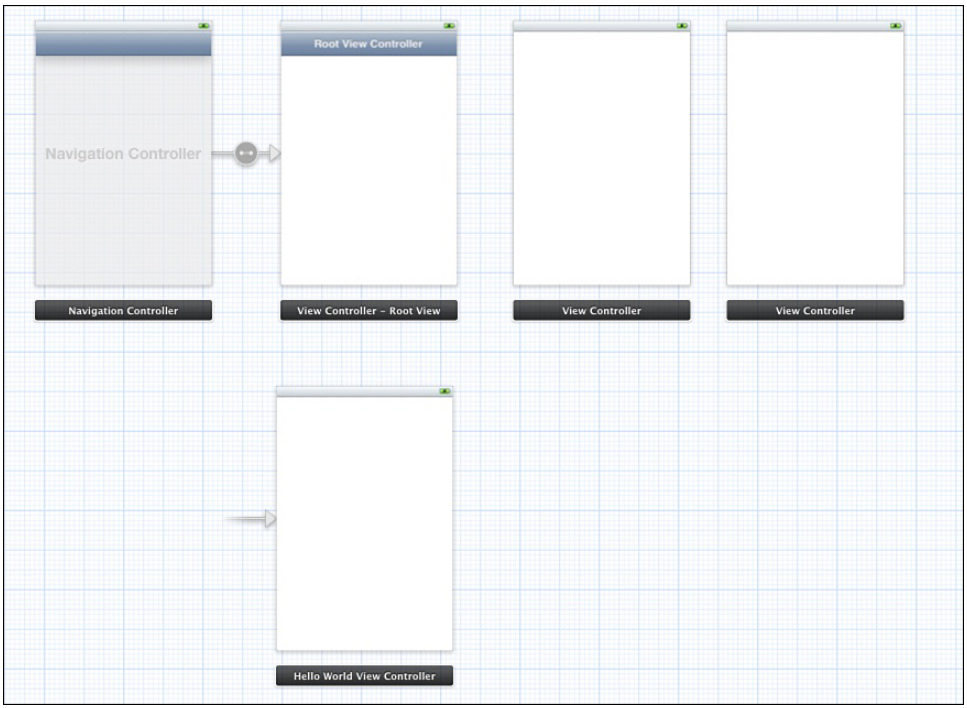


Figure 4-6 Organize your views in preparation for customizing and connecting them up.

Organizing your views allows you to get a sense of how they're going to connect to each other. In this case, you're going to build a fairly standard Navigation Controller presentation along the top row, with a modal "settings" view using the bottom controller. The class for each controller (Navigation Controller, View Controller, Hello World View Controller) appears at the bottom of each scene in a floating dock that moves along with its view.

Update Classes

By default, Xcode uses a primary custom class when creating its template. That's why the Hello World View Controller belongs to a different class than the two view controllers you pulled from the library. Select one of the two other controllers and open the identity inspector (View > Utilities > Show Identity Inspector or click the third item at the top of the Utilities pane). Use the Custom Class > Class pop-up at the top of the inspector to change the controller's class to `Hello_WorldViewController`. The controller updates and

From the Library of Eric Berna

should now match the one on the other row. Repeat for the remaining controller and the one attached to the Navigation Controller so all four controllers are members of the same `Hello_WorldViewController` class.

The identity inspector lets you detect which class any IB item belongs to and reassign that class to another member of the same family. `Hello_WorldViewController` descends from `UIViewController`, so the class identity can be updated from the parent class to the child without affecting any of the instance variables or methods required by that class. After you change the class for your view controllers, your storyboard consists of one navigation controller and four instances of the Hello World View Controller class.

Name Your Scenes

Naming each element in the storyboard lets you instantly know its role. Do this using the same identity inspector. Select the first view controller, the one connected to the Navigation Controller. Look in the Utilities pane below the "Custom Class" used above. Change the Identity > Label from the grayed-out "Xcode Specific Label" to Yellow. In the scene list, its title updates from Hello World View Controller Scene to Yellow Scene. Repeat for the next view controller, changing its label to Green, and the next to Red. Finally, change the controller on the other row to Blue. Double-click the checked background to bring the entire storyboard back into editing mode.

Edit View Attributes

This walkthrough does not create a functional application, but what you're about to do is color in each view's background so it's instantly recognizable which view is on-screen at any time. Click the view background of the Yellow scene and open the attributes inspector. It's the fourth of the six items at the top of the Utilities pane. (You can also choose View > Utilities > Show Attributes Inspector.) Make sure the gray bar at the very top of the utilities pane, just under the button toolbar, says "View." If it does not (for example, if it says Simulated Metrics), you've clicked the wrong part of the view. You want to click in the white background.

In the attributes inspector, locate View > Background and click the background swatch. A color palette appears. Pick a yellow color; the view's background updates to match it. Repeat for the other three controllers, matching each background to the color name. If you are severely color-blind, use shades of gray or otherwise accommodate as needed; you can update the controller names to match (for example, dark gray, light gray, black, white). The key lies in knowing exactly which scene you are looking at with a glance.

Add Navigation Buttons

In the library, search for Bar Button Item. Drag one onto the bar that currently says "Root View Controller." Double-click it and change its text from Item to Green. Control-drag (or right-button-drag) from the Green button to the Green view controller (see Figure 4-7).

From the Library of Eric Berna

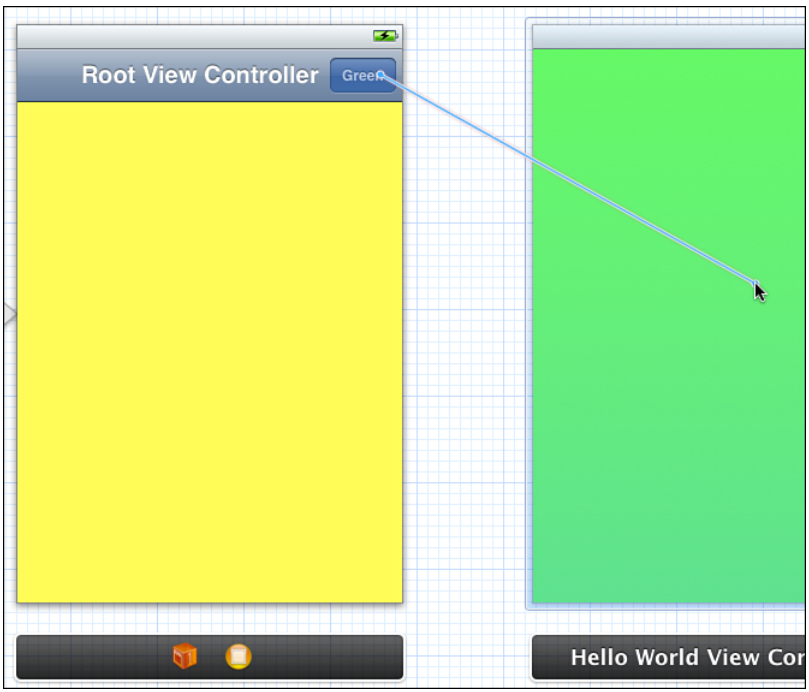


Figure 4-7 Use Control-drag (right-button-drag) to connect elements in Interface Builder. A line follows your drag, showing which object connects to the other.

A pop-up appears allowing you to select Push, Modal, or Custom. Select Push. The green scene flashes briefly and an arrow appears between the two view controllers (see Figure 4-8). The image in the circle indicates the role of the connection:

- A line with a dot on each end shows ownership. In Figure 4-8, the Navigation Controller owns its Root View Controller.
- A box with an arrow pointing left in it means that the destination controller pushes itself onto the navigation stack, over the previous view controller. When the user taps the Green button in the Root View Controller's navigation bar, the green controller will push onscreen and take control.
- A box with a square in it indicates a modal presentation. When connected up in this fashion, the new view controller is presented modally, over the calling controller. You can use a number of transitions to do this, which are set in the Storyboard Segue > Transition pop-up in the attributes inspector.
- A circle with curly brackets (that is, {}) indicates a custom segue style, one that is created in code outside of Interface Builder.

From the Library of Eric Berna

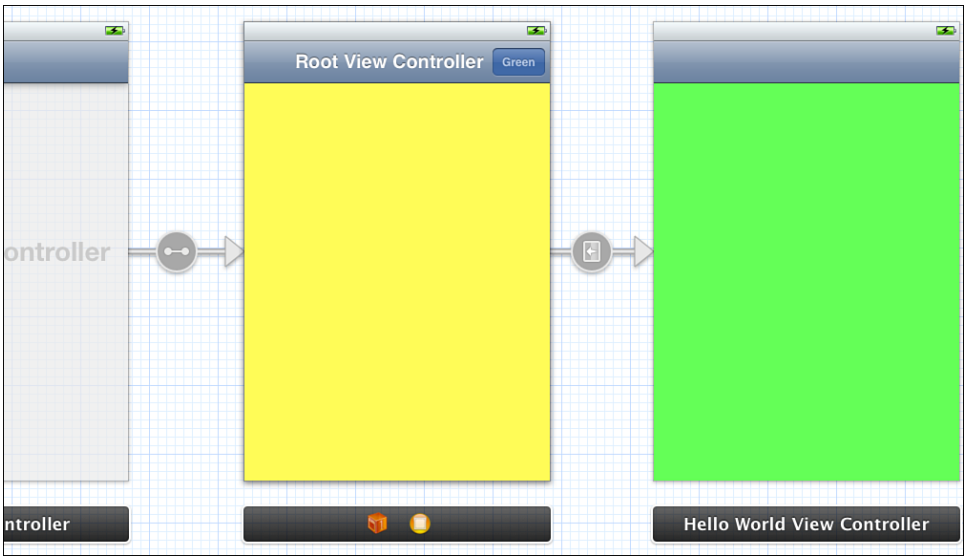


Figure 4-8 Segue arrows indicate how scenes are connected together.

Examine connection types in the attributes inspector by selecting the segue. As you saw with the initial connection pop-up, you can pick Push, Modal, and Custom styles. The modal style offers four transition options: cover vertical, flip horizontal, cross-dissolve, and partial curl. For now, leave the segue style as Push.

As Figure 4-8 shows, connecting the green view controller to the yellow one has added a navigation bar. The green scene is now part of the navigation controller's hierarchy. The navigation bar appeared as a courtesy.

Repeat the previous steps to add a bar button to the green controller. Edit it from Item to Red. Control-drag to connect it to the red controller.

Add Another Navigation Controller

Drag a new Navigation Controller out from the library onto the IB workspace. Select just the new Root View Controller and delete it, leaving just the Navigation Controller. Move the new controller to the left of the blue scene and Control-drag from it to the blue scene. A pop-up appears as before. This time choose `rootViewController`. This is a property of the Navigation Controller. The new connection that appears shows the double-dotted line in its circle. Finally, a navigation bar appears in the blue controller. Drag a bar button item to its right side and then edit its text to Done.

Name the Controllers

Next, name each Navigation Controller. Select the one in the top row, open the identity inspector, and change its Identity > Label to Primary Navigation. In the bottom row, name the new controller Modal Navigation. Although labeling objects is not used outside

From the Library of Eric Berna

of Xcode, these labels help you better organize objects within your storyboards, providing at-a-glance identification.

View controllers can have titles, whose text appears in the center of the navigation bar while an application runs. Double-click in each navigation bar in the center and edit its title. Use the same names (that is, Yellow, Green, Red, and Blue) that identify these controllers in your storyboard. These titles are used to create "back" buttons during the application's execution.

In addition, add identifier names to each view controller. Select the controller and open the attributes inspector. Locate the View Controller > Identifier field. The View Controller section appears below the Simulated Metrics section. Identifiers allow you to extract scenes from storyboard files, which is a very handy tool in your developer arsenal. Make a habit of adding identifiers to allow you to access individual scenes from code.

Tint the Navigation Bars

I'm not a big fan of the blue-gray standard navigation bars. You can easily change this tint for each of your two Navigation Controller hierarchies. Use the scene list on the left side of Interface Builder to access the navigation bars. Select the navigation controller (it is an amber circle with a gray-blue shape in it) and open its disclosure triangle. The navigation bar is listed below. Select it.

Use the attributes inspector to change the Navigation Bar > Tint. You may have to use the scene list on the left. Click the color swatch to open the color picker and choose a medium gray instead.

Notice that the navigation bar belongs to the Navigation Controller but that each navigation item belongs to a View Controller. This mirrors how you access these objects in code.

> **Note**
>
> From code, you can globally set a default tint across an entire class using the new `UIAppearance` protocol—for example, `[[UINavigationBar appearance] setTintColor:[UIColor blackColor]];`. This call affects all instances of that class, both existing and yet-to-be-created.

Add a Button

Drag a bar button item into the left slot of the yellow controller's navigation bar. Open the attributes inspector and change Bar Button Item > Identifier to Bookmarks. The updated button looks like a small open book. Control-drag from the info button to the modal navigation controller, and connect it up as you did before. This time, the arrow snakes its way around to the next row to connect to the controller scene.

Click the segue to select it. It's easiest to do this by clicking the arrowhead next to the Modal Navigation scene. In the attributes inspector, change the style from Push to Modal and from Default to Flip Horizontal. This option means that the view will flip around on when the info button is clicked.

From the Library of Eric Berna

Change the Entry Point

As you can tell from its incoming arrow, the blue scene is still the entry point for your application. Change that by selecting the primary navigation controller. Open the attributes inspector. Check View Controller > Initial Scene > Is Initial View Controller. You do not have to edit the blue scene. Its status automatically updates and its incoming arrow disappears.

You can actually see the incoming arrow best when you zoom out to the miniaturized overview. In this presentation, the trailing end of the arrow narrows and then disappears. This offers a better visual indication of the arrow's role than the standard full view

Add Dismiss Code

You need to add a little code to allow the modal blue scene to dismiss itself. Before you do this, you need to make room. Hide the Utility pane (View > Utilities > Hide Utilities). Then show the assistant view (View > Assistant Editor > Show Assistant Editor). Click the blue scene. The editor updates to show the Hello_WorldViewController.h source code file.

Here's the fun part. Control-drag from the new Done button into the code itself, between the `@interface` and `@end` lines (see Figure 4-9). A pop-up appears. Set the Connection type to Action, leave the Object as Blue, edit the name to `dismissModalController:`, and leave the type as `id`. Click Connect. Xcode automatically adds a new method declaration in the header and creates an empty method skeleton in the implementation file:

```
@interface Hello_WorldViewController : UIViewController
- (IBAction)dismissModalController:(id)sender;
@end
```

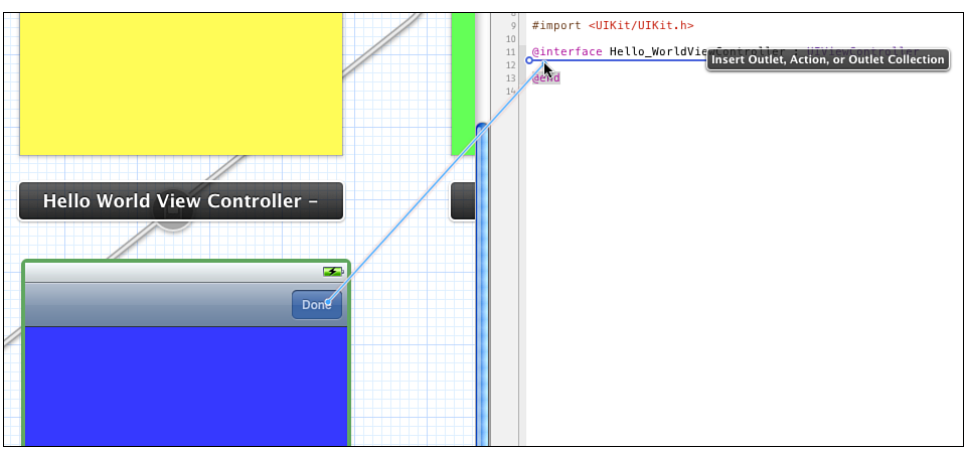


Figure 4-9 Xcode allows you to create button actions by dragging between the visual Interface Builder presentation and source code files.

From the Library of Eric Berna

Switch over to the code's counterpart (that is, its implementation file) by typing Control-Command-Up Arrow. (Control-Command-Down Arrow also works.) Locate the `dismissModalController:` method and edit it to say the following:

```
- (IBAction)dismissModalController:(id)sender
{
    [self dismissModalViewControllerAnimated:YES];
}
```

When the Done button is pressed, it will now cause the modal view (itself) to be dismissed, returning control to the primary Navigation Controller.

Run the App

Save all your work. Locate the scheme menu just to the right of the Run and Stop buttons in the top Xcode toolbar. Use the right side of that pop-up menu to select iPhone Simulator as your destination. Click Run and wait for Xcode to launch the app.

As described in these steps, the application you built consists of a three-stage navigation drill, from yellow to green to red. You can push your way forward through these screens and pop your way back. When viewing the yellow screen, you can also flip to the blue modal presentation and back. This project demonstrated several common design patterns for very simple iPhone/iPod touch applications.

Popover Walkthrough

You've seen how to create and connect view controllers for both navigation drilling and modal presentation. Now it's time for the iPad to take center stage. The iPad differs from the iPhone in several ways. First, and most obviously, the screen is much, much bigger. If you thought storyboarding with iPhone scenes was screen hogging, iPad views are going to blow your mind. Second, iPads allow you to use more than one scene onscreen at a time, especially with constructs such as the split view controller and popovers.

In this walkthrough, you're going to build a popover and use it with a basic navigation view controller. It will show you how to embed a view controller into a popover and access it using IB. The results you'll build are shown in Figure 4-10. This walkthrough uses the same project as the previous one, customizing the iPad storyboard instead of the iPhone one. Make sure you open the Navigator pane and click MainStoryboard_iPad.storyboard to get started. Show the Utilities pane and then optionally hide the navigator, depending on the screen space available to you.

Add a Navigation Controller

Drag in a new Navigation Controller from the library to the drawing area. Either click on the background to scale the new item into view or use the magnifying glass tools in the bottom-right of the IB editor window. Add a new bar button to the Root View Controller's right slot. Rename the button text from item to Popover. This button will be used

From the Library of Eric Berna



Figure 4-10 You cannot entirely create iPad popovers only using Interface Builder. Some coding is required.

to open a new popover when it is pressed. Edit the bar text from Root View Controller to Popover Demo.

Next, switch the Entry Point Controller. Select the Navigation Controller. Open the attributes inspector and check View Controller > Is Initial View Controller. This allows your Navigation Controller to take charge as the application is first launched on an iPad.

Change the View Controller Class

With the view controller selected, open the identity inspector and change its class from `UIViewController` to `Hello_WorldViewController`. To manage a popover, the controller is going to need to add some custom behavior. Using a `UIViewController` subclass allows your app to add the behaviors it needs to the subclass.

Customize the Popover View

The original Hello World View Controller scene will provide your popover view. It needs to be much smaller than it is now. Select the view controller and open the attributes inspector. Set its size from Inferred to Freeform. This change allows you to resize the view independently of the device dimensions.

From the Library of Eric Berna

Next, select its view and open the size inspector (View > Utilities > Show Size Inspector). Change both its width and height to 320. Drag a label into the reduced-size view and edit its text to "Hello World." If you like, you can add in some other controls or tint the background, and so on. If you choose to use different dimensions (other than 320 by 320), make sure to reflect this when you set the popover's content size in Listing 4-1.

Make the Connections

Control-drag from the Popover button in the main demo view controller to the smaller view controller that will be the actual popover and add a new segue. Once it is added, select the segue and open the attributes inspector. Change the segue style from Push to Popover. Edit the Identifier name to "basic pop" (no quotes). Leave the other options as the defaults. When the button is pressed during runtime, and after you do a little code-fu editing, it will launch a new popover presentation.

Edit the Code

To make the popovers work without crashing horribly during runtime, you will need to edit the `Hello_WorldViewController` class code. Listing 4-1 shows the code you will need to add for both the interface and implementation. Make your changes in Xcode to reflect this listing.

This code adds a retained property called `popoverController` that holds onto the newly created popover during its lifetime. You need to do this because without an owner, the popover is quickly released and your application encounters a runtime crash. By holding onto it, your controller ensures that the popover remains available until it is dismissed.

This code also sets the owning view controller as the popover's delegate, allowing it to handle both creation and dismissal details. Should another popover be created while one already exists, the older one is dismissed, allowing double-taps on the popover button as well as multiple popover source points within the application.

The popover's content size is set in code as well, as IB does a fairly poor job (at least at the time this book is being written) of allowing you to predefine it there. Apple should have added it to the view controller's size inspector because it is a native view controller attribute.

Upon dismissing the popover—in this case, when the user has tapped outside the popover onto the background—the view controller relinquishes its ownership, allowing the popover's memory to be returned.

Listing 4-1 **Supporting an IB-Created Popover Controller in Code**

```
/*
  Contents of Hello_WorldViewController.h
*/
#import <UIKit/UIKit.h>

// Implement the popover controller delegate protocol
@interface Hello_WorldViewController : UIViewController
    <UIPopoverControllerDelegate>
```

From the Library of Eric Berna

```
// from prior walk-through
- (IBAction)dismissModalController:(id)sender;

// retain the popover during its lifetime
@property (strong) UIPopoverController *popoverController;
@end

/*
  Contents of Hello_WorldViewController.m
*/
#import "Hello_WorldViewController.h"

@implementation Hello_WorldViewController
@synthesize popoverController;

- (void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender
{
    // dismiss existing popover and release it
    if (self.popoverController)
    {
        [self.popoverController dismissPopoverAnimated:NO];
        self.popoverController = nil;
    }

    // retain the popover and set its content size
    if ([segue.identifier isEqualToString:@"basic pop"])
    {
        UIStoryboardPopoverSegue *popoverSegue =
            (UIStoryboardPopoverSegue *)segue;
        UIPopoverController *thePopoverController =
            [popoverSegue popoverController];
        UIViewController *contentVC =
            thePopoverController.contentViewController;
        contentVC.contentSizeForViewInPopover =
            CGSizeMake(320.0f, 320.0f);
        [thePopoverController setDelegate:self];
        self.popoverController = thePopoverController;
    }
}

// Release ownership of the popover when dismissed
- (void)popoverControllerDidDismissPopover:
    (UIPopoverController *)thePopoverController
{
    self.popoverController = nil;
}
```

From the Library of Eric Berna

```
// From previous walk-through
- (IBAction)dismissModalController:(id)sender
{
    [self dismissModalViewControllerAnimated:YES];
}

// Allow autorotation to all orientations
- (BOOL) shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)toInterfaceOrientation
{
    return YES;
}
@end
```

Walkthrough: Building an iOS-based Temperature Converter with IB

Interface Builder, with its interactive GUI layout tools, helps lay out visual content. The last two walkthroughs introduced storyboards, demonstrating how you can lay out multi-scene content in your applications. The next few walkthroughs shift the focus away from storyboards and to application creation. You will build a classic Fahrenheit-to-Celsius converter using a variety of Xcode/IB design approaches. In this first version the interface is laid out entirely in Interface Builder with a minimum of coding in Xcode.

If you compare this walkthrough with ones published in earlier editions of this book, you'll recognize that the steps have become quite streamlined. That's a side effect of the new storyboard redesign in Interface Builder. It is now far easier to create these simple utility apps using Xcode's updates.

Create a New Project

Launch Xcode and create a new project. Choose File > New > New Project > iOS > Application > Single View Application and then click Next. Enter **Hello World** as the product name, edit your company identifier (mine is com.sadun), select Universal, check Use Storyboard, and uncheck Include Unit Tests. Click Next. Select where to save your project (I recommend the desktop). Leave any Source Control option enabled or disabled according to your current defaults; version control will not be discussed here. Click Create. A new project window opens in Xcode.

The iPhone/iPod deployment info appears in the center of the project window. Enable all supported device orientations by activating Upside Down, the only option disabled by default.

From the Library of Eric Berna

Add Media

Next, you'll add some basic media to the project from the Supporting Artwork folder with the sample code for this chapter. Inside you'll find a number of PNG images for your use. Locate the iPhone/iPod Deployment Info in your Project Summary.

Drag icon.png to the left App Icons box. Check Copy Items into Destination Group's Folder (If Needed) and click Finish. Next, add icon@2x.png to the right box, marked Retina Display, and then Default.png to the left iPhone Launch Images and Default@2x.png to the right.

Scroll down to the iPad deployment section. Confirm that the main storyboard is set to MainStoryboard-iPad. Set the icon to Icon-iPad.png and the two launch images to the portrait and landscape Default images.

> **Note**
>
> When you use a single asset in multiple projects, you can add that file without copying. This maintains a single source version that you can update, and its changes are reflected in each of the projects that use it. On the downside, if you remove the file from any project, you might accidentally delete the original, which can affect multiple projects.

These art files provide the image used for the application icon on iOS's home screen (icon) and the image displayed as the application launches (Default). Each application you build should contain art for these. The size and name of the art varies by platform. IOS uses a 57×57-pixel icon.png file for standard displays and 114×114 for Retina displays. The iPad uses a 72×72-pixel image. The roles of these images are discussed in further detail in Chapter 1, "Introducing the iOS SDK."

Organize your media by creating a new group. Right-click (or Control-click) Hello World in the project navigator and choose New Group from the pop-up. Rename it Art and drag in all seven files. Close up the group and move it into the Supporting Files subgroup. This greatly declutters your navigator.

Interface Builder

Locate MainStoryboard_iPhone.storyboard and select it to open the file in Interface Builder. As with the previous walkthroughs, the storyboard consists of a single `UIViewController` instance. Below the controller's plain white view is a floating black dock. Inside it are two icons. To the left is the First Responder, to the right is the View's Owner (namely, the `UIViewController` that owns the view). You can hover your mouse over each item to reveal its name. Chapter 3 introduced these two items and explained their roles in Interface Builder.

To get started, delete the already-built view controller scene and drag in a new navigation controller, using the same steps from the first two walkthroughs. After doing this, you will have a single navigation controller and a single view controller in your workspace. Change the identity of your view controller's class to Hello_WorldViewController, as before. You do not have to set the launch controller; it will default to the navigation controller, and a green outline should confirm this for you.

From the Library of Eric Berna

Make the following changes to the view controller: Double-click the middle of the blue bar and type **Converter**. Drag a bar button item from the library (Tools > Library, Command-Shift-L) onto the right side of the bar. Double-click and change the word "Item" to "Convert." Figure 4-11 shows the bar after these actions have been performed.

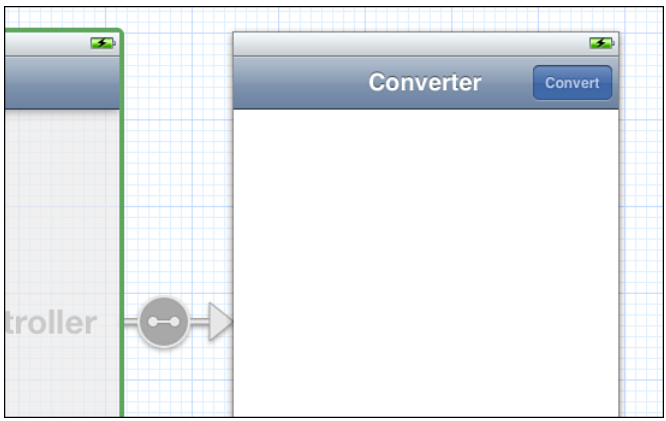


Figure 4-11 You can edit the navigation controller bar directly and add buttons to it.

Add Labels and Views

Drag two text fields and two labels into the view from the library. Then double-click the labels and edit the text, labeling the top one **Fahrenheit** and the bottom **Celsius**.

It's important to specify how you want each text field to interact with users. Among other features, you can choose which keyboard to display, whether a prompt appears in the text box, whether words are autocorrected, autocapitalized, and so forth.

Select the top text field. In the attributes inspector, choose Text Field > Keyboard > Numbers and Punctuation. This ensures that a numeric keyboard is presented when the user taps the top field. Choose Text Field > Clear Button > Is Always Visible. This adds a circled "X" clear button to your text field whenever text has been entered. Choose Text Field > Correction > No, disabling autocorrection.

Select the bottom field, which will be used to display the converted Celsius value. Uncheck Control > Content > Enabled. The bottom field shows results and should not be editable by users.

Save your changes.

> **Note**
>
> As you add more elements to your Interface Builder view, it becomes difficult to select the correct one by clicking it. One handy tip is to Control-Shift-click any view in an Interface Builder edit window to display a list of all views stacked at that point. You can choose an item from that list to select it.

From the Library of Eric Berna

Enable Reorientation

Open the Hello_WorldViewController.m file and locate the `shouldAutorotateToInterfaceOrientation:` method. Edit this to always return `YES` and save your changes.

Test the Interface

Choose the iPhone simulator as your current scheme in the top-left of the Xcode window, just to the right of the Run and Stop buttons. Click Run to build your project as-is and run it in the simulator. While the project is running, make sure that the top field opens a numbers-based keyboard and that the bottom field cannot be edited. You can click the Convert button, but it does not do anything yet. So long as your project can be compiled, you can always check your current progress in the simulator and/or on a device. Make sure to rotate the interface to ensure it works in all orientations.

Add Outlets and an Action

Outlets and actions play important roles in Interface Builder design. **Outlets** connect interfaces to objects; they essentially act as instance variable stand-ins. **Actions** are methods that your IB-created interfaces can invoke. They specify target/action pairs, sending callbacks from control views to objects. For this project you need to create two outlets and one action.

Again, this part of the walkthrough starts with the iPhone connections. With the iPhone storyboard open and displayed in Interface Builder, click the Assistant Editor button at the top-right of the Xcode screen. It is the middle of the three Editor buttons. When this button is clicked, the assistant automatically opens the counterpart for the file you're editing. In this case, Hello_WorldViewController.h should appear to the right of the IB editor.

Preparing for Creating Outlets

By default, your view controller header file looks something like this:

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

@interface Hello_WorldViewController : UIViewController
@end
```

You will greatly simplify your life with Interface Builder by manually adding braces and a little spacing, like this:

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

@interface Hello_WorldViewController : UIViewController
{


}

@end
```

From the Library of Eric Berna

This gives you lots of room to work with when adding outlets by dragging them to your code. You can edit the extraneous carriage returns out later to create a tighter presentation.

Adding Outlets

Next, create your outlet. Press and hold the Control key and drag from the top text field to the interface header file, as shown in Figure 4-12. The drag has to end inside the interface declaration, and when it does, the outlet pop-up appears.

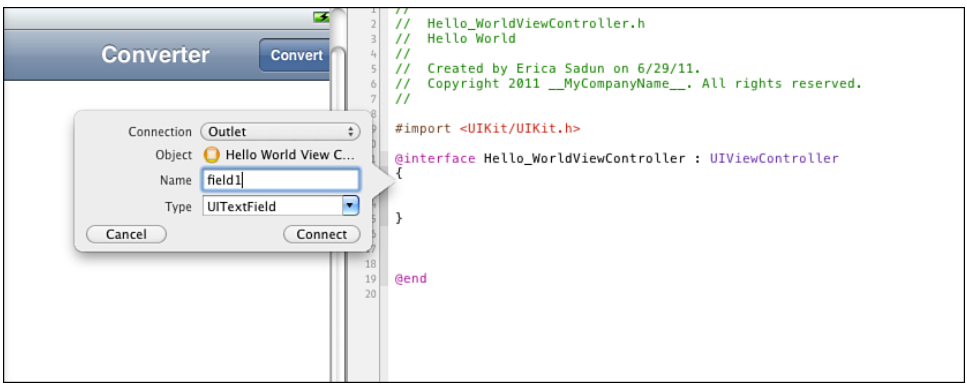


Figure 4-12 Control-drag from a text field to an interface counterpart editor in order to add an outlet. The pop-up appears when the Control-drag is released within the `@interface` declaration.

Ensure that the Connection is set to Outlet, that the type is set to `UITextField`, and then enter the name **field1**. Click Connect. This does two things. First, it creates the new outlet code for you in the text of the header file. Second, it adds the actual connection to your .xib data. Ensure that your drag ends inside the curly braces; otherwise, Xcode will add a property as well as a simple outlet. Repeat the Control-drag from the bottom field to inside the braces to add **field2**.

These actions added two instance variables to the class definition. The same approach works for adding actions by Control-dragging from buttons and other controls. Control-drag from the convert button to the interface definition, this time ending *outside* the curly braces. Set the connection to Action, the name to `convert:` (don't forget the colon), leave the type as `id`, and click Connect. Your interface should now read as follows, with all the internal storyboard hookups properly created:

```
@interface Hello_WorldViewController : UIViewController
{
    IBOutlet UITextField *field1;
    IBOutlet UITextField *field2;
```

From the Library of Eric Berna

```
}
- (IBAction)convert:(id)sender;
@end
```

> **Note**
>
> Control-click (right-click) objects to open a pop-up showing many of the same details that normally display in the connections inspector.

Add the Conversion Method

Select Hello_WorldViewController.m and replace the `convert:` method with the following. Then save and test your application on both the iPhone and iPad simulators. The program now converts Fahrenheit values into Celsius. Test with values of 32 (0 Celsius), 98.6 (37 Celsius), and 212 (100 Celsius).

```
- (IBAction) convert: (id) sender
{
    float invalue = [[field1 text] floatValue];
    float outvalue = (invalue - 32.0f) * 5.0f / 9.0f;
    [field2 setText:[NSString stringWithFormat:@"%3.2f", outvalue]];
    [field1 resignFirstResponder];
}
```

Update the Keyboard Type

Newer versions of the SDK (4.1 and later) have introduced a keyboard that includes a decimal point called the "decimal pad." In use, this keyboard appears as a number pad on the iPhone/iPod and as a standard keyboard on the iPad. Unfortunately, this keyboard style still does not appear in Interface Builder in the current Xcode 4 releases at the time this book is being written. (The NumberPad points to a no-decimal-number pad on the iPhone and to the numbers and punctuation presentation on the iPad.) To set the keyboard to the decimal pad, open RootViewController.m in the Xcode editor pane and add the field1 keyboard assignment shown in this snippet:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view from its nib.
    field1.keyboardType = UIKeyboardTypeDecimalPad;
}
```

Instead of displaying a standard keyboard with numbers selected on the iPhone, this option displays digits, a decimal point, and a delete key (see Figure 4-13). This provides a much better match for the task at hand—namely entering floating-point numbers for Fahrenheit-to-Celsius conversion.

From the Library of Eric Berna



Figure 4-13 By assigning `UIKeyboardTypeDecimalPad` to a field's keyboard type, you allow this keypad-style keyboard to limit a user's entry to numbers and a decimal point.

Connecting the iPad Interface

As far as this simple application is concerned, there's really not much of a difference between the iPhone and iPad implementations. You may want to update interface elements to make them larger, or add some art, but for this walkthrough you can just copy and paste the two labels and text fields from the iPhone version to the iPad one.

After, it takes just three connections to bring your iPad version to life. Once again Control-drag from your top text field to the interface file, but this time connect to the already-defined outlet `field1`, as shown in Figure 4-14. When the proper field highlights

From the Library of Eric Berna

and the Connect Outlet pop-up appears, let go. Repeat for the bottom field (`field2`) and the convert button (`convert:`).

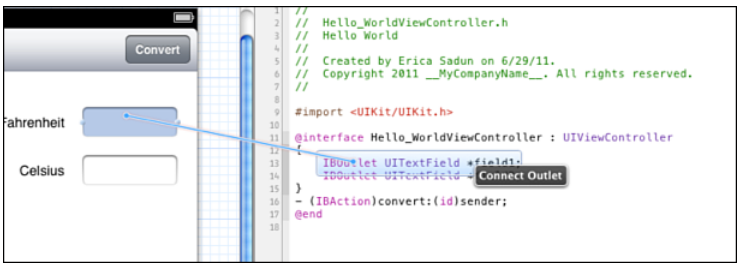


Figure 4-14 You can Control-drag to existing outlets and actions as well as create new ones.

Save your work and test your changes on the iPad simulator. After, you may want to go back and edit your iPad interface to be a bit bigger and more iPad-y.

Walkthrough: Building a Converter Interface by Hand

Anything that can be designed in Interface Builder can be implemented directly using Objective-C and Cocoa Touch. The code in Listing 4-2 duplicates the converter project you just built. Instead of loading an interface from a storyboard file, it manually lays out the elements in the `loadView` method.

The code matches the approach you used in Interface Builder. It adds two labels and two text fields. It sets the label texts to Fahrenheit and Celsius, tells the first field to use a numbers-and-punctuation keyboard, and disables the second. The locations and sizes for these items use view frames derived from the previous walkthrough.

To finish the layout, it adds a Convert button to the navigation bar. The button uses the same `convert:` callback as the IB project and calls the same code.

Listing 4-2 **Code-Based Temperature Converter**

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

// Handy bar button creation macro
#define BARBUTTON(TITLE, SELECTOR)     [[UIBarButtonItem alloc] \
    initWithTitle:TITLE style:UIBarButtonItemStylePlain target:self \
    action:SELECTOR]

@interface HelloWorldController : UIViewController
{
    UITextField *field1; // Fahrenheit
    UITextField *field2; // Celsius
```

From the Library of Eric Berna

```
}
-(void) convert: (id)sender;
@end

@implementation HelloWorldController

// Convert to Celsius
- (void) convert: (id) sender
{
    float invalue = [[field1 text] floatValue];
    float outvalue = (invalue - 32.0f) * 5.0f / 9.0f;
    [field2 setText:[NSString stringWithFormat:@"%3.2f", outvalue]];
    [field1 resignFirstResponder];
}

- (void)loadView
{
    // Create the view
    self.view = [[UIView alloc] initWithFrame:
        [[UIScreen mainScreen] applicationFrame]];
    self.view.backgroundColor = [UIColor whiteColor];

    // Establish two fields and two labels
    field1 = [[UITextField alloc] initWithFrame:
              CGRectMake(185.0, 16.0, 97.0, 31.0)];
    field1.borderStyle = UITextBorderStyleRoundedRect;
    field1.keyboardType = UIKeyboardTypeDecimalPad;
    field1.clearButtonMode = UITextFieldViewModeAlways;

    field2 = [[UITextField alloc] initWithFrame:
              CGRectMake(185.0, 72.0, 97.0, 31.0)];
    field2.borderStyle = UITextBorderStyleRoundedRect;
    field2.enabled = NO;

    UILabel *label1 = [[UILabel alloc] initWithFrame:
                       CGRectMake(95.0, 19.0, 82.0, 21.0)];
    label1.text = @"Fahrenheit";
    label1.textAlignment = UITextAlignmentLeft;
    label1.backgroundColor = [UIColor clearColor];

    UILabel *label2 = [[UILabel alloc] initWithFrame:
                       CGRectMake(121.0, 77.0, 56.0, 21.0)];
    label2.text = @"Celsius";
    label2.textAlignment = UITextAlignmentLeft;
    label2.backgroundColor = [UIColor clearColor];

    // Add items to content view
```

From the Library of Eric Berna

```
    [self.view addSubview:field1];
    [self.view addSubview:field2];
    [self.view addSubview:label1];
    [self.view addSubview:label2];

    // Set title and add convert button
    self.title = @"Converter";
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Convert", @selector(convert:));
}

// Allow app to work in all orientations
- (BOOL) shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)toInterfaceOrientation
{
    return YES;
}
@end

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

@implementation TestBedAppDelegate
- (void)applicationDidFinishLaunching:(UIApplication *)application
{
    window = [[UIWindow alloc] initWithFrame:
        [[UIScreen mainScreen] bounds]];
    UINavigationController *nav = [[UINavigationController alloc]
        initWithRootViewController:[[HelloWorldController alloc] init]];
    window.rootViewController = nav;
    [window makeKeyAndVisible];
}
@end

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

From the Library of Eric Berna

Putting the Project Together

Building this project means adapting one of Xcode's built-in templates. Start by selecting File > New > New Project (Command-Shift-N) > iOS > Application > Empty Application. Click Next. Name the new project **Hello World**, set the Device Family to Universal, unselect Use Core Data, unselect Include Unit Tests, and click Next. Save to the desktop.

This template-based project demands a little file bookkeeping. Entirely delete both AppDelegate files. They will not be used in this demonstration. Next, drag main.m out of the supporting files folder into the top-level Hello World group.

Select the Hello World project and view its Summary tab. As with the previous walkthrough, set the app icons and launch images for both iPhone/iPod and iPad deployment. Leave "Copy items into destination group's folder (if needed)" checked. Create a new Art group, move the default and icon images into that group, and then move the group itself into Supporting Files to get it out of the way.

To finish, open the main.m file, paste in the code from Listing 4-2 (it's in the sample code folder), compile the project, and run it in the simulator. What you'll find is an application that both looks and acts identical to the IB version. Instead of loading the interface from a storyboard file, this version creates it programmatically in the view controller class implementation.

In the real world, of course, you do not program your applications in main.m. You use industry coding standards and break your code into modular class units. Throughout this book, you'll find each recipe, each coding story, primarily told in a single main.m file. Pedagogically it helps to have the entire concept presented in a unified manner in a single place, with only one file to look through. This cookbook trusts you to know what to do with the ideas once you've learned about them, and it attempts to simplify the presentation of those ideas, so you don't have to wade through half a dozen or more files at a time to understand what is motivating them.

Walkthrough: Creating, Loading, and Using Hybrid Interfaces

One of the great things about Cocoa Touch is that you don't have to program entirely by hand or entirely using Interface Builder. You can leverage IB's visual layout and combine it with Xcode-based programming for a better, hybrid solution. This combines the static loading of .xib or .storyboard files provided by IB with a more reusable programmatic dynamic loading approach.

You can integrate IB files into your development without centering your design on a primarily IB approach. How you do so depends on whether you need to extract entire view controllers or just views. You may choose to load a storyboard from code and extract a view controller implementation from there. You might use individual XIB files instead of storyboard and create views from that XIB.

From the Library of Eric Berna

For lightweight reusable views, such as items you might superimpose on the screen for alerts and status updates, you're best going with the XIB approach. Storyboards center on view controllers. Although you can theoretically pull out a view from one of these controllers, you end up fighting Cocoa Touch rather than working with its strengths. Using XIBs helps you create interface elements that aren't tied to controllers. This walkthrough explores creating views from .xib files.

> **Note**
>
> Use `storyboardWithName:bundle:` to load .storyboard files from your application bundle. When working with .storyboard files, make sure you set scene (view controller) identifiers in the identity inspector before trying to extract controllers. Then you can use `instantiateViewControllerWithIdentifier:` to build the controller from code.

Create a New XIB Interface File

Cocoa Touch lets you recover objects from .xib files by calling `loadNibNamed:owner:options:`. This method returns an array of objects initialized from the XIB bundle, which you can then grab and use in your program. Use this feature to load an IB-designed interface from an otherwise hand-built application.

To get started, copy the project from the second walkthrough, the one built entirely by code. You adapt this hand-built code to use an XIB-designed view. Create an .xib file by choosing File > New > New File > iOS > User Interface > Empty. (Not Empty Storyboard.) Click Next. Set the device family to iPhone (it can be used on both platforms, fear not) and then click Next. Name it **ConverterView**, set it so it's added to the Hello World group, with Hello World as its target, and click Save. Xcode adds the file and opens Interface Builder.

Add a View and Populate It

Drag a new `UIView` (not a view controller) in from the library to your workspace. It helps to search for `UIView` even though the Xcode "name" of the class is `View`. This cuts down on the false hits in the Library's search results because "View" matches pretty much every object in the `UIKit` library.

With the view created, you need to add its contents. For simplicity, open the first converter walkthrough project in a new workspace, the one that used IB. Select both labels and text fields and copy them to memory. Close that project and paste those items into your new view. Presto, a fully populated view.

Open the attributes inspector and make sure that the top text field is set to No Correction and that the clear button is always visible. For the bottom field, uncheck Enabled.

Tag Your Views

Tagging views assigns numbers to them. All view classes provide a tag field. Tags are integers that you can utilize to identify view instances. You choose what number to use. Once views are tagged, you can retrieve them from a parent view by calling `viewWithTag:`.

From the Library of Eric Berna

Select the top text field. In the attributes inspector, edit View > Tag to 11. Select the bottom text field and set View > Tag to 12. These are arbitrary numbers. This code uses values higher than 10 to avoid conflict with any possible Apple-introduced tags.

Edit the Code

Save your changes and turn your attention back to the main.m code file. Replace the Hello World Controller class implementation with Listing 4-3. The loading code relies on the fact that there is just one actual view object in that XIB, the main `UIView`. The order of the items in the XIB file array mirrors the order of the items in Interface Builder's project window. Because this XIB contains exactly one top-level item, the code could just as easily use `objectAtIndex:0` as `lastObject`.

Notice how much more succinct Listing 4-3 is compared to the much longer Listing 4-2, which laid out all interface elements. What's more, you can keep editing the XIB interface without updating your code. This provides a great balance of orthogonality with convenience. This approach is far less tied into IBOutlets and IBActions, but at the same time it does add the burden of tag management.

Listing 4-3 **Loading Interfaces from IB Files**

```
@implementation HelloWorldController
- (void) convert: (id) sender
{
    float invalue = [[field1 text] floatValue];
    float outvalue = (invalue - 32.0f) * 5.0f / 9.0f;
    [field2 setText:[NSString stringWithFormat:@"%3.2f", outvalue]];
    [field1 resignFirstResponder];
}

- (void)loadView
{
    self.view = [[[NSBundle mainBundle]
        loadNibNamed:@"ConverterView" owner:self options:NULL]
        lastObject];
    field1 = (UITextField *)[self.view viewWithTag:11];
    field2 = (UITextField *)[self.view viewWithTag:12];
    field1.keyboardType = UIKeyboardTypeDecimalPad;

    // Set title and add convert button
    self.title = @"Converter";
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Convert", @selector(convert:));
}

- (BOOL) shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)toInterfaceOrientation
```

From the Library of Eric Berna

```
{
    return YES;
}
@end
```

Designing for Rotation

On iOS, device orientation changes are a fact of life. How you decide your application should respond to those changes presents a common design challenge. Do you resize onscreen elements, letting them grow and shrink in place like Safari does? Do you move them to new locations to accommodate the different view proportions? Or do you present an entirely different view, like the YouTube and iPod/Music apps do? Each of these choices presents a possible design solution. The one you pick depends on your application's needs and the visual elements in play.

The following sections explore these design approaches. You learn about autosizing and manual view placement as well as view-swapping approaches. In the distant past, Apple indicated it would eventually support separate landscape and portrait views in the SDK. At the time of writing, this functionality has not yet been implemented. On the iPad, the split view controller touches on a unified portrait and landscape design method, but it does so by showing and hiding a secondary panel rather than introducing a way to specify fixed view locations.

Enabling Reorientation

`UIViewController` instances decide whether to respond to iOS orientation by implementing the optional `shouldAutorotateToInterfaceOrientation:` method. This method returns either `YES` or `NO`, depending on whether you want to support autorotation to a given orientation. To allow autorotation to all possible orientations, simply return `YES`:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
        (UIInterfaceOrientation)interfaceOrientation
{

    return YES;
}
```

Possible iOS orientations passed to this method include the following:

- `UIDeviceOrientationUnknown`
- `UIDeviceOrientationPortrait`
- `UIDeviceOrientationPortraitUpsideDown`
- `UIDeviceOrientationLandscapeLeft`
- `UIDeviceOrientationLandscapeRight`

From the Library of Eric Berna

- UIDeviceOrientationFaceUp
- UIDeviceOrientationFaceDown

Of these orientations, the portrait and landscape varieties influence how a view autorotates. If your application is portrait only or landscape only, it might allow flipping between the two available orientations. This code uses the logical OR symbol (||) to combine tests into a single return value:

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation) interfaceOrientation
{
    return ((interfaceOrientation == UIDeviceOrientationPortrait) ||
            (interfaceOrientation ==
                UIDeviceOrientationPortraitUpsideDown))
}
```

When returning `YES`, the view controller uses several flags to determine how the autorotation takes place. For example, you might want to stretch subviews both horizontally and vertically:

```
contentView.autoresizesSubviews = YES;
contentView.autoresizingMask = (UIViewAutoresizingFlexibleWidth |
    UIViewAutoresizingFlexibleHeight);
```

These flags correspond exactly to settings made available in Interface Builder's size inspector, which is discussed in the next section.

You can set an application's initial interface orientation by editing its Info.plist file. The `UIInterfaceOrientation` key (also called "Initial interface orientation" when not using raw keys and values) can be set to the following:

- **`UIInterfaceOrientationPortrait`**—Portrait (bottom home button)
- **`UIInterfaceOrientationPortraitUpsideDown`**—Portrait (top home button)
- **`UIInterfaceOrientationLandscapeLeft`**—Landscape (left home button)
- **`UIInterfaceOrientationLandscapeRight`**—Landscape (right home button)

In addition, you can choose which orientations are supported by the application. Set the `UISupportedInterfaceOrientations` key ("Supported interface orientations") to an array of orientations. For a landscape-only application, add the two landscape items. Xcode 4 provides a lovely supported device orientation settings pane when you select your project and choose the Summary tab. You can set supported orientations separately for iPad- and iPhone-style deployment. Figure 4-15 shows the deployment setting options for the iPhone. Scroll down that pane to find the iPad settings.

From the Library of Eric Berna

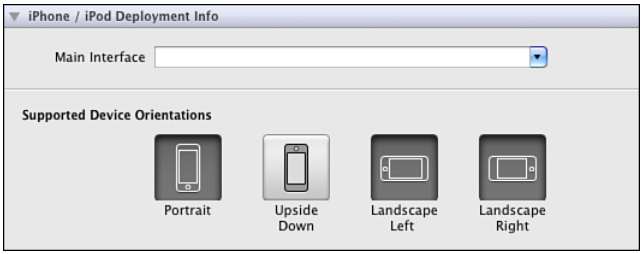


Figure 4-15 New to Xcode 4, you can choose your supported orientations for each deployment platform without having to edit the Info.plist file directly.

On iOS, you may develop App Store apps that are fixed to a single landscape or portrait orientation. On the iPad, Apple requires that users be able to use the device in more than one orientation. What this comes down to, at the time this book is being written, is the following:

- You can submit iPad applications that are landscape-only or portrait-only but they must run in *both* possible orientations. That is, your application must support both landscape-left and landscape-right *or* it must support both portrait and portrait-upside-down orientations.
- All supported orientations must be usable. If your application uses both portrait and landscape layouts, you must provide some level of usable interface for each of these presentations.
- If you do use a split view design for your application, the contents of the left pane shown in landscape mode should be available via a left-side bar button popover in portrait mode. That is, when a user taps on the top-left bar button in portrait orientation, a popover should appear that duplicates the contents of the left pane that is otherwise hidden.
- Match your Default.png choices to the orientations you plan to use. For landscape-only use, consider using a landscape-only Default-Landscape.png. You can even submit separate left and right versions, although I cannot think why you would really need to do so. When you are deploying only to the iPad, avoid submitting a single Default.png file. Apple's Technical Q&A QA1588 notes that "[The Default.png's] usage is strongly discouraged, use more specific launch images instead."

Autosizing

When you tilt an iPhone on its side in Safari, the Safari browser view adjusts its proportions to match the new orientation. It does this through autosizing. Autosizing adds rules

From the Library of Eric Berna

to a view telling it how to reshape itself. It can stretch, stay the same size, and/or be pinned a certain distance from the edge of its parent. These properties can be set by hand in code or in Interface Builder's size inspector, which is shown in Figure 4-16.

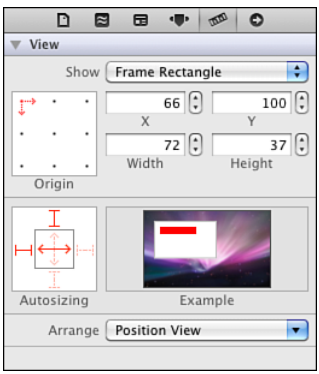


Figure 4-16 Interface Builder's Autosizing pane sets a view's `autoresizingMask`.

This pane adjusts a view's autosizing rules. The control consists of an inner square with two double-arrowed lines and an outer square with four blunt-ended lines. Each item can be set or unset via a click. When enabled, they appear in bright red; when disabled they are dim red in color.

The four outer lines are called "struts." They establish a fixed distance between a view and its parent's edge. Imagine setting a view at 40 points from the top and left of the superview. Enabling the top and left struts (as shown in Figure 4-16) fixes that view at that position. It basically pins the view in place. When you use a right or bottom strut, those distances are also maintained. The view must either move or resize to stay the same point distance from those sides.

The two inner lines are called "springs." They control how a view resizes itself. The example shown in Figure 4-16 has its horizontal spring set, allowing the view to resize horizontally in proportion to the parent view's size.

To allow a view to float—that is, to set it as both unpinned and without automatic resizing—unset all six struts and springs. This option is only available for subviews. The primary view defined in Interface Builder must be set with both springs on.

If you prefer to set these traits by hand, the two properties involved are `autoresizesSubviews`, a Boolean value that determines whether the view provides subview resizing, and `autoresizingMask`, an unsigned integer composed of the following flags, which are combined using the bitwise OR operator (|) to produce a value for the property:

From the Library of Eric Berna

- `UIViewAutoresizingNone` means the view does not resize.
- `UIViewAutoresizingFlexibleLeftMargin`, `UIViewAutoresizingFlexibleRightMargin`, `UIViewAutoresizingFlexibleTopMargin`, and `UIViewAutoresizingFlexibleBottomMargin` allow a view to resize by expanding or shrinking in the direction of a given margin without affecting the size of any items inside. These correspond to the four struts of Interface Builder's Autosizing pane (refer to Figure 4-16) but act in the opposite way. In IB, struts fix the margins; the flags allow flexible resizing along those margins.
- `UIViewAutoresizingFlexibleWidth` and `UIViewAutoresizingFlexibleHeight` control whether a view shrinks or expands along with a view. These correspond directly with Interface Builder's springs. Springs allow flexible resizing, as do these flags.

On the iPad, you can avoid a lot of orientation design headaches if you're willing to ditch about one-third of your available pixel space. A fixed 748-by-748-point view will fit in both portrait and landscape mode, with allowances made for the 20-point-high status bar. (If the status bar is hidden, a 768-by-768-point view can be used instead.) This approach, which is best used for simple utilities rather than complex applications, allows you to disable autoresizing (`UIViewAutoresizingNone`) for the view and its children, and simply rotate it as needed to match the current orientation. I have used this approach in production code and have had no problem with Apple review, but be sure to use good design common sense when handling what you do with the remaining portion of your screen.

Autosizing Example

Consider the view shown in Figure 4-17. It consists of one main view and three subviews—namely the title, a white background splash, and a small piece of art. These subviews represent three typical scenarios you'll encounter while designing applications. The title wants to stay in the same place and maintain its size regardless of orientation. The white splash needs to stretch or shrink to match its parent's geometry, and the butterfly art should float within its parent.

The autosizing behavior of each subview is set in the size inspector. The title requires only a single strut at the top. The splash needs to resize while maintaining its distance from each edge. Setting all six struts and springs (shown in Figure 4-17) produces this result. The art subview uses the opposite setting, with none of the six struts or springs in use.

In theory, you can test the view in its opposite orientation by selecting the Landscape orientation from Simulated Metrics > Orientation in the attributes inspector. (This is like clicking the small curved arrow at the top-right of the view editor window in Xcode 3's Interface Builder.) In reality, at least at the time of writing this update, it's better to test using the simulator or even on the device, providing more accurate rotation updates than the Simulated Metrics approach, which has a high probability of messing up your design.

From the Library of Eric Berna

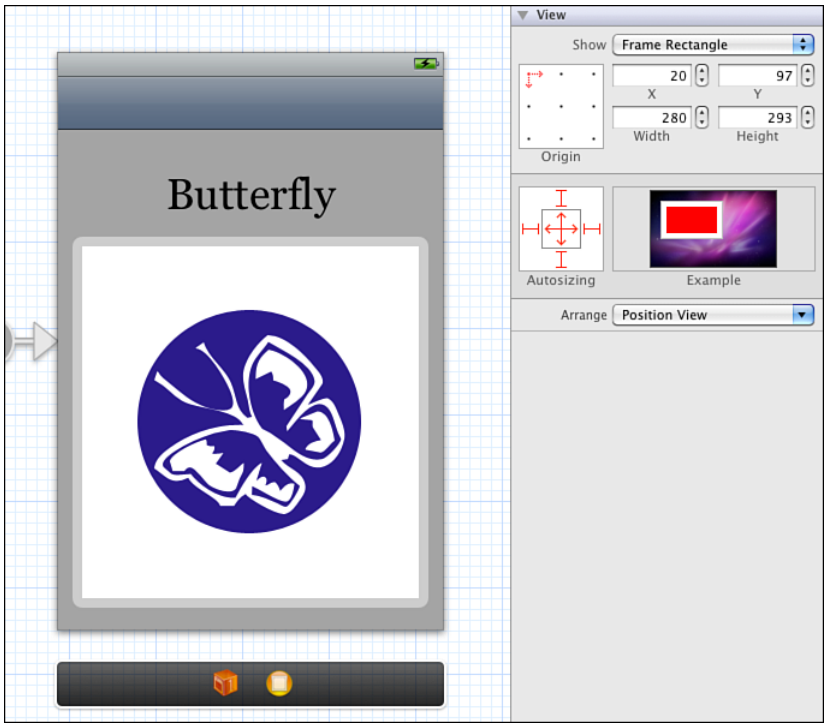


Figure 4-17 Setting view autosizing in Interface Builder.

Figure 4-18 shows the landscape version of this view using these settings, when run in the iPhone simulator. Simulating and switching between portrait and landscape presentation helps you realize how your autoresizing choices work.

Evaluating the Autosize Option

Some iOS classes work well with autosizing. Some do not. Large presentation-based views provide the best results. Web views and text views autosize well. Their content easily adapts to the change in view shape.

Small controls, especially text fields, fare more poorly. These views are not naturally elastic. Moving from landscape to portrait, or portrait to landscape, often leaves either too much room or not enough room to accommodate the previous layout. For these views you might place each item in a custom position rather than depend on autosizing. That's not to say that autosize solutions cannot work for simple layouts, just that as a general rule more complex views with many subviews do not always lend themselves to autosizing.

Image views are another class that doesn't work well with autosizing. Most pictures need to maintain their original aspect ratios. On the iPhone platform, for example, a 320×480 image shown originally in portrait orientation must shrink to 213×320 for

From the Library of Eric Berna



Figure 4-18 This is the landscape version of the view shown in Figure 4-17 using the described autosizing choices. Use the simulator's movement options of Hardware > Rotate Left (Command-left arrow) and Hardware Rotate Right (Command-right arrow) to test your views in various orientations.

landscape. That leaves you with just 45% of the portrait size. Consider swapping out art to a landscape-appropriate version rather than trying to stretch or resize portrait-based originals.

When working with autosizing, always take the keyboard into account. If your main view does not scroll or provide provisions for moving its views into accessible places, a keyboard may hide some of the views it's trying to service. Test your interfaces as you design them, both with Interface Builder's orientation selector and (preferably) in the simulator, to ensure that all elements remain well placed and accessible.

Moving Views

If autosizing provides a practically no-work solution to orientation changes, moving views offers a fix with higher-bookkeeping responsibilities. The idea works like this: After a view controller finishes its orientation, it calls the delegate method `didRotateFromInterfaceOrientation:`. Listing 4-4 implements a method that manually moves each view into place. As you can see, this approach quickly gets tedious, especially when you are dealing with more than four subviews at a time.

Listing 4-4 **Positioning Views in Code**

```
- (void)didRotateFromInterfaceOrientation:
    (UIInterfaceOrientation) fromInterfaceOrientation
{
  // Move the Fahrenheit and Celsius labels and fields into place
  switch ([UIDevice currentDevice].orientation)
```

From the Library of Eric Berna

```
  {
      case UIInterfaceOrientationLandscapeLeft:
      case UIInterfaceOrientationLandscapeRight:
      {
          flabel.center = CGPointMake(61,114);
          clabel.center = CGPointMake(321, 114);
          ffield.center = CGPointMake(184, 116);
          cfield.center = CGPointMake(418, 116);
          break;
      }
      case UIInterfaceOrientationPortrait:
      case UIInterfaceOrientationPortraitUpsideDown:
      {
          flabel.center = CGPointMake(113, 121);
          clabel.center = CGPointMake(139, 160);
          ffield.center = CGPointMake(236, 123);
          cfield.center = CGPointMake(236, 162);
          break;
      }
      default:
          break;
   }
}
```

The big advantage of this moving-subviews approach over presenting two separate views is that you maintain access to your original subviews. Any instance variables in your view controller that point to, say, a text field, continue to do so regardless of where that field is placed onscreen. The data structure of your view controller remains unchanged and independent of location, which is very Model-View-Controller compliant. The disadvantage lies in needing to specify each component's position in advance, in code, with laborious testing to update any element. If you want to move an item around to see where it looks good, Interface Builder offers a superior what-you-see-is-what-you-get approach.

Recipe: Moving Views by Mimicking Templates

There's a much simpler way to accomplish the same movement with less work. That's by creating templates. Using a storyboard-based template allows each view to update itself to match a design that you set in Interface Builder. Here's how it works.

Start with the IB Converter base project. Duplicate it and open it in Xcode. Add tags to each item in the main view. The exact numbers aren't important—what matters is that each view has a unique tag and that those tags exceed the value 10. The over-10 numbering is not official; it's a convention I use to avoid conflicting with any occasional Apple built-in tagging scheme.

From the Library of Eric Berna

Add two new view controllers. Use the attributes inspector to set their storyboard identifiers to Portrait and Landscape. Change Portrait's Simulated Metrics > Orientation to Portrait, and Landscape's to Landscape. Match the metrics to how each view will be used. For here, set the Top Bar to Navigation Bar, just as the real view does.

Select all items in the main view, which will be all four subviews. Copy those to memory and then paste them into both the portrait and landscape items. When copied, each view retains its tag, which is the critical element. Lay out the copied items in each view, exactly the way you want them to appear in that orientation, as shown in Figure 4-19.

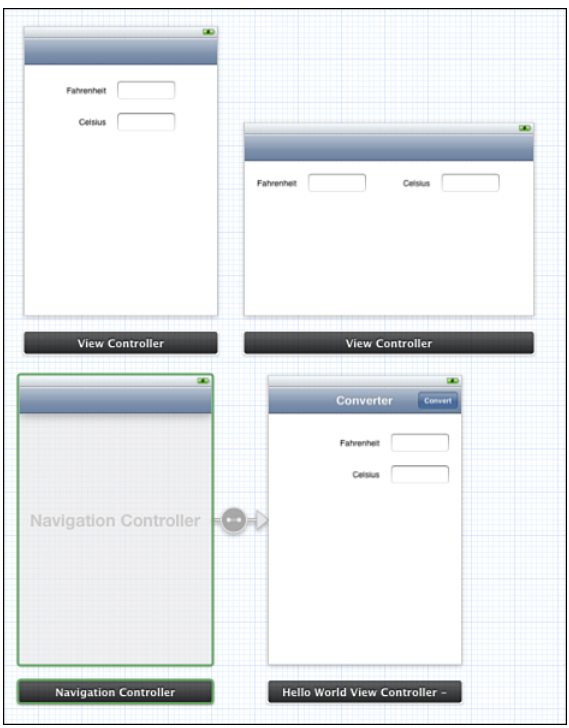


Figure 4-19 Add layout templates to your storyboards to easily respond to interface reorientation events.

Recipe 4-1 keeps using your original view and subviews after orientation changes. What you do is use those extra two view controllers as templates to determine where to place each subview. You move objects into position by matching their location in each template. This approach introduces two enormous advantages. You don't hard-code locations, and you can update the layouts in Interface Builder as needed.

From the Library of Eric Berna

Recipe 4-1 **Moving Views into Place by Matching Against a Template**

```
// Return all subviews via recursive descent
NSArray *allSubviews(UIView *aView)
{
    NSArray *results = [aView subviews];
    for (UIView *eachView in [aView subviews])
    {
        NSArray *theSubviews = allSubviews(eachView);
        if (theSubviews)
            results = [results arrayByAddingObjectsFromArray:theSubviews];
    }
    return results;
}

@implementation Hello_WorldViewController

// Respond to interface re-orientation by updating layout
- (void)didRotateFromInterfaceOrientation:
    (UIInterfaceOrientation)fromInterfaceOrientation
{
    BOOL isPortrait = UIDeviceOrientationIsPortrait(
        [[UIDevice currentDevice] orientation]);
    UIViewController *templateController = [self.storyboard
        instantiateViewControllerWithIdentifier:
        isPortrait ? @"Portrait" : @"Landscape"];
    if (templateController)
    {
        for (UIView *eachView in allSubviews(templateController.view))
        {
            int tag = eachView.tag;
            if (tag < 10) continue;
            [self.view viewWithTag:tag].frame = eachView.frame;
        }
   }
}

- (void) viewDidAppear:(BOOL)animated
{
    [self didRotateFromInterfaceOrientation:0];
}

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

From the Library of Eric Berna

```
- (IBAction) convert: (id) sender
{
    float invalue = [[field1 text] floatValue];
    float outvalue = (invalue - 32.0f) * 5.0f / 9.0f;
    [field2 setText:[NSString stringWithFormat:@"%3.2f", outvalue]];
    [field1 resignFirstResponder];
}
@end
```

Here are a few points about this approach:

- This code ignores untagged views and tags with a value under 10. Apple rarely tags views, but when it does so, it uses small numbers such as 1, 2, and 3. Make sure to tag your views starting with numbers from 10 and up.
- This example uses the storyboard to provide immediate access to the templates. The instantiated view controller selected is based on the device orientation.
- The update method is called both when the view first appears onscreen and whenever a new orientation is detected.
- Unfortunately, you cannot just use the main view as your portrait template. Once rotated, it loses all memory of the proper portrait view positions. At the same time, don't worry too much about making your main view pretty. So long as the proper items with the correct tags are somewhere in the view, it doesn't really matter where they are. The user will never see that layout.

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 4 and open the project for this recipe.

One More Thing: A Few Great Interface Builder Tips

It never hurts to have a few extra tricks up your sleeve when developing with Interface Builder and Xcode. Here are some favorite IB tricks that I use on a regular basis:

- **Selecting from stacked views**—You can use the object hierarchy view to drill down into Interface Builder's view tree. Another way to find and select subviews is by Control-Shift-clicking a view. This exposes all the views layered at that point (see Figure 4-20) and lets you select whichever item you want, regardless of whether it is the top view.
- **Naming views**—Tired of seeing each new item called "View Controller Scene" in your scene list? Give your views more meaningful names. Edit the Interface Builder Identity > Label field. The strings you assign are not used outside of Xcode, but they do help you organize your visual components.

From the Library of Eric Berna

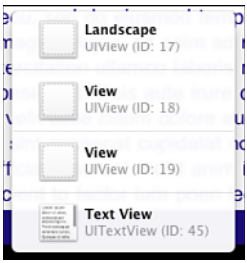


Figure 4-20 Shift-Control-click a view to pop up a view-selection dialog.

- **Moving and resizing scenes**—Drag in the status bar of each scene to move views out of the way. Use the magnifying glasses at the bottom right of the Interface Builder editor to zoom out for an overview or in for detail work.
- **Pulling in media**—Interface Builder's Media library lists the media currently available in your Xcode project. It's found in the Library browser at the bottom of Xcode's Utility pane (View > Utilities > Show Media Library). You can drag artwork from there and drop it onto a view. Interface Builder automatically creates a new `UIImageView` instance, adding the art (as a `UIImage`) to that view.
- **Moving objects**—When you're moving subviews, the arrow keys move you one point in any direction. Hold down the Shift key to move by 5 points at a time.
- **Copy objects**—Option-drag any object to copy it.
- **Show object layout**—Select any onscreen view. Hold down the Option key and hover the mouse over either the selected object or any other view to reveal the pixel-accurate layout information shown in Figure 4-21.

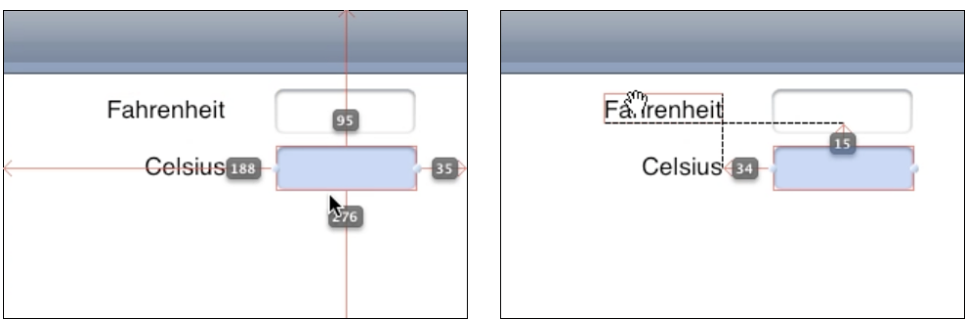


Figure 4-21 Holding the Option key while moving the mouse over views reveals how each view is placed in the view container and how views relate to the selected object in terms of distance.

From the Library of Eric Berna

Summary

This chapter introduced the basics of iOS interface design. You discovered numerous ways to build interfaces: using Interface Builder, using code, and blending the two approaches. You saw reorientation in action and learned about different ways to update your views to live in both the portrait and landscape worlds.

Before moving on to the next chapter, here are a few points to consider about laying out interfaces:

- Interface Builder excels at laying out the content of `UIView` instances. Use its tools to connect those instances to the view controllers in your program and use Interface Builder to refine WYSIWYG-style interfaces such as the temperature converter example covered in this chapter.
- Know when Interface Builder isn't the right solution. When you're building tab bars and navigation controllers with minimal window design (such as for table-based or text-based applications), you don't especially need IB's view layout tools.
- Some views work beautifully under multiple orientations. Some do not. Don't feel that you must provide a landscape version of your application that exactly matches either the look or the functionality of the portrait one, especially when you're working on smaller screen sizes. It's okay to provide different portrait and landscape experiences.
- Always, always save your work in Interface Builder. Until you do so, your project will not be updated with the current version of your files.
- There's no "right" way to design and implement portrait and landscape layouts. Choose the approach that works best for your needs and provides the best experience for your users.

From the Library of Eric Berna

This page intentionally left blank

From the Library of Eric Berna

5

Working with View Controllers

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

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

Developing with Navigation Controllers and Split Views

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

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

From the Library of Eric Berna



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

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

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

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

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

From the Library of Eric Berna

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

Using Navigation Controllers and Stacks

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

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

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

Pushing and Popping View Controllers

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

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

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

From the Library of Eric Berna

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

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

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

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

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

The Navigation Item Class

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

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

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

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

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

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

From the Library of Eric Berna

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

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

Modal Presentation

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

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

You can present a modal dialog in any of four ways, controlled by the `modalTransitionStyle` property of the presented view controller. The standard, `UIModalTransitionStyleCoverVertical`, slides the modal view up and over the current view controller. When dismissed it slides back down. `UIModalTransitionStyleFlipHorizontal` performs a back-to-front flip from right to left. It looks as if you're revealing the back side of the currently presented view. When dismissed, it flips back left to right. `UIModalTransitionStyleCrossDissolve` fades the new view in over the previous one. On dismissal, it fades back to the original view. Use `UIModalTransitionStylePartialCurl` to curl up content (in the way the Maps application does) to reveal a modal settings view "underneath" the primary view controller.

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

From the Library of Eric Berna

Recipe: Building a Simple Two-Item Menu

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

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

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

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

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

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

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

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

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

From the Library of Eric Berna

```
}

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

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

Get This Recipe's Code

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

Recipe: Adding a Segmented Control

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

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

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

Note

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

From the Library of Eric Berna



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

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

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

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

From the Library of Eric Berna

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

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

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

Get This Recipe's Code

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

Recipe: Navigating Between View Controllers

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

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

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

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

From the Library of Eric Berna

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

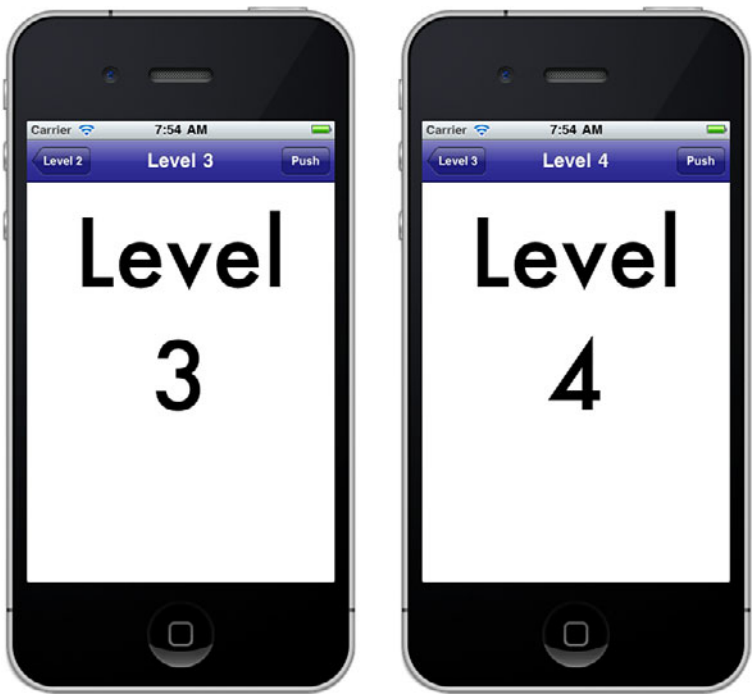


Figure 5-3 The navigation controller automatically creates properly labeled back buttons. After the Level 4 button is selected in the left interface, the navigation controller pushes the Level 4 view controller and creates the Level 3 back button in the right interface.

Recipe 5-3 **Drilling through Views with `UINavigationController`**

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

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

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

From the Library of Eric Berna

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

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

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

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

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

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

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

From the Library of Eric Berna

```
    return self;
}

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

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

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

Recipe: Presenting a Custom Modal Information View

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

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

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

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

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

From the Library of Eric Berna

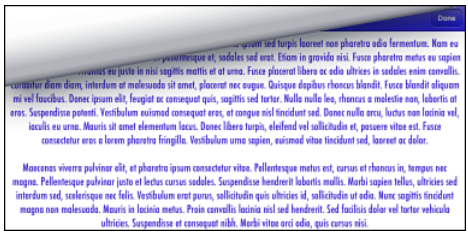


Figure 5-4 This modal view is built using `UIViewController` with a `UINavigationBar`.

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

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

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

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

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

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

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

From the Library of Eric Berna

```
{
    [self dismissModalViewControllerAnimated:YES];
}

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

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

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

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

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

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

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

From the Library of Eric Berna

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

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

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

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

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


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

From the Library of Eric Berna

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

Get This Recipe's Code

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

Recipe: Page View Controllers

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

Book Properties

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

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

From the Library of Eric Berna

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

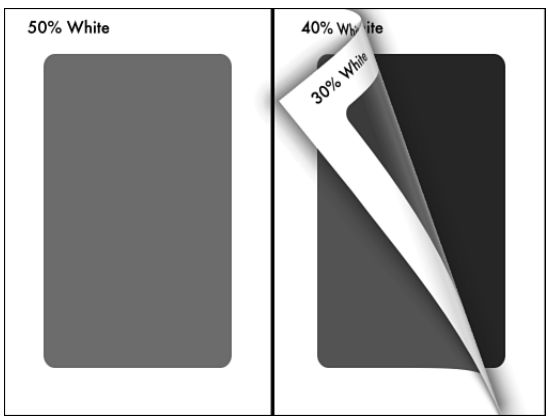


Figure 5-5 The `UIPageViewController` class creates virtual "books" from individual view controllers.

Wrapping the Implementation

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

From the Library of Eric Berna

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

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

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

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

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

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

Exploring the Recipe

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

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

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

From the Library of Eric Berna

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

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

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

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

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

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

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

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

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

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

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

From the Library of Eric Berna

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

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

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

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

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

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

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

From the Library of Eric Berna

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

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

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

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

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

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

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

From the Library of Eric Berna

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

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

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

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

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

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

From the Library of Eric Berna

```
        options:nil];

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

    return bc;
}
@end
```

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

Recipe: Scrubbing Pages in a Page View Controller

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

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

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

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

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

From the Library of Eric Berna

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

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

- (void) viewDidLoad
{
    [super viewDidLoad];

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

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

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

From the Library of Eric Berna

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

Recipe: Tab Bars

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

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

Recipe 5-7 creates 11 simple view controllers of the `BrightnessController` class. This class sets its background to a specified gray level—in this case, from 0% to 100% in steps of 10%. Figure 5-6 (left) shows the interface in its default mode, with the first four items and a More button displayed.

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

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

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

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

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

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

From the Library of Eric Berna



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

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

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

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

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

From the Library of Eric Berna

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

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

    return image;
}

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

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

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

From the Library of Eric Berna

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

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

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

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

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

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

From the Library of Eric Berna

Get This Recipe's Code

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

Recipe: Remembering Tab State

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

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

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

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

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

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

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

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

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

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

From the Library of Eric Berna

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

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

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

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

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

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

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

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

From the Library of Eric Berna

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

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

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

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

From the Library of Eric Berna

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

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

Recipe: Building Split View Controllers

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

Figure 5-7 shows the very basic split view controller built by Recipe 5-9 in landscape and portrait orientations. This controller adjusts the brightness of the detail view by selecting an item from the list in the root view. In landscape, both views are shown at once. In portrait orientation, the user must tap the upper-left button on the detail view to access the root view in a popover. When programming for this orientation, be aware that the popover can interfere with detail view, as it is presented over that view; design accordingly.

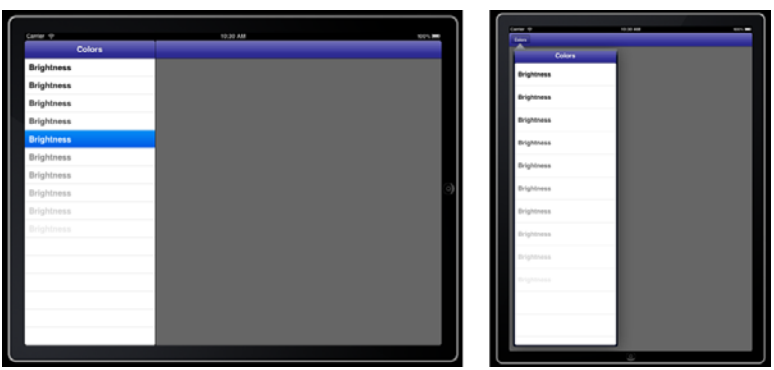


Figure 5-7 At their simplest, split view controllers consist of an organizing pane and a detail view pane. The organizing pane, which is hidden in portrait orientation, can be viewed from a popover accessed from the navigation bar.

From the Library of Eric Berna

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

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

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

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

    return svc;
}
```

The root view controller is typically some kind of table view controller, as is the one in Recipe 5-9. Tables view controllers are discussed in great detail in Chapter 11, "Creating and Managing Table Views," but what you see here is pretty much as bare bones as they get. It is a list of ten items, each one with a cell title that is tinted proportionally between 0% and 90% of pure white.

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

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

From the Library of Eric Berna

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

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

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

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

@implementation DetailViewController
@synthesize popoverController;

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

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

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

From the Library of Eric Berna

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

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


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

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

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

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

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

From the Library of Eric Berna

```
    return cell;
}

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

Get This Recipe's Code

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

Recipe: Creating Universal Split View/Navigation Apps

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

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

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

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

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

```
- (UINavigationController *) navWithColorViewController
{
    ColorViewController *colorViewController =
```

From the Library of Eric Berna

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

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

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

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

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

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

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

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

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

From the Library of Eric Berna

```
        initWithStyle: UITableViewCellStyleDefault
        reuseIdentifier:@"generic"];

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

    cell.accessoryType = IS_IPAD ?
        UITableViewCellAccessoryNone :
        UITableViewCellAccessoryDisclosureIndicator;

    return cell;
}

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

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

Get This Recipe's Code

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

Recipe: Custom Containers and Segues

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

From the Library of Eric Berna

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

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

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

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

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

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

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

From the Library of Eric Berna

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

    vcIndex = newIndex;
}
```

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



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

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

From the Library of Eric Berna

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

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

```
@implementation RotatingSegue
@synthesize goesForward;
@synthesize delegate;

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

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

    return imageLayer;
}

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

From the Library of Eric Berna

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

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

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

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

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

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

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

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

From the Library of Eric Berna

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

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

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

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

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

From the Library of Eric Berna

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

Transitioning Between View Controllers

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

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

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

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

From the Library of Eric Berna

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

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

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

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

One More Thing: Interface Builder and Tab Bar Controllers

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

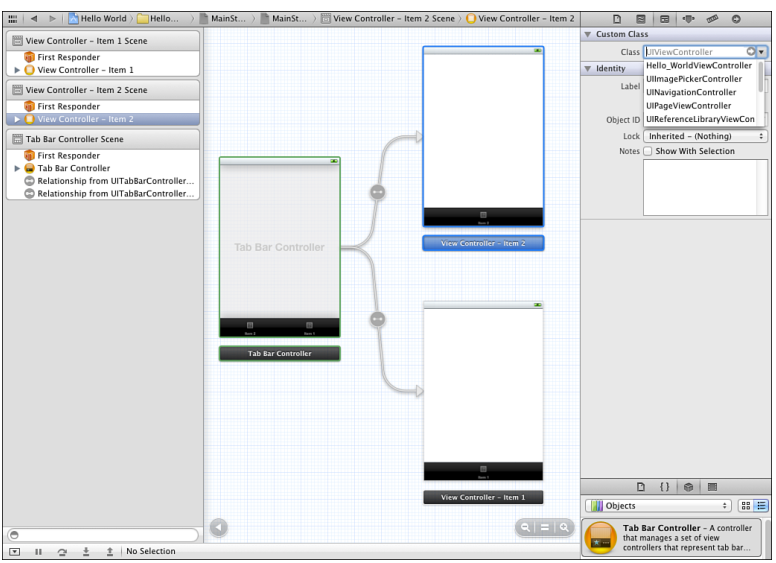


Figure 5-9 Interface Builder storyboards provide tools for laying out tab bar controllers, simplifying laying out what is essentially a logical and not a visual class, compared to what previous versions of Xcode allowed.

From the Library of Eric Berna

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

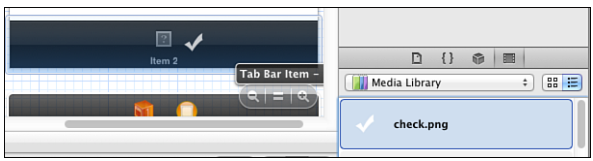


Figure 5-10 Drag art from the media library directly onto the tab bar item shown below each child view controller.

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

Summary

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

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

From the Library of Eric Berna

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

From the Library of Eric Berna

This page intentionally left blank

From the Library of Eric Berna

6

Assembling Views and Animations

The `UIView` class and its subclasses populate the iOS device screens. This chapter introduces views from the ground up. You learn how to build, inspect, and break down view hierarchies and understand how views work together. You discover the role geometry plays in creating and placing views into your interface, and you read about animating views so they move and transform onscreen. This chapter covers everything you need to know to work with views from the lowest levels up.

View Hierarchies

A tree-based hierarchy orders what you see on your iOS screen. Starting with the main window, views are laid out in a specifically hierarchical way. All views may have children, called subviews. Each view, including the window, owns an ordered list of these subviews. Views might own many subviews; they might own none. Your application determines how views are laid out and who owns whom.

Subviews display onscreen in order, always from back to front. This works something like a stack of animation cels—those transparent sheets used to create cartoons. Only the parts of the sheets that have been painted show through. The clear parts allow any visual elements behind that sheet to be seen. Views, too, can have clear and painted parts, and can be layered to build a complex presentation.

Figure 6-1 shows a little of the layering used in a typical window. Here the window owns a `UINavigationController`-based hierarchy. The elements layer together. The window (represented by the empty, rightmost element) owns a navigation bar, which in turn owns two subview buttons (one left and one right). The window also owns a table with its own subviews. These items stack together to build the GUI.

From the Library of Eric Berna

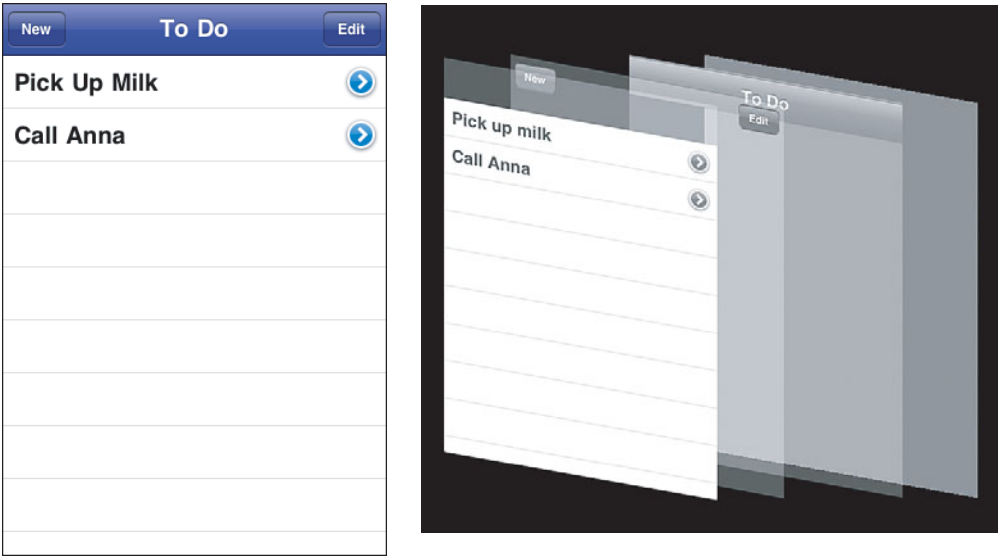


Figure 6-1 Subview hierarchies combine to build complex GUIs.

Listing 6-1 shows the actual view hierarchy of the window in Figure 6-1. The tree starts at the top `UIWindow` and shows the classes for each of the child views. If you trace your way down the tree, you can see the navigation bar (at level 2) with its two buttons (each at level 3) and the table view (level 4) with its two cells (each at level 5). Some of the items in this listing are private classes, automatically added by the SDK when laying out views. For example, the `UILayoutContainerView` is never used directly by developers. It's part of the SDK's `UIWindow` implementation.

The only parts missing from this listing are the dozen or so line separators for the table, omitted for space considerations. Each separator is actually a `UITableViewSeparatorView` instance. They belong to the `UITableView` and would normally display at a depth of 5.

Listing 6-1 **To-Do List View Hierarchy**

```
--[ 1] UILayoutContainerView
----[ 2] UINavigationTransitionView
------[ 3] UIViewControllerWrapperView
--------[ 4] UITableView
----------[ 5] UITableViewCell
------------[ 6] UITableViewCellContentView
--------------[ 7] UILabel
------------[ 6] UIButton
--------------[ 7] UIImageView
------------[ 6] UIView
```

From the Library of Eric Berna

```
----------[ 5] UITableViewCell
------------[ 6] UITableViewCellContentView
--------------[ 7] UILabel
------------[ 6] UIButton
--------------[ 7] UIImageView
------------[ 6] UIView
----------[ 5] UIImageView
----------[ 5] UIImageView
----[ 2] UINavigationBar
------[ 3] UINavigationBarBackground
------[ 3] UINavigationItemView
------[ 3] UINavigationButton
--------[ 4] UIImageView
--------[ 4] UIButtonLabel
------[ 3] UINavigationButton
--------[ 4] UIImageView
--------[ 4] UIButtonLabel
```

Recipe: Recovering a View Hierarchy Tree

Each view knows both its parent (`[aView superview]`) and its children (`[aView subviews]`). A view tree, like the one shown in Listing 6-1, can be built by recursively walking through a view's subviews. Recipe 6-1 does exactly that. It builds a visual tree by noting the class of each view and increasing the indentation level every time it moves down from a parent view to its children. The results are stored into a mutable string and returned from the calling method.

The code shown in Recipe 6-1 was used to create the tree shown in Listing 6-1. The same interface and recipe appear as part of the sample code that accompanies this book. You can use this routine to duplicate the results of Listing 6-1, or you can copy it to other applications to view their hierarchies.

Recipe 6-1 **Extracting a View Hierarchy Tree**

```
// Recursively travel down the view tree, increasing the
// indentation level for children
- (void) dumpView: (UIView *) aView atIndent: (int) indent
    into:(NSMutableString *) outstring
{
    for (int i = 0; i < indent; i++)
        [outstring appendString:@"--"];
    [outstring appendFormat:@"[%2d] %@\n", indent,
        [[aView class] description]];
    for (UIView *view in [aView subviews])
        [self dumpView:view atIndent:indent + 1 into:outstring];
}
```

From the Library of Eric Berna

```
// Start the tree recursion at level 0 with the root view
- (NSString *) displayViews: (UIView *) aView
{
    NSMutableString *outstring = [[NSMutableString alloc] init];
    [self dumpView:aView atIndent:0 into:outstring];
    return outstring;
}
```

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 6 and open the project for this recipe.

Recipe: Querying Subviews

Views store arrays of their children. Retrieve these arrays by calling `[aView subviews]`. Onscreen, the child views are always drawn after the parent, in the order that they appear in the `subviews` array. These views draw in order from back to front, and the `subviews` array mirrors that drawing pattern. Views that appear later in the array are drawn after views that appear earlier.

The `subviews` method returns just those views that are immediate children of a given view. At times, you may want to retrieve a more exhaustive list of subviews, including the children's children. Recipe 6-2 introduces `allSubviews()`, a simple recursive function that returns a full list of descendants for any view. Call this function with a view's window (via `view.window`) to return a complete set of views appearing in the `UIWindow` that hosts that view. This list proves useful when you want to search for a particular view, such as a specific slider or button.

Although it is not typical, iOS applications may include several windows, each of which can contain many views, some of which may be displayed on an external screen. Recover an exhaustive list of all application views by iterating through each available window. The `allApplicationSubviews()` function in Recipe 6-2 does exactly that. A call to `[[UIApplication sharedApplication] windows]` returns the array of application windows. The function iterates through these, adding their subviews to the collection.

In addition to knowing its subviews, each view knows the window it belongs to. The view's `window` property points to the window that owns it. Recipe 6-2 also includes a simple function called `pathToView()` that returns an array of superviews, from the window down to the view in question. It does this by calling `superview` repeatedly until arriving at a window instance.

Views can also check their superview ancestry in another way. The `isDescendantOfView:` method determines whether a view lives within another view, even if that view is not its direct superview. This method returns a simple Boolean value. `YES` means the view descends from the view passed as a parameter to the method.

From the Library of Eric Berna

Recipe 6-2 **Subview Utility Functions**

```
// Return an exhaustive descent of the view's subviews
NSArray *allSubviews(UIView *aView)
{
    NSArray *results = [aView subviews];
    for (UIView *eachView in [aView subviews])
    {
        NSArray *subviews = allSubviews(eachView);
        if (subviews)
            results = [results arrayByAddingObjectsFromArray: subviews];
    }
    return results;
}

// Return all views throughout the application
NSArray *allApplicationViews()
{
    NSArray *results = [[UIApplication sharedApplication] windows];
    for (UIWindow *window in [[UIApplication sharedApplication]
        windows])
    {
        NSArray *subviews = allSubviews(window);
        if (subviews) results =
            [results arrayByAddingObjectsFromArray: subviews];
    }
    return results;
}

// Return an array of parent views from the window down to the view
NSArray *pathToView(UIView *aView)
{
    NSMutableArray *array = [NSMutableArray arrayWithObject:aView];
    UIView *view = aView;
    UIWindow *window = aView.window;
    while (view != window)
    {
        view = [view superview];
        [array insertObject:view atIndex:0];
    }
    return array;
}
```

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 6 and open the project for this recipe.

From the Library of Eric Berna

Managing Subviews

The `UIView` class offers numerous methods that help build and manage views. These methods let you add, order, remove, and query the view hierarchy. Since this hierarchy controls what you see onscreen, updating the way that views relate to each other changes what you see on iOS. Here are some approaches for typical view-management tasks.

Adding Subviews

Call `[parentView addSubview:child]` to add new subviews to a parent. Newly added subviews are always placed frontmost on your screen; iOS adds them on top of any existing views. To insert a subview into the view hierarchy at a particular location other than the front, the SDK offers a trio of utility methods:

- `insertSubview:atIndex:`
- `insertSubview:aboveSubview:`
- `insertSubview:belowSubview:`

These methods control where view insertion happens. That insertion can remain relative to another view, or it can move into a specific index of the subviews array. The `above` and `below` methods add subviews in front of or behind a given child, respectively. Insertion pushes other views forward and does not replace any views that are already there.

Reordering and Removing Subviews

Applications often need to reorder and remove views as users interact with the screen. The iOS SDK offers several easy ways to do this, allowing you to change the view order and contents:

- Use `[parentView exchangeSubviewAtIndex:i withSubviewAtIndex:j]` to exchange the positions of two views.
- Move subviews to the front or back using `bringSubviewToFront:` and `sendSubviewToBack`.
- To remove a subview from its parent, call `[childView removeFromSuperview]`. If the child view had been onscreen, it disappears. Removing a child from the superview calls a release on the subview, allowing its memory to be freed if its retain count has returned to 0.

When you reorder, add, or remove views, the screen automatically redraws to show the new view presentation.

From the Library of Eric Berna

View Callbacks

When the view hierarchy changes, callbacks can be sent to the views in question. The iOS SDK offers six callback methods. These callbacks may help your application keep track of views that are moving and changing parents:

- `didAddSubview:` is sent to a view after a successful invocation of `addSubview:` lets subclasses of `UIView` perform additional actions when new views are added.
- `didMoveToSuperview:` informs views that they've been re-parented to a new superview. The view may want to respond to that new parent in some way. When the view was removed from its superview, the new parent is `nil`.
- `willMoveToSuperview:` is sent before the move occurs.
- `didMoveToWindow:` provides the callback equivalent of `didMoveToSuperview` but when the view moves to a new `Window` hierarchy instead of to just a new superview.
- `willMoveToWindow:` is, again, sent before the move occurs.
- `willRemoveSubview:` informs the parent view that the child view is about to be removed.

I rarely use these methods, but when I do they're almost always a lifesaver, allowing me to add behavior without having to know in advance what kind of subview or superview class is being used. The window callbacks are used primarily for displaying overlay views in a secondary window such as alerts and input views such as keyboards.

Recipe: Tagging and Retrieving Views

The iOS SDK offers a built-in search feature that lets you recover views by tagging them. Tags are just numbers, usually positive integers, that identify a view. Assign them using the view's `tag` property: for example, `myView.tag = 101`. In Interface Builder, you can set a view's tag in the attributes inspector. As Figure 6-2 shows, you specify the tag in the View section.

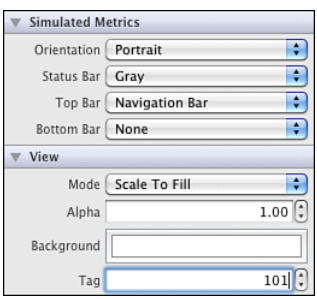


Figure 6-2 Set the tag for any view in Interface Builder's attributes inspector.

From the Library of Eric Berna

Tags are completely arbitrary. The only "reserved" tag is 0, which is the default property setting for all newly created views. It's up to you to decide how you want to tag your views and which values to use. You can tag any instance that is a child of `UIView`, including windows and controls. So if you have many onscreen buttons and switches, adding tags helps tell them apart when users trigger them. You can add a simple switch statement to your callback methods that looks at the tag and determines how to react.

Apple rarely tags subviews. The only instance I have ever found of their view tagging has been in `UIAlertViews` where the buttons use tags of 1, 2, and so forth. (I'm mostly convinced Apple left this tagging in there as a mistake.) If you worry about conflicting with Apple tags, start your numbering at 10 or 100, or some other number higher than any value Apple might use.

Using Tags to Find Views

Tags let you avoid passing user interface elements around your program by making them directly accessible from any parent view. The `viewWithTag:` method recovers a tagged view from a child hierarchy. The search is recursive, so the tagged item need not be an immediate child of the view in question. You can search from the window with `[window viewWithTag:101]` and find a view that is several branches down the hierarchy tree. When more than one view uses the same tag, `viewWithTag:` returns the first item it finds.

The problem with `viewWithTag:` is that it returns a `UIView` object. This means you often have to cast it to the proper type before you can use it. Say you want to retrieve a label and set its text, like this:

```
UILabel *label = (UILabel *)[self.view.window viewWithTag:101];
label.text = @"Hello World";
```

It would be far easier to use a call that returned an already typed object and then be able to use that object right away, as these calls do:

```
- (IBAction)updateTime:(id)sender
{
    // set the label to the current time
    [self.view.window labelWithTag:LABEL_TAG].text =
        [[NSDate date] description];
}

- (IBAction)updateSwitch:(id)sender
{
    // toggle the switch from its current setting
    UISwitch *s = [self.view.window switchWithTag:SWITCH_TAG];
    [s setOn:!s.isOn];
}
```

Recipe 6-3 extends the behavior of `UIView` to introduce a new category, `TagExtensions`. This category adds just two typed tag methods, for `UILabel` and `UISwitch`. The sample code for this book extends this to include a full suite of typed tag

From the Library of Eric Berna

utilities. The additional classes were omitted for space considerations; they follow the same pattern of casting from `viewWithTag:`. Access the full collection by including the `UIView-TagExtensions` files in your projects.

> **Note**
>
> Here's another useful thing to do with tags. When you're working with table views with cells, you can tag buttons with the `indexPath`'s row. This allows you to retrieve the cell you're working from when the button is pressed by the user.

Recipe 6-3 **Recovering Tagged Views with Properly Cast Objects**

```
@interface UIView (TagExtensions)
- (UILabel *) labelWithTag: (NSInteger) aTag;
- (UISwitch *) switchWithTag: (NSInteger) aTag;
@end

@implementation UIView (TagExtensions)
- (UILabel *) labelWithTag: (NSInteger) aTag
{
    UIView *aView = [self viewWithTag:aTag];
    if (aView && [aView isKindOfClass:[UILabel class]])
        return (UILabel *) aView;
    return nil;
}

- (UISwitch *) switchWithTag: (NSInteger) aTag
{
    UIView *aView = [self viewWithTag:aTag];
    if (aView && [aView isKindOfClass:[UISwitch class]])
        return (UISwitch *) aView;
    return nil;
}
@end
```

> **Get This Recipe's Code**
>
> To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from the book, go to the folder for Chapter 6 and open the project for this recipe.

Recipe: Naming Views

Although tagging offers a thorough approach to identifying views, some developers may prefer to work with names rather than numbers. Using names adds an extra level of meaning to your view identification schemes. Instead of referring to "the view with a tag of 101," a switch named "Ignition Switch" describes its role and adds a level of self-documentation missing from a plain number.

From the Library of Eric Berna

```
// Toggle switch
UISwitch *s = [self.view switchNamed:@"Ignition Switch"];
[s setOn:!s.isOn];
```

It's relatively easy to design a class that associates strings with view tags. There are two ways to accomplish this. The first is to design a custom class that stores a dictionary that matches names with tags, allowing views to register and unregister those names. The second is to use Objective-C's runtime-associated object functions.

Associated Objects

Although associated objects are cleaner and easier to use, they are just barely supported in iOS 5. You must, at the time this book is being written, manually declare both constants and functions when linking to the Foundation framework. If this kind of code work bothers you, you're probably better off using the dictionary-based recipe that follows this section.

```
enum {
    OBJC_ASSOCIATION_ASSIGN = 0,
    OBJC_ASSOCIATION_RETAIN_NONATOMIC = 1,
    OBJC_ASSOCIATION_COPY_NONATOMIC = 3,
    OBJC_ASSOCIATION_RETAIN = 01401,
    OBJC_ASSOCIATION_COPY = 01403
};

typedef uintptr_t objc_AssociationPolicy;
id objc_getAssociatedObject(id object, void *key);
void objc_setAssociatedObject(id object, void *key, id value,
objc_AssociationPolicy policy);
void objc_removeAssociatedObjects(id object);
```

You can declare a custom `nametag` property to leverage associated objects. This property does not create new storage in a `UIView`; it adds two methods (a setter and a getter) that will, instead, call on the runtime functions.

```
@interface UIView (Nametags)
@property (nonatomic, strong) NSString *nametag;
- (UIView *) viewWithNametag: (NSString *) aName;
@end
```

The implementation is simple. Setting and retrieving the associated `nametag` involves calling two Objective-C runtime functions.

```
static const char *NametagKey = "Nametag Key";


- (id) nametag
{
    return objc_getAssociatedObject(self, (void *) NametagKey);
}

- (void)setNametag:(NSString *) theNametag
```

From the Library of Eric Berna

```
{
    objc_setAssociatedObject(self, (void *) NametagKey,
        theNametag, OBJC_ASSOCIATION_RETAIN_NONATOMIC);
}
```

To retrieve a view by its nametag, perform a basic depth-first search. This `viewWithNameTag:` method returns the first view whose name tag is equal to a given string. If no views match, the method returns `nil`.

```
- (UIView *) viewWithNametag: (NSString *) aName
{
    if (!aName) return nil;

    // Is this the right view?
    if ([self.nametag isEqualToString:aName])
        return self;

    // Recurse depth first on subviews
    for (UIView *subview in self.subviews)
    {
        UIView *resultView = [subview viewNamed:aName];
        if (resultView) return resultView;
    }

    // Not found
    return nil;
}
```

Using a Name Dictionary

Recipe 6-4 shows how to build that view name manager, which uses a singleton instance (`[ViewIndexer sharedInstance]`) to store its tag and name dictionary.

The class demands unique names. If a view name is already registered, a new registration request will fail. If a view was already registered under another name, a second registration request will unregister the first name. There are ways to fool this of course. If you change a view's tag and then register it again, the indexer has no way of knowing that the view had been previously registered. So if you decide to use this approach, set your tags in Interface Builder or let the registration process automatically tag the view but otherwise leave the tags be.

If you build your views by hand, register them at the same point you create them and add them into your overall view hierarchy. When using an IB-defined view, register your names in `viewDidLaoad` using the tag numbers you set in the attributes inspector.

```
- (void) viewDidLoad
{
    [[self.view viewWithTag:LABEL_TAG] registerName:@"my label"];
    [[self.view viewWithTag:SWITCH_TAG] registerName:@"my switch"];
}
```

From the Library of Eric Berna

Recipe 6-4 hides the view indexer class from public view. It wraps its calls inside a `UIView` category for name extensions. This allows you to register, retrieve, and unregister views without using `ViewIndexer` directly. For reasons of space, the recipe omits typed name retrievals such as `labelNamed:` and `textFieldNamed:`, but these are included in the sample code for the chapter.

Recipe 6-4 **Creating a View Name Manager**

```
@interface ViewIndexer : NSObject {
    NSMutableDictionary *tagdict;
    NSInteger count;
}
@end

@implementation ViewIndexer
static ViewIndexer *sharedInstance = nil;

+(ViewIndexer *) sharedInstance {
    if(!sharedInstance) sharedInstance = [[self alloc] init];
    return sharedInstance;
}

- (id) init
{
    if (!(self = [super init])) return self;
    tagdict = [NSMutableDictionary dictionary];
    count = 10000;
    return self;
}

// Pull a new number and increase the count
- (NSInteger) pullNumber
{
    return count++;
}

// Check to see if name exists in dictionary
- (BOOL) nameExists: (NSString *) aName
{
    return [tagdict objectForKey:aName] != nil;
}

// Pull out first matching name for tag
- (NSString *) nameForTag: (NSInteger) aTag
{
    NSNumber *tag = [NSNumber numberWithInt:aTag];
```

From the Library of Eric Berna

```
    NSArray *names = [tagdict allKeysForObject:tag];
    if (!names) return nil;
    if ([names count] == 0) return nil;
    return [names objectAtIndex:0];
}

// Return the tag for a registered name. 0 if not found
- (NSInteger) tagForName: (NSString *)aName
{
    NSNumber *tag = [tagdict objectForKey:aName];
    if (!tag) return 0;
    return [tag intValue];
}

// Unregistering reverts tag to 0
- (BOOL) unregisterName: (NSString *) aName forView: (UIView *) aView
{
    NSNumber *tag = [self.tagdict objectForKey:aName];

    // tag not found
    if (!tag) return NO;

    // tag does not match registered name
    if (aView.tag != [tag intValue]) return NO;

    aView.tag = 0;
    [tagdict removeObjectForKey:aName];
    return YES;
}

// Register a new name. Names will not re-register. (Unregister first,
// please). If a view is already registered, it is unregistered and
// re-registered
- (NSInteger) registerName:(NSString *)aName forView: (UIView *) aView
{
    // You cannot re-register an existing name
    if ([[ViewIndexer sharedInstance] nameExists:aName]) return 0;

    // Check to see if the view is named already. If so, unregister.
    NSString *currentName = [self nameForTag:aView.tag];
    if (currentName) [self unregisterName:currentName forView:aView];

    // Register the existing tag or pull a new tag if aView.tag is 0
    if (!aView.tag) aView.tag = [[ViewIndexer sharedInstance]
        pullNumber];
    [tagdict
        setObject:[NSNumber numberWithInt:aView.tag]
```

From the Library of Eric Berna

```
        forKey: aName];
    return aView.tag;
}
@end

@implementation UIView (NameExtensions)

- (NSInteger) registerName: (NSString *) aName
{
    return [[ViewIndexer sharedInstance] registerName: aName
        forView: self];
}

- (BOOL) unregisterName: (NSString *) aName
{
    return [[ViewIndexer sharedInstance] unregisterName: aName
        forView:self];
}

- (UIView *) viewNamed: (NSString *) aName
{
    NSInteger tag = [[ViewIndexer sharedInstance] tagForName: aName];
    return [self viewWithTag: tag];
}
@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 6 and open the project for this recipe.

View Geometry

As you'd expect, geometry plays an important role when working with views. Geometry defines where each view appears onscreen, what its size is, and how it is oriented. The `UIView` class provides two built-in properties that define these aspects.

Every view uses a `frame` to define its boundaries. The frame specifies the outline of the view: its location, width, and height. If you change a view's frame, the view updates to match the new frame. Use a bigger width and the view stretches. Use a new location and the view moves. The view's frame delineates each view's onscreen outline. View sizes are not limited to the screen size. A view can be smaller than the screen or larger. It can also be smaller or larger than its parent.

Views also use a `transform` property that sets the view's orientation and any geometric transformations that have been applied to it. For example, a view might be stretched or

From the Library of Eric Berna

squashed by applying a transform, or it might be rotated away from vertical. Together the frame and transform fully define a view's geometry.

Frames

Frame rectangles use a `CGRect` structure, which is defined as part of the Core Graphics framework as its "CG" prefix suggests. A `CGRect` is made up of an origin (a `CGPoint`, x and y) and a size (a `CGSize`, width and height). When you create views, you normally allocate them and initialize them with a frame. Here's an example:

```
CGRect rect = CGRectMake(0.0f, 0.0f, 320.0f, 416.0f);
myView = [[UIView alloc] initWithFrame: rect];
```

The `CGRectMake` function creates a new rectangle using four parameters: the origin's x and y locations, the width of the rectangle, and its height. In addition to `CGRectMake`, you may want to be aware of several other convenience functions that help you work with rectangles and frames:

- `NSStringFromCGRect(aCGRect)` converts a `CGRect` structure to a formatted string. This function makes it easy to log a view's frame when you're debugging.
- `CGRectFromString(aString)` recovers a rectangle from its string representation. It proves useful when you've stored a view's frame as a string in user defaults and want to convert that string back to a `CGRect`.
- `CGRectInset(aRect, xinset, yinset)` enables you to create a smaller or larger rectangle that's centered on the same point as the source rectangle. Use a positive inset for smaller rectangles, negative for larger ones.
- `CGRectOffset(aRect, xoffset, yoffset)` returns a rectangle that's offset from the original rectangle by an x and y amount you specify. This is handy for moving frames around the screen and for creating easy drop-shadow effects with a view's sublayers.
- `CGRectGetMidX(aRect)` and `CGRectGetMidY(aRect)` recover the x and y coordinates in the center of a rectangle. These functions make it very convenient to recover the mid points of bounds and frames.
- `CGRectIntersectsRect(rect1, rect2)` lets you know whether rectangle structures intersect. Use this function to know when two rectangular onscreen objects overlap.
- `CGRectZero` is a rectangle constant located at `(0,0)` whose width and height are zero. You can use this constant when you're required to create a frame but are unsure what that frame size or location will be at the time of creation.

The `CGRect` structure is made up of two substructures: `CGPoint`, which defines the rectangle's origin, and `CGSize`, which defines its bounds. Points refer to locations defined with x and y coordinates; sizes have width and height. Use `CGPointMake(x, y)` to create points. `CGSizeMake(width, height)` creates sizes. Although these two structures appear

From the Library of Eric Berna

to be the same (two floating-point values), the iOS SDK differentiates between them. Points refer to locations. Sizes refer to extents. You cannot set `myFrame.origin` to a size.

As with rectangles, you can convert them to and from strings: `NSStringFromCGPoint()`, `NSStringFromCGSize()`, `CGSizeFromString()`, and `CGPointFromString()` perform these functions. You can also transform points and sizes to and from dictionaries.

Transforms

The iOS SDK supports standard affine transformations as part of its Core Graphics implementation. Affine transforms allow points in one coordinate system to transform into another coordinate system. These functions are widely used in both 2D and 3D animations. The version used in the iOS SDK uses a 3-by-3 matrix to define `UIView` transforms, making it a 2D-only solution. 3D transforms use a 4-by-4 matrix, and are the default for Core Animation layers. With affine transforms, you can scale, translate, and rotate your views in real time. You do so by setting the view's `transform` property. Here's an example:

```
float angle = theta * (PI / 100);
CGAffineTransform transform = CGAffineTransformMakeRotation(angle);
myView.transform = transform;
```

The transform is always applied with respect to the view's center. So when you apply a rotation like this, the view rotates around its center. If you need to rotate around another point, you must first translate the view, then rotate, and then return from that translation.

To revert any changes, set the `transform` property to the identity transform. This restores the view back to the last settings for its frame.

```
myView.transform = CGAffineTransformIdentity;
```

> **Note**
>
> On iOS, the y coordinate starts at the top and increases downward. This is similar to the coordinate system in PostScript but opposite the Quartz coordinate system historically used on the Mac. On iOS, the origin is in the top-left corner, not the bottom-left.

Coordinate Systems

Views live in two worlds. Their frames are defined in the coordinate system of their parents. Their bounds and subviews are defined in their own coordinate system. The iOS SDK offers several utilities that allow you to move between these coordinate systems so long as the views involved live within the same `UIWindow`. To convert a point from another view into your own coordinate system, use `convertPoint:fromView:`. Here's an example:

```
myPoint = [myView convertPoint:somePoint fromView:otherView];
```

If the original point indicated the location of some object, the new point retains that location but gives the coordinates with respect to `myView`'s origin. To go the other way, use

From the Library of Eric Berna

`convertPoint:toView:` to transform a point into another view's coordinate system. Similarly, `convertRect:toView:` and `convertRect:fromView:` work with `CGRect` structures rather than `CGPoint` ones.

Be aware that the coordinate system for an iOS device may not match the pixel system used to display that system. The discrete 640-by-960-pixel Retina display on the iPhone 4 and fourth-generation iPod touch, for example, is addressed through a continuous 320-by-480 coordinate system in the SDK. Although you can supply higher quality art to fill those pixels on newer units, any locations you specify in your code access the same coordinate system used for older, lower pixel-density units. The position (160.0, 240.0) remains approximately in the center of the iPhone or iPod touch screen.

Recipe: Working with View Frames

When you change a view's frame, you update its size (that is, its width and height) and its location. For example, you might move a frame as follows. This code creates a subview located at (0.0, 0.0) and then moves it down to (0.0, 30.0):

```
CGRect initialRect = CGRectMake(0.0f, 0.0f, 320.0f, 50.0f);
myView = [[UIView alloc] initWithFrame:initialRect];
[topView addSubview:myView];
myView.frame = CGRectMake(0.0f, 30.0f, 320.0f, 50.0f);
```

This approach is fairly uncommon. The iOS SDK does not expect you to move a view by changing its frame. Instead, it provides you with a way to update a view's position. The preferred way to do this is by setting the view's center. `center` is a built-in view property, which you can access directly:

```
myView.center = CGPointMake(160.0f, 55.0f);
```

Although you'd expect the SDK to offer a way to move a view by updating its origin, no such option exists. It's easy enough to build your own class extension. Retrieve the view frame, set the origin to the requested point, and then update the frame with the change. This snippet creates a new origin property letting you retrieve and change the view's origin:

```
@interface UIView (ViewFrameGeometry)
@property CGPoint origin;
@end

@implementation UIView (ViewFrameGeometry)
- (CGPoint) origin
{
    return self.frame.origin;
}

- (void) setOrigin: (CGPoint) aPoint
{
    CGRect newframe = self.frame;
```

From the Library of Eric Berna

```
    newframe.origin = aPoint;
    self.frame = newframe;
}
@end
```

Because this extension uses such an obvious property name, if Apple eventually implements the features shown here, your code may break due to name overlap. In my examples in this book, I widely use obvious names. Avoid doing so in your production code. Using your personal or company initials as a prefix helps distinguish in-house material.

When you move a view, you don't need to worry about things such as rectangular sections that have been exposed or hidden. iOS takes care of the redrawing. This lets you treat your views like tangible objects and delegate rendering issues to Cocoa Touch.

Adjusting Sizes

In the simplest usage patterns, a view's frame and bounds control its size. Frames, as you've already seen, define the location of a view in its parent's coordinate system. If the frame's origin is set to (0.0, 30.0), the view appears in the superview flush with the left side of the view and offset 30 coordinate units from the top. On older displays, this corresponds to 30 pixels down; on Retina displays, it is 60 pixels down. Bounds define a view within its own coordinate system. That means the origin for a view's bounds (that is, `myView.bounds`) is always (0.0,0.0), and its size matches its normal extent (that is, the frame's `size` property).

You can change a view's size onscreen by adjusting either its frame or its bounds. In practical terms, you're updating the size component of those structures. As with moving origins, it's simple to create your own utility method to do this directly:

```
- (void) setSize: (CGSize) aSize
{
    CGRect newframe = self.frame;
    newframe.size = aSize;
    self.frame = newframe;
}
```

When a view's size changes, the view itself updates live onscreen. Depending on how the elements within the view are defined and the class of the view itself, subviews may shrink to fit or they may get cropped. There's no single rule that covers all circumstances. Xcode's Interface Builder's size inspector offers interactive resizing options that define how subviews respond to changes in a superview's frame. See Chapter 4, "Designing Interfaces," for more details about laying out items in Interface Builder in Xcode.

> **Note**
>
> Bounds are affected by a view's transform, a mathematical component that changes the way the view appears onscreen. Do not manipulate a view's frame when working with transforms because it may not produce expected results. For example, after a transform, the frame's origin may no longer correspond to (0,0) of the bounds. The normal order of updating a view is to set its frame or bounds, then set its center, and then set its transforms if applicable.

From the Library of Eric Berna

Sometimes, you need to resize a view before adding it to a new parent. For example, you might have an image view to place into an alert view. To fit that view into place without changing its aspect ratio, you might use a method like this to ensure that both the height and width scale appropriately:

```
- (void) fitInSize: (CGSize) aSize
{
    CGFloat scale;
    CGRect newframe = self.frame;

    if (newframe.size.height > aSize.height)
    {
        scale = aSize.height / newframe.size.height;
        newframe.size.width *= scale;
        newframe.size.height *= scale;
    }

    if (newframe.size.width >= aSize.width)
    {
        scale = aSize.width / newframe.size.width;
        newframe.size.width *= scale;
        newframe.size.height *= scale;
    }

    self.frame = newframe;
}
```

`CGRects` and Centers

As you've seen, `UIView` instances use `CGRect` structures composed of an origin and a size to define their frames. This structure contains no references to a center point. At the same time, `UIViews` depend on their `center` property to update a view's position when you move a view to a new point. Unfortunately, Core Graphics doesn't use centers as a primary rectangle concept. As far as centers are concerned, Core Graphics' built-in utilities are limited to recovering a rectangle's midpoint along the x- or y-axis.

You can bridge this gap by constructing functions that coordinate between the origin-based `CGRect` struct and center-based `UIView` objects. This function retrieves the center from a rectangle by building a point from the x and y midpoints. It takes one argument, a rectangle, and returns its center point:

```
CGPoint CGRectGetCenter(CGRect rect)
{
    CGPoint pt;
    pt.x = CGRectGetMidX(rect);
    pt.y = CGRectGetMidY(rect);
    return pt;
}
```

From the Library of Eric Berna

Moving a rectangle by its center point is another function that may prove helpful, and one that mimics the way `UIViews` work. Say you need to move a view to a new position but need to keep it inside its parent's frame. To test before you move, you'd use a function like this to offset the view frame to a new center. You could then test that offset frame against the parent (use `CGRectContainsRect()`) and ensure that the view won't stray outside its container.

```
CGRect CGRectMoveToCenter(CGRect rect, CGPoint center)
{
    CGRect newrect = CGRectZero;
    newrect.origin.x = center.x-CGRectGetMidX(rect);
    newrect.origin.y = center.y-CGRectGetMidY(rect);
    newrect.size = rect.size;
    return newrect;
}
```

Often you need to center one view in another. Here's how you can retrieve a frame that corresponds to a centered subrectangle. Pass the outer view's bounds when adding a subview (the subview coordinate system needs to start with 0,0), or its frame when adding a view to the outer view's parent:

```
CGRect CGRectCenteredInRect(CGRect rect, CGRect mainRect)
{
    CGFloat xOffset = CGRectGetMidX(mainRect)-CGRectGetMidX(rect);
    CGFloat yOffset = CGRectGetMidY(mainRect)-CGRectGetMidY(rect);
    return CGRectOffset(rect, xOffset, yOffset);
}
```

Other Utility Methods

As you've seen, it's convenient to expose a view's origin and size in parallel to its center, allowing you to work more natively with Core Graphics calls. You can build on this idea to expose other properties of the view, including its `width` and `height`, as well as basic geometry such as its `left`, `right`, `top`, and `bottom` points.

In some ways, this breaks Apple's design philosophy. This exposes items that normally fall into structures without reflecting the structures. At the same time, it can be argued that these elements are true view properties. They reflect fundamental view characteristics and deserve to be exposed as properties.

Recipe 6-5 provides a full view frame utility category for `UIView`, letting you make the choice of whether to use these properties. These properties do not take transforms into account and are meant for simple views that do not use affine transforms.

Recipe 6-5 **`UIView` Frame Geometry Category**

```
@interface UIView (ViewFrameGeometry)
@property CGPoint origin;
@property CGSize size;
```

From the Library of Eric Berna

```
@property (readonly) CGPoint midpoint;

// topLeft is synonymous with origin, so not included here
@property (readonly) CGPoint bottomLeft;
@property (readonly) CGPoint bottomRight;
@property (readonly) CGPoint topRight;

@property CGFloat height;
@property CGFloat width;
@property CGFloat top;
@property CGFloat left;
@property CGFloat bottom;
@property CGFloat right;

- (void) moveBy: (CGPoint) delta;
- (void) scaleBy: (CGFloat) scaleFactor;
- (void) fitInSize: (CGSize) aSize;
@end

@implementation UIView (ViewGeometry)
// Retrieve and set the origin
- (CGPoint) origin
{
    return self.frame.origin;
}

- (void) setOrigin: (CGPoint) aPoint
{
    CGRect newframe = self.frame;
    newframe.origin = aPoint;
    self.frame = newframe;
}

// Retrieve and set the size
- (CGSize) size
{
    return self.frame.size;
}

- (void) setSize: (CGSize) aSize
{
    CGRect newframe = self.frame;
    newframe.size = aSize;
    self.frame = newframe;
}
```

From the Library of Eric Berna

```
// Query other frame locations

- (CGPoint) midpoint
{
    // midpoint is with respect to a view's own coordinate system
    // versus its center, which is with respect to its parent
    CGFloat x = CGRectGetMidX(self.bounds);
    CGFloat y = CGRectGetMidY(self.bounds);
    return CGPointMake(x, y);
}

- (CGPoint) bottomRight
{
    CGFloat x = self.frame.origin.x + self.frame.size.width;
    CGFloat y = self.frame.origin.y + self.frame.size.height;
    return CGPointMake(x, y);
}

- (CGPoint) bottomLeft
{
    CGFloat x = self.frame.origin.x;
    CGFloat y = self.frame.origin.y + self.frame.size.height;
    return CGPointMake(x, y);
}

- (CGPoint) topRight
{
    CGFloat x = self.frame.origin.x + self.frame.size.width;
    CGFloat y = self.frame.origin.y;
    return CGPointMake(x, y);
}


// Retrieve and set height, width, top, bottom, left, right
- (CGFloat) height
{
    return self.frame.size.height;
}

- (void) setHeight: (CGFloat) newheight
{
    CGRect newframe = self.frame;
    newframe.size.height = newheight;
    self.frame = newframe;
}
```

From the Library of Eric Berna

```
- (CGFloat) width
{
    return self.frame.size.width;
}

- (void) setWidth: (CGFloat) newwidth
{
    CGRect newframe = self.frame;
    newframe.size.width = newwidth;
    self.frame = newframe;
}

- (CGFloat) top
{
    return self.frame.origin.y;
}

- (void) setTop: (CGFloat) newtop
{
    CGRect newframe = self.frame;
    newframe.origin.y = newtop;
    self.frame = newframe;
}

- (CGFloat) left
{
    return self.frame.origin.x;
}

- (void) setLeft: (CGFloat) newleft
{
    CGRect newframe = self.frame;
    newframe.origin.x = newleft;
    self.frame = newframe;
}

- (CGFloat) bottom
{
    return self.frame.origin.y + self.frame.size.height;
}

- (void) setBottom: (CGFloat) newbottom
{
    CGRect newframe = self.frame;
    newframe.origin.y = newbottom - self.frame.size.height;
    self.frame = newframe;
}
```

From the Library of Eric Berna

```
- (CGFloat) right
{
    return self.frame.origin.x + self.frame.size.width;
}

- (void) setRight: (CGFloat) newright
{
    CGFloat delta = newright -
        (self.frame.origin.x + self.frame.size.width);
    CGRect newframe = self.frame;
    newframe.origin.x += delta;
    self.frame = newframe;
}
@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 6 and open the project for this recipe.

Recipe: Randomly Moving a Bounded View

When you move a view to a random point, you must take into account several things. Often a view must fit entirely within its parent's view container so that no parts of the view are clipped off. You may also want to add a boundary to that container so the view does not quite touch the parent's edge at any time. Finally, if you're working with out-of-the-box SDK versions of the `UIView` class, you need to work with random centers, not random positions, as discussed earlier in this chapter. Just picking a point somewhere in the parent view fails some or all these qualifications.

Recipe 6-6 approaches this problem by creating a series of insets. It uses the `UIEdgeInset` structure to define the boundaries for the view. This structure contains four inset values, corresponding to the amount to inset a rectangle at its top, left, bottom, and right:

```
typedef struct {
    CGFloat top, left, bottom, right;
} UIEdgeInsets;
```

This method uses the `UIEdgeInsetsInsetRect()` function to narrow a `CGRect` rectangle to create an inner container, which is called `innerRect` in this method.

It then narrows the container even further. It insets that rectangle by half the child's height and width. This leaves enough room around any point in the subrectangle to allow the placement of the child view, guaranteeing that the view can do so without overlapping the inner bounded rectangle. Select any point in that subrectangle to return a valid center for the child view.

From the Library of Eric Berna

Recipe 6-6 **Randomly Moving a Bounded View**

```
- (CGPoint) randomCenterInView: (UIView *) aView
    withInsets: (UIEdgeInsets) insets
{
    // Move in by the inset amount and then by size of the subview
    CGRect innerRect = UIEdgeInsetsInsetRect([aView bounds], insets);
    CGRect subRect = CGRectInset(innerRect,
        self.frame.size.width / 2.0f, self.frame.size.height / 2.0f);

    // Return a random point
    float rx = (float)(random() % (int)floor(subRect.size.width));
    float ry = (float)(random() % (int)floor(subRect.size.height));
    return CGPointMake(rx + subRect.origin.x, ry + subRect.origin.y);
}

- (CGPoint) randomCenterInView: (UIView *) aView
        withInset: (float) inset
{
    UIEdgeInsets insets = UIEdgeInsetsMake(inset, inset, inset, inset);
    return [self randomCenterInView:aView withInsets:insets];
}

- (void) moveToRandomLocationInView: (UIView *) aView {
    self.center = [self randomCenterInView:aView withInset:5];
    return;
}
```

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 6 and open the project for this recipe.

Recipe: Transforming Views

Affine transforms enable you to change an object's geometry by mapping that object from one view coordinate system into another. The iOS SDK fully supports standard affine 2D transforms. With them, you can scale, translate, rotate, and skew your views however your heart desires and your application demands.

Transforms are defined in Core Graphics and consist of calls such as `CGAffineTransformMakeRotation` and `CGAffineTransformScale`. These build and modify the 3-by-3 transform matrices. Once these are built, use `UIView`'s `setTransform` call to apply 2D affine transformations to `UIView` objects.

Recipe 6-7 demonstrates how to build and apply an affine transform of a `UIView`. To create the example, I kept things simple. I built an `NSTimer` that ticks every 1/30th of a second. On ticking, it rotates a view by 1% of pi and scales over a cosine curve. I use the

From the Library of Eric Berna

cosine's absolute value for two reasons. It keeps the view visible at all times, and it provides a nice bounce effect when the scaling changes direction. This produces a rotating bounce animation.

This is one of those examples that it's best to build and view as you read through the code. You are better able to see how the `handleTimer:` method correlates to the visual effects you're looking at.

> **Note**
>
> This recipe uses the standard C math library, which provides both the `cosine` function and the `M_PI` constant.

Recipe 6-7 **Example of an Affine Transform of a `UIView`**

```
- (void) move: (NSTimer *) aTimer
{
    // Rotate each iteration by 1% of PI
    CGFloat angle = theta * (M_PI / 100.0f);
    CGAffineTransform transform = CGAffineTransformMakeRotation(angle);

    // Theta ranges between 0% and 199% of PI, i.e. between 0 and 2*PI
    theta = (theta + 1) % 200;

    // For fun, scale by the absolute value of the cosine
    float degree = cos(angle);
    if (degree < 0.0) degree *= -1.0f;
    degree += 0.5f;

    // Create add scaling to the rotation transform
    CGAffineTransform scaled =
        CGAffineTransformScale(transform, degree, degree);

    // Apply the affine transform
    imageView.transform = scaled;
}
```

> **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 6 and open the project for this recipe.

Display and Interaction Traits

In addition to physical screen layout, the `UIView` class provides properties that control how your view appears onscreen and whether users can interact with it. Every view uses a translucency factor (`alpha`) that ranges between opaque and transparent. Adjust this by

From the Library of Eric Berna

issuing `[myView setAlpha:value]` or setting the `myView.alpha` property where the `alpha` values fall between 0.0 (fully transparent) and 1.0 (fully opaque). This is a great way to hide views and to fade them in and out onscreen.

You can assign a color to the background of any view. For example, the following property colors your view red:

```
myView.backgroundColor = [UIColor redColor]
```

This property affects different view classes in different ways, depending on whether those views contain subviews that block the background. Create a transparent background by setting the view's background color to clear, as shown here:

```
myView.backgroundColor = [UIColor clearColor];
```

Every view offers a background color property regardless of whether you can see the background. Using bright, contrasting background colors is a great way to visualize the true extents of views. When you're new to iOS development, coloring in views provides you a concrete sense of what is and is not onscreen and where each component is located.

The `userInteractionEnabled` property controls whether users can touch and interact with a given view. For most views, this property defaults to `YES`. For `UIImageView`, it defaults to `NO`, which can cause a lot of grief among beginning developers. They often place a `UIImageView` as their backsplash and don't understand why their switches, text entry fields, and buttons do not work. Make sure to enable the property for any view that needs to accept touches, whether for itself or for its subviews, which may include buttons, switches, pickers, and other controls. If you're experiencing trouble with items that seem unresponsive to touch, you should check the `userInteractionEnabled` property value for that item and for its parents.

Disable this property for any display-only view you layer over your interaction area. To show a noninteractive overlay clock, for example, via a transparent full-screen view, unset its interaction. This allows touches to pass through the view and fall below to the actual interaction area of your application. To create a please-wait-style blocker, on the other hand, make sure to enable user interaction for your overlay. This catches user taps and prevents users from accessing your primary interface behind that overlay.

You may also want to disable the property during transitions, to ensure that user taps do not trigger actions as views are being animated. Unwanted touches can be a problem particularly for games and puzzles.

`UIView` Animations

`UIView` animation provides one of the odd but lovely perks of working with iOS as a development platform. It enables you to create a moving expression of visual changes when updating views, producing smooth animated results that enhance the user experience. Best of all, this all occurs without you having to do much work.

From the Library of Eric Berna

`UIView` animations are perfect for building a visual bridge between a view's current and changed states. With them, you emphasize visual change and create an animation that links those changes together. Animatable changes include the following:

- **Changes in location**—Moving a view around the screen
- **Changes in size**—Updating the view's frame and bounds
- **Changes in stretching**—Updating the view's content stretch regions
- **Changes in transparency**—Altering the view's alpha value
- **Changes in states**—Hidden versus showing
- **Changes in view order**—Altering which view is in front
- **Changes in rotation**—Or any other affine transforms you apply to a view

Animations underwent a profound redesign between the 3.*x* and 4.*x* SDKs. Starting with the 4.*x* SDK, developers were offered a way to use the new Objective-C blocks paradigm to simplify animation tasks. Although you can still work with the original animation transaction techniques, the new alternatives provide a much easier approach. The next sections review the original approach and introduce the enhancements.

Building `UIView` Animation Transactions

In their older nonblock form, `UIView` animations work as transactions; they are a series of operations that progress at once. To create a transaction, start by issuing `beginAnimations:context:`. End with `commitAnimations`. You can also set animation options for the transaction. Send these class methods to `UIView` and not to individual views. Between the begin and end calls, you define the way the animation works and perform the actual view updates. The animation controls you use are as follows:

- **`beginAnimations:context`**—Marks the start of the animation transaction.
- **`setAnimationCurve`**—Defines the way the animation accelerates and decelerates. Use ease-in/ease-out (`UIViewAnimationCurveEaseInOut`) unless you have some compelling reason to select another curve. The other curve types are ease-in (accelerate into the animation), linear (no animation acceleration), and ease-out (accelerate out of the animation). Ease-in/ease-out provides the most natural-feeling animation style.
- **`setAnimationDuration`**—Specifies the length of the animation, in seconds. This is really the cool bit. You can stretch out the animation for as long as you need it to run. Be aware of straining your user's patience and keep your animations below a second or two in length. As a point of reference, the keyboard animation, when it slides on or offscreen, lasts 0.3 seconds.
- **`commitAnimations`**—Marks the end of the animation transaction.

Sandwich your actual view change commands after setting up the animation details and before ending the animation:

From the Library of Eric Berna

```
CGContextRef context = UIGraphicsGetCurrentContext();
[UIView beginAnimations:nil context:context];
[UIView setAnimationCurve:UIViewAnimationCurveEaseInOut];
[UIView setAnimationDuration:1.0f];

// View changes go here
[contentView setAlpha:0.0f];

[UIView commitAnimations];
```

This snippet shows `UIView` animations in action by setting an animation curve and the animation duration (here, 1 second). The actual change being animated is a transparency update. The alpha value of the content view goes to zero, turning it invisible. Instead of the view simply disappearing, this animation block slows down the change and fades it out of sight. Notice the call to `UIGraphicsGetCurrentContext()`, which returns the graphics context at the top of the current view stack. A graphics context provides a virtual connection between your abstract drawing calls and the actual pixels on your screen (or within an image). As a rule, you can pass `nil` for this argument without ill effect in newer SDKs.

Transaction-based view animations can notify an optional delegate about state changes—namely that an animation has started or ended. (As you'll see in just a bit, it's far easier to do the same with blocks.) This allows you to catch the end of an animation to start the next animation in a sequence or otherwise perform post-animation tasks. To set the delegate, use `setAnimationDelegate:`. Here's an example:

```
[UIView setAnimationDelegate:self];
```

To set up an end-of-animation callback, supply the selector sent to the delegate:

```
[UIView setAnimationDidStopSelector:@selector(animationDidStop:finished:context:)];
```

When the animation completes, the delegate is sent the selector in question and can then execute any code related to the animation's endpoint.

> **Note**
>
> Most Apple-native animations last about a third or a half a second. When working with onscreen helper views (playing supporting roles that are similar to Apple's keyboard or alerts), you may want to match your animation durations to these elements.

Building Animations with Blocks

Blocks constructs simplify creating basic animation effects in your code. Consider the following snippet. It produces the same fade-out as the previous snippet but does so with a much more intuitive structure:

```
[UIView animationWithDuration: 1.0f
    animations:^{contentView.alpha = 0.0f;}];
```

Instead of six statements, the same effect is achieved with one statement with an embedded block. The animation curve defaults to ease-in/ease-out.

From the Library of Eric Berna

Adding a completion block also proves much simpler. The following snippet fades out the content view and then removes it from its superview upon completing the animation:

```
[UIView animationWithDuration: 1.0f
    animations:^{contentView.alpha = 0.0f;}
    completion:^(BOOL done){[contentView removeFromSuperview];}];
```

Once you've moved from the old approach into the blocks approach, you'll recognize the simple elegance of the newer implementations. Should you need to add further options to your animations, a full-service blocks-based method (`animateWithDuration:delay:options:animations:completion:`) provides both a way to pass animation options (as a mask) and to delay the animation (allowing a simple approach to animation "chaining").

Conditional Animation

I continue to use old-style transactions rather than blocks in those rare times when I deal with animation conditionality—that is, when the animation itself is optional. Here's an example of where blocks would be less effective than old-style `UIView` animation:

```
if (animated)
{
    [UIView beginAnimations:@"SwitchAnimation" context:nil];
    [UIView setAnimationCurve:UIViewAnimationCurveEaseInOut];
    [UIView setAnimationDuration:0.1f];
}

if (self.on)
    rect.origin = CGPointMake(0.0f, 0.0f);
else
    rect.origin = CGPointMake(newX, 0.0f);
backdrop.frame = rect;

if (animated) [UIView commitAnimations];
```

As you can see here, although it's possible to create an approach that checks for animation and either calls an animation block or executes the code separately, it's much easier to add `if` statements around the older-style animation calls.

Recipe: Fading a View In and Out

At times, you want to add information to your screen that overlays your view but does not of itself do anything. For example, you might show a top-scores list or some instructions or provide a context-sensitive tooltip. Recipe 6-8 demonstrates how to use a `UIView` animation block to fade a view into and out of sight. This recipe follows the most basic animation approach. It creates a view animation block that sets the `alpha` property.

Note how this code controls the behavior of the right bar button item. When tapped, it is immediately disabled until the animation concludes. The animation's completion block

From the Library of Eric Berna

reenables the button and flips the button text and callback selector to the opposite state. This allows the button to toggle the animation from on to off, and from off to back on.

Recipe 6-8 **Animating Transparency Changes to a View's `Alpha` Property**

```
- (void) fadeOut: (id) sender
{
    self.navigationItem.rightBarButtonItem.enabled = NO;
    [UIView animateWithDuration:1.0f
              animations:^{
                  // Here's where the actual fade out takes place
                  imageView.alpha = 0.0f;
              }
              completion:^(BOOL done){
                  self.navigationItem.rightBarButtonItem.enabled = YES;
                  self.navigationItem.rightBarButtonItem =
                      BARBUTTON(@"Fade In", @selector(fadeIn:));
              }];
}


- (void) fadeIn: (id) sender
{
    self.navigationItem.rightBarButtonItem.enabled = NO;
    [UIView animateWithDuration:1.0f
              animations:^{
                 // Here's where the fade in occurs
                 imageView.alpha = 1.0f;
              }
              completion:^(BOOL done){
                  self.navigationItem.rightBarButtonItem.enabled = YES;
                  self.navigationItem.rightBarButtonItem =
                      BARBUTTON(@"Fade Out", @selector(fadeOut:));
              }];
}

- (void) viewDidAppear: (BOOL) animated
{
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Fade Out", @selector(fadeOut:));

    imageView = [[UIImageView alloc] initWithImage:
        [UIImage imageNamed:@"BFlyCircle.png"]];
    [self.view addSubview:imageView];
}
```

From the Library of Eric Berna

> **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 6 and open the project for this recipe.

Recipe: Swapping Views

The `UIView` animation block doesn't limit you to a single change. You can place as many animation differences as needed in the animations block. Recipe 6-9 combines size transformations with transparency changes to create a more compelling animation. It does this by adding several directives at once to the animation block. This recipe performs five actions at a time. It zooms and fades one view into place while zooming out and fading away another and then exchanges the two in the subview array list.

You'll want to prepare the back object for animation by shrinking it and making it transparent. When the `swap:` method first executes, that view will be ready to appear and zoom to size. As with Recipe 6-8, the completion block reenables the right-hand bar button, allowing successive presses.

Recipe 6-9 **Combining Multiple View Changes in Animation Blocks**

```
@implementation TestBedViewController
- (void) swap: (id) sender
{
    self.navigationItem.rightBarButtonItem.enabled = NO;
    [UIView animateWithDuration:1.0f
      animations:^{
        frontObject.alpha = 0.0f;
        backObject.alpha = 1.0f;
        frontObject.transform = CGAffineTransformMakeScale(0.25f, 0.25f);
        backObject.transform = CGAffineTransformIdentity;
        [self.view exchangeSubviewAtIndex:0
            withSubviewAtIndex:1];
      }
      completion:^(BOOL done){
        self.navigationItem.rightBarButtonItem.enabled = YES;

        // Swap the view references
        UIImageView *tmp = frontObject;
        frontObject = backObject;
        backObject = tmp;
      }];
}
```

From the Library of Eric Berna

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 6 and open the project for this recipe.

Recipe: Flipping Views

Transitions extend `UIView` animation blocks to add even more visual flair. Four transition styles—`UIViewAnimationOptionTransitionFlipFromLeft`, `UIViewAnimationOptionTransitionFlipFromRight`, `UIViewAnimationOptionTransitionCurlUp`, and `UIViewAnimationOptionTransitionCurlDown`—do just what their names suggest. You can flip views left or flip views right to their backs, and curl views up and down in the manner of the Maps application. Recipe 6-10 demonstrates how to do this.

Recipe 6-10 uses the block-based API for `transitionFromView:toView:duration:options:completion:`. This method replaces a view by removing it from its superview and adding the new view to the initial view's parent. It animates this over the supplied duration using the transition specified in the options flags. Recipe 6-10 uses a flip-from-left transition, although you can use any of the other four transitions as desired.

Superviews normally retain the views added to them and release them when they are removed. You must be careful to retain the view that has been removed if you plan to be able to add it back at a later time.

Recipe 6-10 **Using Transitions with `UIView` Animations**

```
- (void) flip: (id) sender
{
    self.navigationItem.rightBarButtonItem.enabled = NO;
    [UIView transitionFromView: fromPurple ? purple : maroon
                toView: fromPurple ? maroon : purple
                duration: 1.0f
                options: UIViewAnimationOptionTransitionFlipFromLeft
                completion: ^(BOOL done){
                    self.navigationItem.rightBarButtonItem.enabled = YES;
                    fromPurple = !fromPurple;
                }];
}
```

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 6 and open the project for this recipe.

From the Library of Eric Berna

Recipe: Using Core Animation Transitions

In addition to `UIView` animations, iOS supports Core Animation as part of its Quartz Core framework. The Core Animation API offers highly flexible animation solutions for your iOS applications. Specifically, it offers built-in transitions that provide the same kind of view-to-view changes you've been reading about in the previous recipe.

Core Animation Transitions expand your `UIView` animation vocabulary with just a few small differences in implementation. `CATransitions` work on layers rather than on views. Layers are the Core Animation rendering surfaces associated with each `UIView`. When working with Core Animation, you apply `CATransitions` to a view's default layer (`[myView layer]`) rather than the view itself.

With these transitions, you don't set your parameters through `UIView` the way you do with `UIView` animation. You create a Core Animation object, set its parameters, and then add the parameterized transition to the layer.

```
CATransition *animation = [CATransition animation];
animation.delegate = self;
animation.duration = 1.0f;
animation.type = kCATransitionMoveIn;
animation.subtype = kCATransitionFromTop;

// Perform some kind of view exchange or removal here

[myView.layer addAnimation:animation forKey:@"move in"];
```

Animations use both a type and a subtype. The **type** specifies the kind of transition used. The **subtype** sets its direction. Together the type and subtype tell how the views should act when you apply the animation to them.

Core Animation transitions are distinct from the `UIViewAnimationTransitions` discussed in previous recipes. Cocoa Touch offers four types of Core Animation transitions, which are highlighted in Recipe 6-11. These available types include cross-fades, pushes (one view pushes another offscreen), reveals (one view slides off another), and covers (one view slides onto another). The last three types enable you to specify the direction of motion for the transition using their subtypes. For obvious reasons, cross-fades do not have a direction and they do not use subtypes.

Because Core Animation is part of the Quartz Core framework, you must add the Quartz Core framework to your project and import `<QuartzCore/QuartzCore.h>` into your code when using these features.

> **Note**
>
> Apple's Core Animation features 2D and 3D routines built around Objective-C classes. These classes provide graphics rendering and animation for your iOS and Macintosh applications. Core Animation avoids many low-level development details associated with, for example, direct Open GL while retaining the simplicity of working with hierarchical views.

From the Library of Eric Berna

Recipe 6-11 **Animating Transitions with Core Animation**

```
- (void) animate: (id) sender
{
    // Set up the animation
    CATransition *animation = [CATransition animation];
    animation.delegate = self;
    animation.duration = 1.0f;

    switch ([(UISegmentedControl *)self.navigationItem.titleView
                selectedSegmentIndex])
    {
        case 0:
            animation.type = kCATransitionFade;
            break;
        case 1:
            animation.type = kCATransitionMoveIn;
            break;
        case 2:
            animation.type = kCATransitionPush;
            break;
        case 3:
            animation.type = kCATransitionReveal;
        default:
            break;
    }
    animation.subtype = kCATransitionFromLeft;

    // Perform the animation


    [self.view exchangeSubviewAtIndex:0 withSubviewAtIndex:1];
    [self.view.layer addAnimation:animation forKey:@"animation"];
}
```

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 6 and open the project for this recipe.

Recipe: Bouncing Views as They Appear

Apple often uses two animation blocks, one called after another finishes, to add bounce to their animations. For example, they might zoom into a view a bit more than needed and then use a second animation to bring that enlarged view down to its final size. Using "bounces" adds a little more life to your animation sequences, providing an extra physical touch.

From the Library of Eric Berna

When calling one animation after another, be sure that the animations do not overlap. The previous edition of this book covered blocking modal animations; however, since iOS introduced completion block support, it is now far easier to use a nested set of animation blocks with chained animations in the completion blocks. Recipe 6-12 uses this new approach to bounce views slightly larger than their end size and then shrink them back down to the desired frame.

This recipe uses two simple `typedefs` to simplify the declaration of each animation and completion block. Notice that the animation block stages that do the work of scaling the view in question are defined in order. The first block shrinks the view, the second one zooms it extra large, and the third restores it to its original size.

The completion blocks go the opposite way. Because each block depends on the one before it, you must create them in reverse order. Start with the final side effects and work your way back to the original. In Recipe 6-12, `bounceLarge` depends on `shrinkBack`, which in turn depends on `reenable`. This reverse definition can be a bit tricky to work with but it certainly beats laying out all your code in nested blocks.

Recipe 6-12 **Bouncing Views**

```
typedef void (^AnimationBlock)(void);
typedef void (^CompletionBlock)(BOOL finished);

- (void) bounce: (id) sender
{
        self.navigationItem.rightBarButtonItem.enabled = NO;

    // Define the three stages of the animation in forward order
    AnimationBlock makeSmall = ^(void){
        bounceView.transform = CGAffineTransformMakeScale(0.01f, 0.01f);};
    AnimationBlock makeLarge = ^(void){
        bounceView.transform = CGAffineTransformMakeScale(1.15f, 1.15f);};
    AnimationBlock restoreToOriginal = ^(void) {
        bounceView.transform = CGAffineTransformIdentity;};

    // Create the three completion links in reverse order
    CompletionBlock reenable = ^(BOOL finished) {
        self.navigationItem.rightBarButtonItem =
            BARBUTTON(@"Start", @selector(bounce:));};
    CompletionBlock shrinkBack = ^(BOOL finished) {
        [UIView animateWithDuration:0.2f
            animations:restoreToOriginal completion: reenable];};
    CompletionBlock bounceLarge = ^(BOOL finished){
        [UIView animateWithDuration:0.2
            animations:makeLarge completion:shrinkBack];};
```

From the Library of Eric Berna

```
    // Start the animation
    [UIView animateWithDuration: 0.5f
         animations:makeSmall completion:bounceLarge];
}
```

> **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 6 and open the project for this recipe.

Recipe: Image View Animations

In addition to displaying static pictures, the `UIImageView` class supports built-in animation. After loading an array of images, you can tell instances to animate them. Recipe 6-13 shows you how.

Start by creating an array populated by individual images loaded from files and assign this array to the `UIImageView` instance's `animationImages` property. Set the `animationDuration` to the total loop time for displaying all the images in the array. Finally, begin animating by sending the `startAnimating` message. (There's a matching `stopAnimating` method available for use as well.)

Once you add the animating image view into your interface, you can place it into a single location, or you can animate it just as you would animate any other `UIView` instance.

Recipe 6-13 **Using `UIImageView` Animation**

```
NSMutableArray *bflies = [NSMutableArray array];
// Load the butterfly images
for (int i = 1; i <= 17; i++)
    [bflies addObject:[UIImage imageWithContentsOfFile:
         [[NSBundle mainBundle]
             pathForResource: [NSString stringWithFormat:@"bf_%d", i]
             ofType:@"png"]]];

// Create the view
UIImageView *butterflyView = [[UIImageView alloc]
    initWithFrame:CGRectMake(40.0f, 300.0f, 60.0f, 60.0f)];

// Set the animation cells, and duration
butterflyView.animationImages = bflies;
butterflyView.animationDuration = 0.75f;
[butterflyView startAnimating];

// Add the view to the parent and release
[self.view addSubview:butterflyView];
[butterflyView release];
```

From the Library of Eric Berna

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 6 and open the project for this recipe.

One More Thing: Adding Reflections to Views

Reflections enhance the reality of onscreen objects. They provide a little extra visual spice beyond the views-floating-over-a-backsplash, which prevails as the norm. Reflections have become especially easy to implement thanks to the `CAReplicatorLayer` class. This class replicates a view's layer, and allows you to apply transforms to that replication.

Listing 6-2 shows how you can use replicator layers to build a view that automatically reflects itself. The class overrides `layerClass` to ensure that the view's layer defaults to a replicator. That replicator is given the `instanceCount` of 2, so it duplicates the original view to a second instance. Its `instanceTransform` specifies how the second instance is manipulated and placed onscreen: namely, flipped, squeezed, and then moved vertically below the original view. This creates a small reflection at the foot of the original view.

Due to the replicator, that reflection is completely "live." Any changes to the main view immediately update in the reflection layer, which you can test yourself using the sample code that accompanies this chapter. You can add scrolling text views, web views, switches, and so forth. Any changes to the original view are replicated, creating a completely passive reflection system.

Good reflections use a natural attrition as the "reflection" moves further away from the original view. This listing adds an optional gradient overlay (using the `usesGradientOverlay` property) that creates a visual drop-off from the bottom of the view. This code adds the gradient as a `CAGradientLayer`.

This code adds an arbitrary gap between the bottom of the view and its reflection. Here, it is hardwired to 10 points, and demonstrated in Figure 6-3, but you can adjust this as you like.

Listing 6-2 **Creating Reflections**

```
@implementation ReflectingView
@synthesize usesGradientOverlay;

// Always use a replicator as the base layer
+ (Class) layerClass
{
    return [CAReplicatorLayer class];
}

// Clean up any existing gradient from parent
- (void) dealloc
{
    [gradient removeFromSuperlayer];
}
```

From the Library of Eric Berna

```
- (void) setupGradient
{
    // Add a new gradient layer to the parent
    UIView *parent = self.superview;
    if (!gradient)
    {
        gradient = [CAGradientLayer layer];
        CGColorRef c1 = [[UIColor blackColor]
            colorWithAlphaComponent:0.5f].CGColor;
        CGColorRef c2 = [[UIColor blackColor]
            colorWithAlphaComponent:0.9f].CGColor;
        [gradient setColors:[NSArray arrayWithObjects:
            (__bridge id)c1, (__bridge id)c2, nil]];
        [parent.layer addSublayer:gradient];
    }

    // Place the gradient just below the view using the
    // reflection's geometry
    float desiredGap = 10.0f; // gap between view and its reflection
    CGFloat shrinkFactor = 0.25f; // reflection size
    CGFloat height = self.bounds.size.height;
    CGFloat width = self.bounds.size.width;
    CGFloat y = self.frame.origin.y;

    [gradient setAnchorPoint:CGPointMake(0.0f,0.0f)];
    [gradient setFrame:CGRectMake(0.0f, y + height + desiredGap,
        width, height * shrinkFactor)];
    [gradient removeAllAnimations];

    [gradient setAnchorPoint:CGPointMake(0.0f,0.0f)];
    [gradient setFrame:CGRectMake(0.0f, y + height + desiredGap,
        maxDimension, height * shrinkFactor)];
    [gradient removeAllAnimations];
}

- (void) setupReflection
{
    CGFloat height = self.bounds.size.height;
    CGFloat shrinkFactor = 0.25f;

    CATransform3D t = CATransform3DMakeScale(1.0, -shrinkFactor, 1.0);

    // scaling centers the shadow in the view. translate in shrunken terms
    float offsetFromBottom = height * ((1.0f - shrinkFactor) / 2.0f);
    float inverse = 1.0 / shrinkFactor;
    float desiredGap = 10.0f;
    t = CATransform3DTranslate(t, 0.0, -offsetFromBottom * inverse
```

From the Library of Eric Berna

```
        - height - inverse * desiredGap, 0.0f);

    CAReplicatorLayer *replicatorLayer = (CAReplicatorLayer*)self.layer;
    replicatorLayer.instanceTransform = t;
    replicatorLayer.instanceCount = 2;

    // Gradient use must be explicitly set
    if (usesGradientOverlay)
        [self setupGradient];
    else
    {
        // Darken the reflection when not using a gradient
        replicatorLayer.instanceRedOffset = -0.75;
        replicatorLayer.instanceGreenOffset = -0.75;
        replicatorLayer.instanceBlueOffset = -0.75;
    }
}
@end
```



Figure 6-3 This layer-based reflection updates in real time to match the view it reflects.

From the Library of Eric Berna

Summary

`UIViews` provide the onscreen components your users see and interact with. As this chapter showed, even in their most basic form, they offer incredible flexibility and power. You discovered how to use views to build up elements on a screen, retrieve views by tag or name, and introduce eye-catching animation. Here's a collection of thoughts about the recipes you saw in this chapter that you might want to ponder before moving on:

- When you're dealing with multiple onscreen views, hierarchy should always remain in your mind. Use your view hierarchy vocabulary (`bringSubviewToFront:`, `sendSubviewToBack:`, `exchangeSubviewAtIndex:withSubviewAtIndex:`) to take charge of your views and always present the proper visual context to your users.
- Don't let the Core Graphics frame/`UIKit` center dichotomy stand in your way. Use functions that help you move between these structures to produce the results you need, especially when you're working with simple views that don't use transforms.
- Make friends with tags. They provide immediate access to views in the same way that your program's symbol table provides access to variables. They are not evil or wrong and can play a useful role in your development vocabulary.
- Blocks are wonderful. Use them to simplify your life, your code, and your animations.
- Animate everything. Animations don't have to be loud, splashy, or bad design. The iOS SDK's strong animation support enables you to add smooth transitions between user tasks. The essence of the iOS experience is subtle, smooth transitions. Short, smooth, focused changes are iOS's bread and butter.

From the Library of Eric Berna

This page intentionally left blank

From the Library of Eric Berna

7

Working with Images

On iOS, images and views have distinct roles. Unlike views, images have no onscreen presence. Although views can use and display images, they are not themselves images, not even `UIImageView` objects, which have the word "image" in their class name. Images are abstract representations of pictures, storing the data that makes up those pictures. This chapter leaves views to the side and focuses just on images. It introduces Cocoa Touch images, specifically the `UIImage` class, and teaches you all the basic know-how you need for working with image data on the iOS. In this chapter, you'll learn how to load, store, and modify image data in your applications. You'll see how to add images to views and how to convert views into images. You'll 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 a device's built-in camera.

Finding and Loading Images

iOS images are generally stored in one of five places. These sources allow you to access image data and display that data in your programs. These sources include the photo album, the application bundle, the sandbox, iCloud, and the Internet:

- **Photo album**—The iOS's photo album contains a camera roll (for camera-ready units), a saved pictures roll, and photos synced from the user's computer or transferred from another digital device using a camera connection kit. Users can request images from this album using the interactive dialog supplied by the `UIImagePickerController` class. The dialog lets users browse through stored photos and select the image they want to work with on an album-by-album basis.
- **Application bundle**—Your application bundle may store images along with your application executable, Info.plist file, and other resources. You can read these bundle-based images using their local file paths and display them in your application.
- **Sandbox**—Your application can also write image files into your sandbox and read them back as needed. The sandbox lets you store files to the Documents, Library, and tmp folders. Each of these folders is readable by your application, and you can

From the Library of Eric Berna

create new images by supplying a file path. The top-level Documents directory can be populated by and accessed from iTunes. Users can add images as well as other files to this shared folder from their computer when the application sets the `UIFileSharingEnabled` key in the Info.plist file. Although other parts of iOS outside the sandbox are technically readable, Apple has made it clear that these areas are off limits for App Store applications.

> **Note**
>
> When you use the Document file sharing option, make sure you store any application-specific files that should not be shared directly with your users in your Library folder.

- **iCloud**—Apple's iCloud service allows you to store documents in a shared central location and access them from all your user's computers and iOS devices. On iOS, you typically use the `UIDocument` class to load iCloud images into your apps.
- **Internet**—Your application can download images from the Net using URL resources to point to web-based files. To make this work, iOS needs an active web connection, but once connected the data from a remote image is just as accessible as data stored locally.

In addition to these sources, iOS applications can also work with image data from at least two other destinations under limited conditions. First, applications can use image data that is stored in the system pasteboard. That's because image data is indistinguishable from any other kind of pasteable data. You can recover that data by querying the `patesboardTypes`. This returns an array of uniform type identifiers that specify what kind of data is currently available on the pasteboard. Image UTIs are typically of the form public.png, public.tiff, or public.jpg, or similar. Each application can choose whether it can or cannot handle the contents of the pasteboard based on those UTIs.

iOS applications can also open and display image files sent by other applications. Applications that declare support for image file types (by defining a `CFBundleDocumentTypes` array in their Info.plist file) can be called upon to open those files. They do so by implementing the `application:didFinishLaunchingWithOptions:` method in their application delegate class. This method is the same application delegate method that handles remote notification events.

Search the incoming options dictionary for the `UIApplicationLaunchOptionsURLKey` key. When that key appears, you know that another application is trying to pass you data, and that you have registered your application to handle that kind of data. Recover the path to the file by extracting the `UIApplicationLaunchOptionsURLKey` object from the options dictionary. Use `UIImage`'s `imageWithData:` class method to produce an image from the data stored at the passed URL.

You'll likely want to copy the image out of the Inbox folder that iOS automatically creates to store passed data and then clean up the Inbox once you are done using that data.

From the Library of Eric Berna

Reading Image Data

An image's file location controls how you can read its data. You'd imagine that you could just use a method such as `UIImage`'s `imageWithContentsOfFile:` to load your images, regardless of their source. In reality, you cannot. Photo album pictures and their paths are (at least officially) hidden from direct application access. Only end users are allowed to browse and choose images, making the chosen image available to the application.

`UIImage` Convenience Methods

The `UIImage` class offers a simple method that loads any image stored in the application bundle. Call `imageNamed:` with a filename, including its extension, for example:

```
myImage = [UIImage imageNamed:@"icon.png"];
```

This method looks for an image with the supplied name in the top-level folder of the application bundle. If found, the image loads and is cached by iOS. That means the image is memory managed by that cache.

If your data is located in another location in your sandbox, use `imageWithContentsOfFile:` instead. Pass it a string that points to the image file's path. This method will not cache the object the way `imageNamed:` does.

As a rule, `imageNamed:` does more work upfront. It non-lazily decodes the file into raw pixels and adds it to the cache, but it gives you memory management for free. Use it for images that you use over and over, especially for small ones. Caching large images may cause more re-reading overhead as you exhaust memory. The method has an undeserved bad reputation due to early iOS bugs, but those issues have long since been resolved.

Both the `named` and `contents` methods offer a huge advantage over other image-loading approaches. For higher scale devices, namely the iPhone 4 and later, with their higher pixel density screens, you can use these methods to automatically select between low- and high-resolution images. For screen scales of 2.0, the method first searches for filenames using a `@2x` suffix.

That means a call to `[UIImage imageNamed:@"basicArt.png"]` will load the file basicArt@2x.png, if available, on higher scale devices, and basicArt.png on 1.0 scale devices. One assumes that this naming convention will continue for even higher scale units should they become available.

This extends further to device names (that is, `~iphone` and `~ipad`). The basicArt~iphone.png image will be loaded on iPhone and iPod touch devices, and basicArt~ipad.png on iPads. Assuming that Apple someday introduces a Retina-display-based iPad, that device would first load (the now theoretical) basicArt@2x~ipad.png over the existing basicArt.png, basicArt~ipad.png, basicArt~iphone.png, or basicArt@2x~iphone.png variants.

> **Note**
>
> iOS supports the following image types: PNG, JPG, JPEG, TIF, TIFF, GIF, BMP, BMPF, ICO, CUR, XBM, and PDF. The `UIImage` class does not read or display PDF files; use `UIWebView` instead.

From the Library of Eric Berna

Finding Images in the Sandbox

By default, each sandbox contains three folders: Documents, Library, and tmp. User-generated data files, including images, normally reside in the Documents folder. This folder does exactly what the name suggests. You store documents to and access them from this directory. Keep document file data here that is created by or browsed from your program. You can also use the Documents folder and iTunes to allow desktop-based users direct access to data.

The Library folder stores user defaults and other state information for your program. Use the Library folder for any application support files that must be retained between successive runs and that are not meant for general end-user access.

The tmp folder provides a place to create transient files on the fly. Unlike in tmp, files in Documents and Library are not transient. iTunes backs up all Documents and most Library files (items in Library/Caches are not backed up, just like on the Mac) whenever iOS syncs. In contrast, iOS discards any tmp files when it reboots.

These directories demonstrate one of the key differences between Mac OS X and iOS programming. You're free to use both standard and nonstandard file locations on the Macintosh. iOS with its sandbox is far more structured—rigidly so by Apple's dictates; its files appear in better-defined locations. On the Macintosh, locating the Documents folder usually means searching the user domain. This is the standard way to locate Documents folders, which works on both platforms:

```
NSArray *paths = [NSSearchPathForDirectoriesInDomains(
    NSDocumentDirectory, NSUserDomainMask, YES);
return [paths lastObject];
```

iOS is more constrained. You can reliably locate the top sandbox folder by calling a utility home directory function. The result of `NSHomeDirectory()` lets you navigate down one level to Documents with full assurance of reaching the proper destination. The following function provides a handy way to return paths to the Documents folders:

```
NSString *documentsFolder()
{
    return [NSHomeDirectory()
        stringByAppendingPathComponent:@"Documents"];
}
```

To load an image, append its filename to the returned path and tell `UIImage` to create a new image with those contents. This code loads a file named image.png from the top level of the documents folder and returns a `UIImage` instance initialized with that data:

```
path = [documentsFolder() stringByAppendingPathComponent:@"image.png"];
return [UIImage imageWithContentsOfFile:path];
```

This call returns `nil` if the path is not valid or points to a non-image resource.

Loading Images from URLs

The `UIImage` class can load images from `NSData` instances, but it cannot do so directly from URL strings or `NSURL` objects. So supply `UIImage` with data already downloaded

From the Library of Eric Berna

from a URL. This snippet downloads the latest United States weather map from Weather.com and then creates a new image using the weather data. First, it constructs an `NSURL` object, and then it creates an `NSData` instance initialized with the contents of that URL. The data returned helps build the `UIImage` instance.

```
NSURL *url = [NSURL URLWithString:
    @"http://image.weather.com/images/maps/current/curwx_600x405.jpg"];
UIImage *img = [UIImage imageWithData:
    [NSData dataWithContentsOfURL:url]];
```

It's easy enough to write a method that handles this process for you, letting you supply a URL string to retrieve a `UIImage`. This method takes one argument, a URL string, and returns a `UIImage` built from that resource:

```
+ (UIImage *) imageFromURLString: (NSString *) urlstring
{
    // This call is synchronous and blocking
    return [UIImage imageWithData:[NSData
        dataWithContentsOfURL:[NSURL URLWithString:urlstring]]];
}
```

This is a synchronous method, with certain drawbacks. It may fail without feedback and doesn't have a built-in timeout. You can adapt this code to operate in the background, allowing the data to load itself asynchronously without blocking the main thread. Here's one approach:

```
// Create an asynchronous background queue
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
[queue addOperationWithBlock:
^{
    // Load the weather data
    NSURL *weatherURL = [NSURL URLWithString:
        @"http://image.weather.com/images\
        /maps/current/curwx_600x405.jpg"];
    NSData *imageData = [NSData dataWithContentsOfURL:weatherURL];

    // Update the image on the main thread using the main queue
    [[NSOperationQueue mainQueue] addOperationWithBlock:^{
        UIImage *weatherImage = [UIImage imageWithData:imageData];
        imageView.image = weatherImage;}];
```

A more thorough implementation would return a placeholder image, cache the retrieved data locally, and update the main thread once the placeholder could be replaced with the downloaded asset.

Loading Data from the Photo Album

The `UIImagePickerController` class helps users select images from the iOS photo album. It provides a standalone view controller that you present modally (on the iPhone)

From the Library of Eric Berna

or from a popover controller (on the iPad). The controller sends back delegate messages reflecting the image choice made by the user. Both the controller and the popover also issue delegate callbacks regarding the stage of the image-selection process. This approach is discussed in the following recipe.

Recipe: Accessing Photos from the iOS Photo Album

The `UIImagePickerController` class offers a highly specialized interface with relatively few public methods and some modest quirks. It's designed to operate solely as a modal dialog or popover, and it has its own navigation controller built in. If you push it onto an existing navigation controller–based view scheme, it adds a second navigation bar below the first. That means that although you can use it with a tab bar and as an independent view system, you can't really push it onto an existing navigation stack and have it look right.

Recipe 7-1 shows the picker in its simplest mode. It enables users to select an image from any of the onboard albums; this operation is seen in Figure 7-1. Set the picker to use any of the legal source types. The three kinds of sources follow:

- **`UIImagePickerControllerSourceTypePhotoLibrary`**—All images synced to iOS plus any camera roll, including pictures snapped by the user.
- **`UIImagePickerControllerSourceTypeSavedPhotosAlbum`**—Also called the "camera roll." Refers to pictures snapped by the user on camera-ready units or to the Saved Photos album for noncamera units.
- **`UIImagePickerControllerSourceTypeCamera`**—Allows users to shoot pictures with a built-in iPhone camera, including front and back camera selection.

Working with the Image Picker

Recipe 7-1 follows a basic work path. Select an album, select an image, display the selected image, and then repeat. This simple flow works because there's no image editing involved. The picker's image editing property defaults to `NO`. This `allowsEditing` property tells the image picker whether to allow users to frame and stretch an image. When it is disabled, any selection (basically any image tap) redirects control to the `UIImagePickerControllerDelegate` object via the finished picking image method.

The delegate for an image picker must conform to two protocols, namely `UINavigationControllerDelegate` and `UIImagePickerControllerDelegate`. In addition, when using an image picker controller with the iPad, you must declare the `UIPopoverControllerDelegate` protocol because the image picker needs to run as part of a popover. Be sure to declare these in the interface for the object you set as the picker delegate.

From the Library of Eric Berna

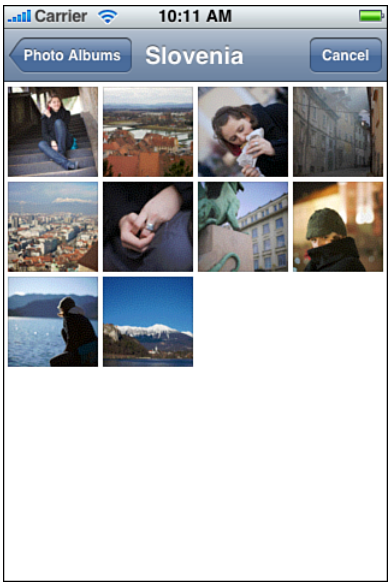


Figure 7-1 The core image picker allows users to select images by offering them small thumbnails.

Adding iPad Support

On the iPhone, you create a picker controller and present it modally. On the iPad, you embed a popover controller with your image picker. Because of this difference, you need to check for the runtime device when building universal applications. The following test produces a true value when run on an iPhone, false on an iPad:

```
#define IS_IPHONE \
    (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPhone)
```

Use this test both when creating your picker and whenever you receive a media-picking delegate callback. On the iPhone, you will want to dismiss and release your modal controller during that callback. On the iPad, you need to wait until the popover is dismissed before allowing the picker to be released. Recipe 7-1 demonstrates how to create and present the proper picker interface depending on the device in use.

```
- (void)popoverControllerDidDismissPopover:
    (UIPopoverController *)aPopoverController
{
    popoverController = nil;
    imagePickerController = nil;
}
```

From the Library of Eric Berna

Populate the Simulator's Photo Collection for Testing

Populate the simulator's photo collection by dropping images onto it from Finder. Each image opens in Mobile Safari, where you can then tap-and-hold and choose Save Image to copy the image to your photo library. Once you've set up your photos as you like, navigate to ~/Library/Application Support/iPhone Simulator and into the iOS version you're currently using. Back up your Media folder so you can restore it in the future without having to re-add each photo individually.

Customizing Images

To enable image editing in a `UIImagePickerController`, set the `allowsEditing` property to `YES`. This allows users to scale and position images after selection, or in the case of camera shots, after snapping a photo. You can see this editor in action on iOS when using the Set Wallpaper feature of Settings. Figure 7-2 shows the post-selection editor using both the iPhone and iPad. The iPad picker is presented in a popover view.

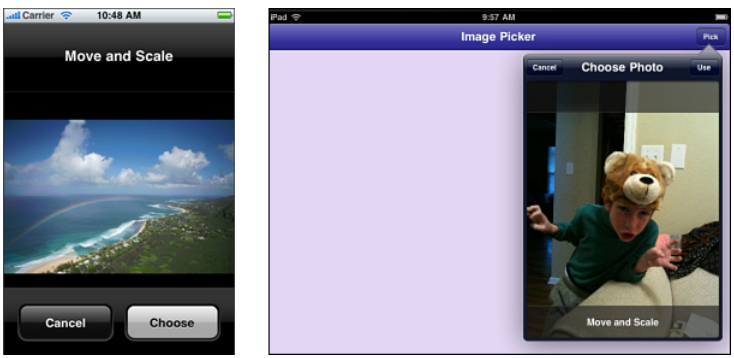


Figure 7-2 The interactive image editor allows users to move, scale, and choose their final presentation.

This window allows users to move and scale the image as desired. Pinching and unpinching changes the image scale. Dragging resets the image origin. When the user taps Use, control moves to the picker delegate, and your program picks up from there. Something different happens when users tap Cancel. Control returns to the album view, allowing the user to select another image and start over.

Recovering Image Edit Information

The picker callback returns a dictionary containing information about the selected image. The info dictionary contains keys that provide access to important dictionary data:

- **`UIImagePickerControllerMediaType`**—Defines the kind of media selected by the user—normally `public.image` for images or `public.movie` for movies. Media types are defined in the UTCoreTypes.h header file, which is part of the Mobile

From the Library of Eric Berna

Core Services framework. Media types are primarily used for adding items to the system pasteboard.

- **`UIImagePickerControllerCropRect`**—Returns the section of the image selected by the user. Oddly enough, this returns as an `NSRect`, a data type equivalent to `CGRect` but more normally used on Mac OS X rather than iOS.
- **`UIImagePickerControllerOriginalImage`**—Stores a `UIImage` instance with the original (nonedited) image contents.
- **`UIImagePickerControllerEditedImage`**—Provides the edited version of the image, containing the portion of the picture selected by the user. The `UIImage` returned is small, sized to fit the iPhone screen.
- **`UIImagePickerControllerReferenceURL`**—Specifies a filesystem URL for the selected asset, as discussed in Recipe 7-2.

> **Note**
>
> When it comes to user interaction elements, the `UIImagePickerController` is a cow. It is slow to load. It hogs application memory. Be aware of these limitations when designing your apps.

Recipe 7-1 **Simple `UIImagePickerController` Image Selection**

```
// Update image and for iPhone, dismiss the controller
- (void)imagePickerController:(UIImagePickerController *)picker
    didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    // Retrieve the edited image if available, otherwise original
    UIImage *image = [info objectForKey:
        UIImagePickerControllerEditedImage];
    if (!image)
        image = [info objectForKey:
            UIImagePickerControllerOriginalImage];
    imageView.image = image;
    if (IS_IPHONE)
    {
        [self dismissModalViewControllerAnimated:YES];
        imagePickerController = nil;
    }
}

// Dismiss picker
- (void) imagePickerControllerDidCancel:
    (UIImagePickerController *)picker
{
    [self dismissModalViewControllerAnimated:YES];
    imagePickerController = nil;
}
```

From the Library of Eric Berna

```
// Popover was dismissed. Clean up here for iPad
- (void)popoverControllerDidDismissPopover:
    (UIPopoverController *)aPopoverController
{
    imagePickerController = nil;
    popoverController = nil;
}

- (void) pickImage: (id) sender
{
    // Create and initialize the picker
    imagePickerController = [[UIImagePickerController alloc] init];
    imagePickerController.sourceType =
        UIImagePickerControllerSourceTypePhotoLibrary;
    imagePickerController.allowsEditing = editSwitch.isOn;
    imagePickerController.delegate = self;

    if (IS_IPHONE)
    {
        [self presentModalViewController:imagePickerController
            animated:YES];
    }
    else
    {
        // Clean up any pre-existing popover
        if (popoverController)
            [popoverController dismissPopoverAnimated:NO];

        // Establish the new popover and hold onto it
        popoverController = [[UIPopoverController alloc]
           initWithContentViewController:imagePickerController];
        popoverController.delegate = self;

        // Present the popover
        [popoverController presentPopoverFromBarButtonItem:
                self.navigationItem.rightBarButtonItem
            permittedArrowDirections:UIPopoverArrowDirectionAny
            animated:YES];
    }
}

- (void) loadView
{
    [super loadView];

    imageView = [[UIImageView alloc] init];
    imageView.contentMode = UIViewContentModeScaleAspectFit;
```

From the Library of Eric Berna

```
    [self.view addSubview:imageView];

    // Allow or disallow edits
    editSwitch = [[UISwitch alloc] init];
        self.navigationItem.titleView = editSwitch;


    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Pick", @selector(pickImage:));
}
```

> **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 7 and open the project for this recipe.

Recipe: Retrieving Images from Asset URLs

When image picker controllers retrieve images, they offer URLs that point to the selected image. A typical URL looks something like this:

```
assets-library://asset/asset.JPG?id=553F6592-43C9-45A0-B851-28A726727436&ext=JPG
```

You retrieve it from the picker's info dictionary:

```
NSURL *url = [info objectForKey:UIImagePickerControllerReferenceURL];
```

This URL offers continued direct access to the asset in question even though your application normally is not given permission to access the system's onboard DCIM folder. You can use this URL to load images at any time, even after the image picker is no longer in use.

Recipe 7-2 demonstrates how to use the Assets Library to pull out an image via its URL. To compile this application, you must add the AssetsLibrary framework to your build phases. This recipe is device-only. It will compile but crash at runtime on the iOS simulator when you ask the retrieved asset representation for its data.

Be aware that if you use this URL with Assets Library access methods, your user will be prompted to agree to Core Location tracking, as shown in Figure 7-3. Although this makes absolutely no sense, it's a grim reality of the current SDK—although Apple may update this approach in the future. You can warn the user in advance that this dialog will appear and caution the user to tap OK, but most people will still tap "Don't Allow" out of reflex, caution, or sheer bullheadedness.

If the user denies the Core Location request, you will *not* be able to load the asset. The library retrieval will fail, producing the following error:

```
The user has denied the application access to their media.
```

From the Library of Eric Berna

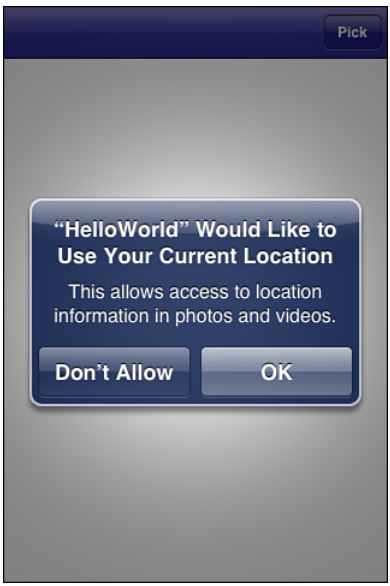


Figure 7-3 Users are asked to grant permission to use their location when accessing the Assets Library.

To program around this, you might store the selected image to a local cache in the did-finish-picking method. That is the only time the image and its data are made directly available for you without user consent.

Recipe 7-2 **Using the Assets Library to Retrieve Images**

```
- (void) loadImageFromAssetURL: (NSURL *) assetURL
{
    ALAssetsLibrary* library = [[ALAssetsLibrary alloc] init];

    ALAssetsLibraryAssetForURLResultBlock result =
    ^(ALAsset *__strong asset){
        ALAssetRepresentation *assetRepresentation =
            [asset defaultRepresentation];
        // This data retrieval crashes on the simulator.
        CGImageRef cgImage =
            [assetRepresentation CGImageWithOptions:nil];
        if (cgImage)
            imageView.image = [UIImage imageWithCGImage:cgImage];
    };

    ALAssetsLibraryAccessFailureBlock failure =
    ^(NSError *__strong error){
```

From the Library of Eric Berna

```
        NSLog(@"Error retrieving asset from url: %@",
            [error localizedFailureReason]);
    };

    [library assetForURL:assetURL
        resultBlock:result failureBlock:failure];
}

// Retrieve image by URL
- (void)imagePickerController:(UIImagePickerController *)picker
    didFinishPickingMediaWithInfo:(NSDictionary *)info
{
        if (IS_IPHONE)
        {
        [self dismissModalViewControllerAnimated:YES];
        imagePickerController = nil;
        }

    NSURL *url = [info objectForKey:UIImagePickerControllerReferenceURL];
    NSLog(@"About to load asset from %@", url);
    [self loadImageFromAssetURL:url];
}
```

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 7 and open the project for this recipe.

Recipe: Snapping Photos and Writing Them to the Photo Album

The image picker controller allows you to snap photos with the device's built-in camera. Users shoot a picture and decide whether to use that image. Because cameras are not available on all iOS units (specifically, older iPod touch and iPad devices), begin by checking whether the system running the application supports camera usage.

This snippet checks for a camera, limiting access to the "snap" button. It presents a "Camera not available" message when an onboard camera is not available. In real-world applications, you'll want to skip the modal alert; it is used for demonstration purposes in this sample code, to give positive feedback for camera-less systems.

```
if ([UIImagePickerController isSourceTypeAvailable:
    UIImagePickerControllerSourceTypeCamera])
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Snap", @selector(snapImage:));
else
    self.title = @"Camera not available";
```

From the Library of Eric Berna

As with other modes, you can allow or disallow image editing as part of the photo-capture process. One feature the camera interaction brings that has no parallel is the Preview screen. This displays after the user taps the camera icon, which is shown in Figure 7-4. The Preview screen lets users retake the photo or use the photo as is. Once Use is tapped, control passes to the next phase. If you've enabled image editing, the user can do so next. If not, control moves to the standard "did finish picking" method.

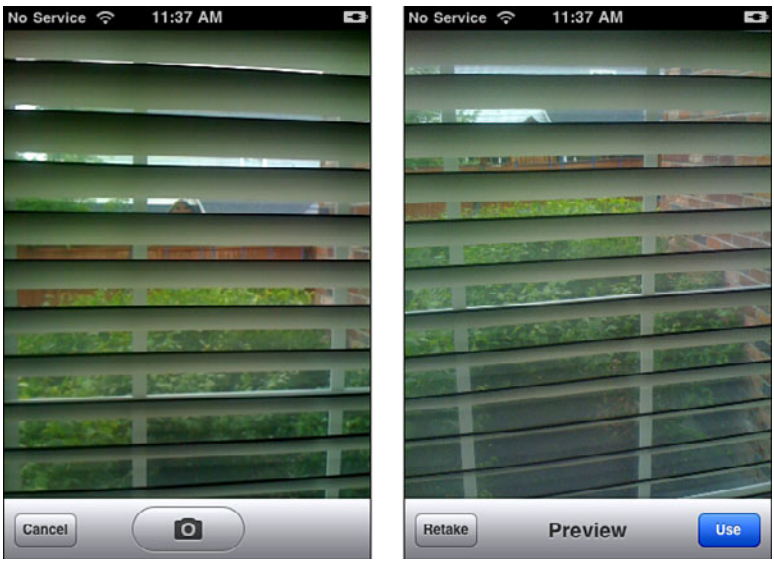


Figure 7-4 After the snap button (Camera icon, left) is pressed, the Preview screen lets users choose whether to use or retake the image.

The sample code that accompanies this recipe displays the returned image using a content mode of `UIViewContentModeScaleAspectFit`. Without this option, just a part of the image would be shown. That's because the captured picture can be much larger than the device screen for better cameras, specifically the nonvideo still cameras.

This code also saves the snapped image to the photo album by calling `UIImageWriteToSavedPhotosAlbum()`. This function can save any image, not just those from the onboard camera. Its second and third arguments specify a callback target and selector. The selector must take three arguments itself, as shown in Recipe 7-3; these are an image, an error, and a pointer to context information. Photos snapped from applications do not contain geotagging information.

This same approach should work when Apple adds cameras to newer versions of its devices, such as future models of the iPad and iPod touch.

From the Library of Eric Berna

Choosing Between Cameras

Newer devices starting with the iPhone 4, iPod touch fourth generation, and iPad 2 offer more than one onboard camera. You can select which camera you wish to use by assigning the `cameraDevice` property. Query whether a camera device is available using the `isCameraDeviceAvailable:` class method. The rear camera is always the default.

```
if ([UIImagePickerController isCameraDeviceAvailable:
    UIImagePickerControllerCameraDeviceFront])
    ipc.cameraDevice = UIImagePickerControllerCameraDeviceFront;
```

Here are a few more points about the onboard camera or cameras that you can access through the `UIImagePickerController` class:

- You can query the device's ability to use flash using the `isFlashAvailableForCameraDevice:` class method. Supply either the front or back device constant. This method returns `YES` for available flash, or otherwise `NO`.
- When a camera supports flash, you can set the `cameraFlashMode` property directly to auto (`UIImagePickerControllerCameraFlashModeAuto`, which is the default), to always used (`UIImagePickerControllerCameraFlashModeOn`), or always off (`UIImagePickerControllerCameraFlashModeOff`). Selecting "off" disables the flash regardless of ambient light conditions.
- Choose between photo and video capture by setting the `cameraCaptureMode` property. The picker defaults to photo capture mode. You can test what modes are available for a device using `availableCaptureModesForCameraDevice:`. This returns an array of `NSNumber` objects, each of which encodes a valid capture mode, either photo (`UIImagePickerControllerCameraCaptureModePhoto`) or video (`UIImagePickerControllerCameraCaptureModeVideo`).

Recipe 7-3 **Snapping Photo Images with the Onboard Camera**

```
- (void) snapImage: (id) sender
{
    // Create and initialize the picker
    imagePickerController = [[UIImagePickerController alloc] init];
    imagePickerController.sourceType =
        UIImagePickerControllerSourceTypeCamera;
    imagePickerController.delegate = self;

    if (IS_IPHONE)
    {
        [self presentModalViewController:imagePickerController
            animated:YES];
    }
    else
    {
        if (popoverController)
```

From the Library of Eric Berna

```
            [popoverController dismissPopoverAnimated:NO];
        popoverController = [[UIPopoverController alloc]
            initWithContentViewController:imagePickerController];
        popoverController.delegate = self;
        [popoverController presentPopoverFromBarButtonItem:
                self.navigationItem.rightBarButtonItem
            permittedArrowDirections: UIPopoverArrowDirectionAny
            animated:YES];
    }
}

- (void)imagePickerController:(UIImagePickerController *)picker
    didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    UIImage *image =
        [info objectForKey: UIImagePickerControllerEditedImage];
    if (!image)
        image = [info objectForKey:
            UIImagePickerControllerOriginalImage];
    imageView.image = image;

    UIImageWriteToSavedPhotosAlbum(image, self,
        @selector(image:didFinishSavingWithError:contextInfo:), nil);

    if (IS_IPHONE)
    {
        [self dismissModalViewControllerAnimated:YES];
        imagePickerController = nil;
    }
}

- (void)image:(UIImage *)image
    didFinishSavingWithError: (NSError *)error
    contextInfo:(void *)contextInfo;
{
    // Handle the end of the image write process
    if (!error)
        NSLog(@"Image written to photo album");
    else
        NSLog(@"Error writing to photo album: %@",
            [error localizedDescription]);
}
```

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 7 and open the project for this recipe.

From the Library of Eric Berna

Adding a Flashlight

The iPhone 4 introduced the first iOS device with a built-in LED camera flash. It's simple to control that LED from your application using what's called "torch" mode. The following snippet uses a button that toggles the light on and off. The button's selected state determines whether the light is switched off or illuminated.

```
- (IBAction) toggleLightSwitch: (UIButton *) sender
{
    // Recover the default back camera
    AVCaptureDevice *device = [AVCaptureDevice
        defaultDeviceWithMediaType:AVMediaTypeVideo];

    // Lock the device
    if ([device lockForConfiguration:nil])
    {
        // Toggle the button from on to off or vice versa
        button.selected = !button.selected;

        // Toggle the light on or off
        device.torchMode = (button.selected) ?
            AVCaptureTorchModeOn : AVCaptureTorchModeOff;

        // Unlock and proceed
        [device unlockForConfiguration];
    }
}
```

Saving Pictures to the Documents Folder

Each `UIImage` can convert itself into JPEG or PNG data. Two built-in `UIKit` functions produce the necessary `NSData` from `UIImage` instances. These functions are `UIImageJPEGRepresentation()` and `UIImagePNGRepresentation()`. The JPEG version takes two arguments—the image and a compression quality that ranges from 0.0 (lowest quality, maximum compression) to 1.0 (highest quality, minimum compression). The PNG version takes one argument—the image.

To write the image to file, use the `NSData` object that is returned by either function and call the `writeToFile:atomically:` method. This stores the image data to a path that you specify. Setting the second argument to `YES` ensures that the entire file gets written before being placed into that path. This guarantees that you won't have to handle the consequences of partial writes.

```
UIImage *image = [info objectForKey:
        @"UIImagePickerControllerOriginalImage"];
[UIImagePNGRepresentation(image)
        writeToFile:uniqueSavePath() atomically:YES];
```

From the Library of Eric Berna

You'll want to save your new file in such a way that it does not overwrite any existing image. The following snippet generates a name that will not overlap any existing file in the Documents folder, allowing you to search for a unique name to save your file to. This is, admittedly, an approach that's not meant to be used over and over. If you need to name a lot of files, consider creating time-based filenames (via `NSDate`), unique IDs (using `CFUUIDCreateString()`), or just saving the most recently used tag to application defaults, so you can pick up counting where you left off.

```
NSString *uniqueSavePath()
{
    int i = 1;
    NSString *path;
    do {
        // iterate until a name does not match an existing file
        path = [NSString stringWithFormat:
            @"%@/Documents/IMAGE_%04d.PNG", NSHomeDirectory(), i++];
    } while ([[NSFileManager defaultManager] fileExistsAtPath:path]);

    return path;
}
```

File-writing speed varies. On the simulator, it runs very fast. On older iPhones, it may take several seconds. You can always write out data on secondary threads to keep your main thread responsive.

Recipe: E-mailing Pictures

The Message UI framework allows users to compose e-mail directly within applications. You can add the framework via the TARGETS > Build Phases > Link Binary With Libraries settings. Click +, select MessageUI.framework, click Add. Drag the newly added framework down to the Frameworks group.

As with camera access and the image picker, you must check to see if a user's device has been enabled for e-mail. A simple test allows you to determine if mail is available:

```
[MFMailComposeViewController canSendMail]
```

When mail capabilities are enabled, allow users to send their photographs via instances of `MFMailComposeViewController`. Recipe 7-4 uses this composition class to create a new mail item populated with the user-selected photograph. Like other special-purpose view controllers, the mail composition controller works best as a modally presented client. Your primary view controller presents it and waits for results via a delegate callback.

Creating Message Contents

The composition controller's properties allow you to programmatically build a message including attachments. Recipe 7-4 demonstrates the creation of a simple HTML message with an attachment. Properties are almost universally optional. Define the subject and

From the Library of Eric Berna

body contents via `setSubject:` and `setMessageBody:`. Each method takes a string as its argument.

Leave the "To Recipients" unassigned to greet the user with an unaddressed message. The only time you'll want to pre-fill this field is when adding call-home features such as "Report a bug" or "Send feedback" to the developer or when you allow the user to choose a favorite recipient in your settings.

Creating the attachment requires slightly more work. To add an attachment, you need to provide all the file components expected by the mail client. You supply data (via an `NSData` object), a MIME type (a string), and a filename (another string). Retrieve the image data using the `UIImageJPEGRepresentation()` function. This function takes some time, often several seconds, to work. So expect a delay before the message view appears.

This recipe uses a hardcoded MIME type of `image/jpeg`. If you want to send other data types, you can query iOS for MIME types via typical file extensions. Use `UTTypeCopyPreferredTagWithClass()` as shown in the following method, which is defined in the MobileCoreServices framework:

```
#import <MobileCoreServices/UTType.h>
- (NSString *) mimeTypeForExtension: (NSString *) ext
{
    // Request the UTI via the file extension
    CFStringRef UTI = UTTypeCreatePreferredIdentifierForTag(
        kUTTagClassFilenameExtension,
        (__bridge CFStringRef) ext, NULL);
    if (!UTI) return nil;

    // Request the MIME file type via the UTI,
    // may return nil for unrecognized MIME types
    NSString *mimeType = (__bridge_transfer NSString *)
        UTTypeCopyPreferredTagWithClass(UTI, kUTTagClassMIMEType);

    return mimeType;
}
```

This method returns a standard MIME type based on the file extension passed to it, such as JPG, PNG, TXT, HTML, and so on. Always test to see if this method returns `nil` because the iOS's built-in knowledge base of extension-MIME type matches is limited. Alternatively, search on the Internet for the proper MIME representations, adding them to your project by hand.

The e-mail uses a filename you specify to transmit the data you send. Use any name you like. Here, the name is set to pickerimage.jpg. Because you're just sending data, there's no true connection between the content you send and the name you assign.

```
[mcvc addAttachmentData:UIImageJPEGRepresentation(image, 1.0f)
    mimeType:@"image/jpeg" fileName:@"pickerimage.jpg"];
```

From the Library of Eric Berna

Presenting the Composition Controller

How you present depends on the platform. Recall that the image picker appears in a popover on iPads and in a modal presentation on iPhone-style devices. Each platform requires a slightly nuanced approach.

Although you can dismiss the image picker on the iPhone, which is itself presented as a modal controller, you run into issues of timing. Overlapping the dismissal and the presentation is a bad thing. Instead, Recipe 7-4 presents the composition controller from the image picker itself.

On the iPad, the recipe takes a different approach. First, it dismisses the active popover and then it presents the composer as a new modal presentation using a flipped form sheet. These styles are set in the `emailImage:` method before the controller is presented, allowing the e-mail composition view to take center stage onscreen.

Once it is presented, control passes to the composition controller and its delegate. The primary controller declares the `MFMailComposeViewControllerDelegate` protocol and implements the single callback that is responsible for dismissing the controller.

Recipe 7-4 **Sending Images by E-mail**

```
- (NSString *) mimeTypeForExtension: (NSString *) ext
{
    // Request the UTI via the file extension
    CFStringRef UTI = UTTypeCreatePreferredIdentifierForTag(
        kUTTagClassFilenameExtension, (__bridge CFStringRef)
        ext, NULL);
    if (!UTI) return nil;

    // Request the MIME file type via the UTI,
    // may return nil for unrecognized MIME types

    NSString *mimeType = (__bridge_transfer NSString *)
        UTTypeCopyPreferredTagWithClass(UTI, kUTTagClassMIMEType);

    return mimeType;
}

- (void)mailComposeController:(MFMailComposeViewController*)controller
          didFinishWithResult:(MFMailComposeResult)result
                        error:(NSError*)error
{
    // Dismiss the e-mail controller once the user is done
    [self dismissModalViewControllerAnimated:YES];
}


- (void) emailImage: (UIImage *) image
{
    if (![MFMailComposeViewController canSendMail])
```

From the Library of Eric Berna

```
    {
        if (IS_IPHONE)
        {
            [self dismissModalViewControllerAnimated:YES];
            imagePickerController = nil;
        }
        return;
    }


    // Customize the e-mail
    MFMailComposeViewController *mcvc =
        [[MFMailComposeViewController alloc] init];
    mcvc.mailComposeDelegate = self;
    [mcvc setSubject:@"Here's a great photo!"];
    NSString *body = @"<h1>Check this out</h1>\
        <p>I selected this image from the\
        <code><b>UIImagePickerController</b></code>.</p>";
    [mcvc setMessageBody:body isHTML:YES];
    [mcvc addAttachmentData:UIImageJPEGRepresentation(image, 1.0f)
                   mimeType:@"image/jpeg" fileName:@"pickerimage.jpg"];

    // Present the e-mail composition controller
    if (IS_IPHONE)
        [picker presentModalViewController:mcvc animated:YES];
    else
    {
        [popoverController dismissPopoverAnimated:NO];
        mcvc.modalTransitionStyle = UIModalTransitionStyleFlipHorizontal;
        mcvc.modalPresentationStyle = UIModalPresentationFormSheet;
        [self presentModalViewController:mcvc animated:YES];
    }
}



// Update image and for iPhone, dismiss the controller
- (void)imagePickerController:(UIImagePickerController *)picker
    didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    UIImage *image = [info objectForKey:
        UIImagePickerControllerEditedImage];
    if (!image)
        image = [info objectForKey:UIImagePickerControllerOriginalImage];
    imageView.image = image;

    [self emailImage:image];
}
```

From the Library of Eric Berna

> **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 7 and open the project for this recipe.

Automating Camera Shots

There are times when you just want to use the camera to take a quick shot without user interaction. For example, you might write a utility that does time-lapse photography as you're biking, or you may want to build an application that builds stop-motion animation. You can accomplish this by building an AVFoundation solution, but for a simpler approach, you can just use an image picker controller.

Two APIs enable this kind of capture. The `showsCameraControls` property allows you to hide the normal camera GUI, presenting a full-screen camera preview instead. Set this property to `NO`.

```
ipc.showsCameraControls = NO;
```

To programmatically capture an image rather than depend on user input, call the `takePicture` method. (Corresponding video methods are `startVideoCapture` and `stopVideoCapture`.) This begins the photo-acquisition process, just as if a user had pressed the snap button. When the photo is ready, the picker sends the `imagePickerController:didFinishPickingMediaWithInfo:` callback to its delegate. You cannot capture another picture until after this method is called.

When using the iPhone to snap photos over a long period of time, make sure to disable the `UIApplication`'s idle timer, as follows. This code ensures that the device will not sleep even though a user has not interacted with it for a while:

```
[UIApplication sharedApplication].idleTimerDisabled = YES;
```

Using a Custom Camera Overlay

Custom overlays create a GUI that floats over the live camera preview. You can add buttons and other user interface controls to snap photographs and dismiss the controller. Figure 7-5 shows a rudimentary overlay with two buttons: one for snapping a photo, the other (the small circled "X") for dismissing the image picker controller. Notice the black bar behind the overlay and under the camera preview. The overlay is deliberately offset to highlight this bar. Hiding the standard controls leaves this black bar in place. You'll want to design your overlay to best take advantage of this geometry.

Set the overlay by assigning a view to the picker's `cameraOverlayView` property and hide the normal controls. When you present the picker, the custom overlay, not the built-in one, appears. You'll want to match your custom camera overlay to the screen properties of the device you're working with.

From the Library of Eric Berna

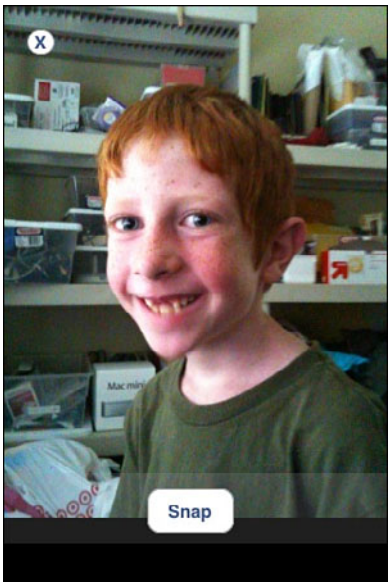


Figure 7-5 Snapping photos with a custom image picker overlay.

The `cameraViewTransform` property provides a way to change how the camera view is shown. This property comes in handy for front-mounted cameras. Use a negative-X transform (basically scale by –1 and 1) to flip the preview.

Recipe: Accessing the AVFoundation Camera

AVFoundation allows you access to camera buffer without using the cumbersome image picker. It's faster and more responsive than the picker, but does not offer the friendly built-in interface. For example, you might want to build an application that evaluates the contents of the camera for augmented reality or that allows users to play with a video feed like my Collage application does.

AVFoundation allows you to retrieve both a live preview as well as raw image buffer data from the camera, providing valuable tools in your development arsenal. To get started with AVFoundation, you need to build an Xcode project that uses quite a number of frameworks. You add these in (Project) > TARGETS > Build Phases > Link Binary With Libraries.

Here are the frameworks I recommend you initially add and the roles they play in your application. Only the first three are required, but I suggest you add all five because both Core Image and Quartz Core are typically used with AVFoundation video feeds.

- **AVFoundation**— Manage and play audio-visual media in your iOS applications: `<AVFoundation/AVFoundation.h>`.

From the Library of Eric Berna

- **CoreVideo**—Play and process movies with frame-by-frame control: `<CoreVideo/CoreVideo.h>`.
- **CoreMedia**—Handle time-based AV assets: `<CoreMedia/CoreMedia.h>`.
- **CoreImage**—Use pixel-accurate near-real-time image processing: `<CoreImage/CoreImage.h>`.
- **QuartzCore**—Add 2D graphics rendering support: `<QuartzCore/QuartzCore.h>`.

Requiring Cameras

If your application is built around a live camera feed, it should not install on any device that doesn't offer a built-in camera. You can mandate this by editing the Info.plist file, adding items to a `UIRequiredDeviceCapabilities` array. Add `still-camera` to assert that a camera, any camera, is available. You can narrow down device models and capabilities by specifying there must be an `auto-focus-camera`, a `front-facing-camera`, `camera-flash`, and `video-camera` as well. Required items are discussed further in Chapter 14, "Device Capabilities."

Querying and Retrieving Cameras

iOS devices may offer zero, one, or two cameras. For multi-camera devices, you can allow your user to select which camera to use, switching between the front and back. Listing 7-1 shows how to query the number of cameras, to check whether the front and back cameras are available on a given device, and how to return `AVCaptureDevice` instances for each.

Listing 7-1 **Cameras**

```
+ (int) numberOfCameras
{
    return [AVCaptureDevice devicesWithMediaType:AVMediaTypeVideo].count;
}

+ (BOOL) backCameraAvailable
{
    NSArray *videoDevices =
        [AVCaptureDevice devicesWithMediaType:AVMediaTypeVideo];
    for (AVCaptureDevice *device in videoDevices)
        if (device.position == AVCaptureDevicePositionBack) return YES;
    return NO;
}

+ (BOOL) frontCameraAvailable
{
    NSArray *videoDevices =
        [AVCaptureDevice devicesWithMediaType:AVMediaTypeVideo];
    for (AVCaptureDevice *device in videoDevices)
```

From the Library of Eric Berna

```
        if (device.position == AVCaptureDevicePositionFront) return YES;
    return NO;
}

+ (AVCaptureDevice *)backCamera
{
    NSArray *videoDevices =
        [AVCaptureDevice devicesWithMediaType:AVMediaTypeVideo];
    for (AVCaptureDevice *device in videoDevices)
        if (device.position == AVCaptureDevicePositionBack)
            return device;

    return [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo];
}

+ (AVCaptureDevice *)frontCamera
{
    NSArray *videoDevices =
        [AVCaptureDevice devicesWithMediaType:AVMediaTypeVideo];
    for (AVCaptureDevice *device in videoDevices)
        if (device.position == AVCaptureDevicePositionFront)
            return device;

    return [AVCaptureDevice defaultDeviceWithMediaType:AVMediaTypeVideo];
}
```

Establishing a Camera Session

Once you've selected a device to work with and retrieved its `AVCaptureDevice` instance, you can establish a new camera session. Here are the steps involved. A session includes creating a capture input, which grabs data from the selected camera, and a capture output, which sends buffer data to its delegate.

```
// Create the capture input
AVCaptureDeviceInput *captureInput =
    [AVCaptureDeviceInput deviceInputWithDevice:device error:&error];
if (!captureInput)
{
    NSLog(@"Error establishing device input: %@", error);
    return;
}

// Create capture output.
char *queueName = "com.sadun.tasks.grabFrames";
dispatch_queue_t queue = dispatch_queue_create(queueName, NULL);
AVCaptureVideoDataOutput *captureOutput =
```

From the Library of Eric Berna

```
    [[AVCaptureVideoDataOutput alloc] init];
captureOutput.alwaysDiscardsLateVideoFrames = YES;
[captureOutput setSampleBufferDelegate:self queue:queue];

// Establish settings
NSDictionary *settings = [NSDictionary
    dictionaryWithObject:
        [NSNumber numberWithUnsignedInt:kCVPixelFormatType_32BGRA]
    forKey:(NSString *)kCVPixelBufferPixelFormatTypeKey];
[captureOutput setVideoSettings:settings];

// Create a session
session = [[AVCaptureSession alloc] init];
[session addInput:captureInput];
[session addOutput:captureOutput];
```

Once a camera session is created, you can start it and stop it by sending `startRunning` and `stopRunning` method calls, as shown here:

```
[session startRunning]
```

When it is running, it sends regular buffer updates to its delegate. Here is where you can catch the raw data and convert it into a form that's better suited for image processing work. This method stores a `CIImage` instance as a retained property. Assuming you implement the capture routine in a helper class, you can retrieve this data from your application on demand.

```
- (void)captureOutput:(AVCaptureOutput *)captureOutput
    didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
    fromConnection:(AVCaptureConnection *)connection
{
    @autoreleasepool
    {
        // Transfer into a Core Video image buffer
        CVImageBufferRef imageBuffer =
            CMSampleBufferGetImageBuffer(sampleBuffer);

        // Create a Core Image result
        CFDictionaryRef attachments =
            CMCopyDictionaryOfAttachments(kCFAllocatorDefault,
                sampleBuffer, kCMAttachmentMode_ShouldPropagate);
        self.ciImage = [[CIImage alloc]
            initWithCVPixelBuffer:imageBuffer
            options:(__bridge_transfer NSDictionary *)attachments];
    }
}
```

From the Library of Eric Berna

Core Image's `CIImage` class is similar to `UIImage` and can be converted back and forth as needed. You can initialize a `UIImage` instance with a Core Image instance as follows:

```
UIImage *newImage = [UIImage imageWithCIImage:self.ciImage];
return newImage;
```

> **Note**
>
> At the time this book was being written, the `imageWithCIImage:` method was not yet functional. The workaround methods in the sample code that accompanies this chapter were added because of that.

The capture-output delegate method uses the Core Image version for several reasons. First, it's the natural API recipient for Core Video pixel buffers. Converting to a `CIImage` only takes a single call. Second, it allows you to integrate your captured image with Core Image filtering and feature detection such as face detection. You'll likely want to add filters outside the capture routine because these can slow down processing; you don't want to add filtering or detection to every frame you capture.

Switching Cameras

Once you've established a session, you can switch cameras without stopping or tearing down that session. Enter configuration mode, make your changes, and then commit the configuration updates, as shown here:

```
- (void) switchCameras
{
    if (![CameraImageHelper numberOfCameras] > 1) return;

    isUsingFrontCamera = !isUsingFrontCamera;
    AVCaptureDevice *newDevice = isUsingFrontCamera ?
        [CameraImageHelper frontCamera] : [CameraImageHelper backCamera];

    [session beginConfiguration];

    // Remove existing inputs
    for (AVCaptureInput *input in [session inputs])
        [session removeInput:input];

    // Change the input
    AVCaptureDeviceInput *captureInput =
        [AVCaptureDeviceInput deviceInputWithDevice: newDevice error:nil];
    [session addInput:captureInput];

    [session commitConfiguration];
}
```

From the Library of Eric Berna

Camera Previews

AVFoundation's `AVCaptureVideoPreviewLayer` offers a live image preview layer that you can embed into views. Preview layers are both simple to use and powerful in their effect. The following method creates a new preview layer and inserts it into a view, matching the view's frame. The video gravity here defaults to resize-aspect, which preserves the video's aspect ratio while fitting it within a given layer's bounds.

```
- (void) embedPreviewInView: (UIView *) aView
{
    if (!session) return;

    AVCaptureVideoPreviewLayer *preview =
        [AVCaptureVideoPreviewLayer layerWithSession: session];
    preview.frame = aView.bounds;
    preview.videoGravity = AVLayerVideoGravityResizeAspect;
    [aView.layer addSublayer: preview];
}
```

You can retrieve the preview layer from a view by searching for a layer of the right class.

```
- (AVCaptureVideoPreviewLayer *) previewInView: (UIView *) view
{
    for (CALayer *layer in view.layer.sublayers)
        if ([layer isKindOfClass:[AVCaptureVideoPreviewLayer class]])
            return (AVCaptureVideoPreviewLayer *)layer;

    return nil;
}
```

Laying Out a Camera Preview

One of the core challenges of working with live camera previews is keeping the video pointing up regardless of the device orientation. Although the camera connection does handle some orientation issues, it's easiest to work directly with the preview layer and transform it to the proper orientation.

```
- (void) layoutPreviewInView: (UIView *) aView
{
    AVCaptureVideoPreviewLayer *layer = [self previewInView:aView];
    if (!layer) return;

    UIDeviceOrientation orientation =
        [UIDevice currentDevice].orientation;
    CATransform3D transform = CATransform3DIdentity;
    if (orientation == UIDeviceOrientationPortrait) ;
    else if (orientation == UIDeviceOrientationLandscapeLeft)
        transform = CATransform3DMakeRotation(-M_PI_2, 0.0f, 0.0f, 1.0f);
    else if (orientation == UIDeviceOrientationLandscapeRight)
```

From the Library of Eric Berna

```
        transform = CATransform3DMakeRotation(M_PI_2, 0.0f, 0.0f, 1.0f);
    else if (orientation == UIDeviceOrientationPortraitUpsideDown)
        transform = CATransform3DMakeRotation(M_PI, 0.0f, 0.0f, 1.0f);

    layer.transform = transform;
    layer.frame = aView.frame;
}
```

EXIF

EXIF is the Exchangeable Image File Format created by the Japan Electronic Industries Development Association (JEIDA). It offers a standardized set of image annotations that include camera settings (such as shutter speed and metering mode), date and time information, and an image preview. Other metadata standards include IPTC (International Press Telecommunications Council) and XMP (Adobe's Extensible Metadata Platform).

Orientation is the only meta-information that is currently exposed directly through the `UIImage` class. This information is accessible through its `imageOrientation` property; as you'll see, it bears a correspondence with EXIF orientation, but the value uses a different numbering system.

Core Image and `CGImageSource` offer lower-level access to your image metadata. As Apple's Technical Q&A QA1622 points out, you must embed metadata yourself, employing whatever techniques, libraries, and standards you wish to use. Google "iOS EXIF projects" to find current development efforts at open-source repositories such as Google Code and github. Once the metadata is embedded, you can send your metadata-enriched image via e-mail, just as you would standard image data.

> **Note**
>
> You may also want to consider using the `ALAssetsLibrary`'s `writeImageToSavePhotosAlbum:metadata:completionBlock:` method.

Image Geometry

Geometry is the most difficult portion of working with AVFoundation camera feeds. Unlike using image pickers, which automatically handle matters for you, you must deal with raw image buffers. The camera you use (front or back) and the orientation of the device itself influence how the image data is oriented, as shown in Table 7-1.

Table 7-1 **Mapping Device Orientation to Image Orientation**

Orientation	Home Button	Camera	Natural Output	Mirrored
Portrait	Bottom	Front	Left Mirrored	Right
LandscapeLeft	Right	Front	Down Mirrored	Down
PortraitUpsideDown	Top	Front	Right Mirrored	Left
LandscapeRight	Left	Front	Up Mirrored	Up

From the Library of Eric Berna

Table 7-1 **Mapping Device Orientation to Image Orientation**

Orientation	Home Button	Camera	Natural Output	Mirrored
Portrait	Bottom	Back	Right	Left Mirrored
LandscapeLeft	Right	Back	Up	Up Mirrored
PortraitUpsideDown	Top	Back	Left	Right Mirrored
LandscapeRight	Left	Back	Down	Down Mirrored

The Natural Output column in Table 7-1 indicates the orientation produced by a given device orientation/camera combination. The orientation equivalent in Table 7-2 tells iOS how to display the data to restore it to the proper presentation. These constants enable `UIImage` instances to map the raw data (the F's shown in the first column) to the desired presentation (represented by the first F in the table).

Table 7-2 **Image Orientations**

Orientation	UIImageOrientation	EXIF Equivalent
XXXXXX XX XXXX XX XX	`UIImageOrientationUp` 0	Top Left 1
XX XX XXXX XX XXXXXX	`UIImageOrientationDown` 1	Bottom Right 3
XX XX XX XXXXXXXXXX	`UIImageOrientationLeft` 2	Right Top 6
XXXXXXXXXX XX XX XX	`UIImageOrientationRight` 3	Left Bottom 8
XXXXXX XX XXXX XX XX	`UIImageOrientationUpMirrored` 4	Top Right 2

From the Library of Eric Berna

Table 7-2 **Image Orientations**

Orientation	UIImageOrientation	EXIF Equivalent
XX XX XXXX XX XXXXXX	UIImageOrientationDownMirrored 5	Bottom Left 4
XXXXXXXXXX XX XX XX	UIImageOrientationLeftMirrored 6	Left Top 5
XX XX XX XXXXXXXXXX	UIImageOrientationRightMirrored 7	Right Bottom 7

The last column in Table 7-2 shows the EXIF orientation equivalent for each `UIImageOrientation`. This conversion becomes important when you are working with Core Image because CI uses EXIF not `UIImage` orientation.

Building Camera Helper

Recipe 7-5 combines all the methods described in this section into a simple helper class. This class offers all the basic features needed to query for cameras, establish a session, create a preview, and retrieve an image. To use this class, create a new instance, start running a session, and maybe embed a preview into one of your main views. Make sure to lay out your preview again any time you autorotate the interface, so the preview will remain properly oriented. You can do this in the `UIViewController` class's `viewDidLayoutSubviews` callback method.

Recipe 7-5 **Camera Image Helper**

```
@interface CameraImageHelper : NSObject
    <AVCaptureVideoDataOutputSampleBufferDelegate>

@property (strong)   AVCaptureSession *session;
@property (strong)   CIImage *ciImage;
@property (readonly) UIImage *currentImage;
@property (readonly) BOOL isUsingFrontCamera;

+ (int) numberOfCameras;
+ (BOOL) backCameraAvailable;
+ (BOOL) frontCameraAvailable;
+ (AVCaptureDevice *)backCamera;
+ (AVCaptureDevice *)frontCamera;
```

From the Library of Eric Berna

```
+ (id) helperWithCamera: (uint) whichCamera;

- (void) startRunningSession;
- (void) stopRunningSession;
- (void) switchCameras;

- (void) embedPreviewInView: (UIView *) aView;
- (AVCaptureVideoPreviewLayer *) previewInView: (UIView *) view;
- (void) layoutPreviewInView: (UIView *) aView;
@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 7 and open the project for this recipe.

Recipe: Adding a Core Image Filter

Core Image filters allow you to process images extremely quickly. Recipe 7-6 introduces CI filtering by applying a simple sepia filter to images captured from the onboard camera. This filter is not applied to a live video feed but rather to individual images.

This recipe is powered by a simple `NSTimer` that fires ten times a second. Each time it fires, the `snap:` method grabs the current `ciImage` from its helper (as defined in Recipe 7-5). If the user has enabled filtering (using a simple bar button toggle), the code creates a new sepia tone filter, sets the image as its input, adjusts the input intensity to 75%, and then retrieves and displays the output image.

iOS offers a limited set of CI filters, although these are expected to grow over time. You can query these by looking at the built-in set of filter names:

```
NSLog(@"%@", [CIFilter filterNamesInCategory: kCICategoryBuiltIn]);
```

Filters include color adjustments, geometric transforms, cropping, and compositing. Often you'll feed the results of one filter into the next. For example, to create a picture-in-picture effect, you'd scale and translate one image and then composite it over another.

Each filter uses a number of inputs, which include both source data and parameters. These vary by filter. Recipe 7-6 uses the sepia tone filter whose inputs are `inputImage` and `inputIntensity`. Either look up these inputs online in the latest Core Image Filter Reference document from Apple or query them directly from the filter by calling `inputKeys`, which returns an array of input parameter names.

You can push Recipe 7-6 from 10 frames a second (timer set to fire every 0.1 seconds) up to 30 frames a second (every 0.03 seconds) to create a more video-realistic speed. However, be aware a trade-off is involved between smooth presentation and processing overhead.

From the Library of Eric Berna

Recipe 7-6 **Adding a Simple Core Image Filter**

```
@implementation TestBedViewController

// Switch between cameras
- (void) switch: (id) sender
{
    [helper switchCameras];
}

// Enable/disable the sepia filter
- (void) toggleFilter: (id) sender
{
    useFilter = !useFilter;
}

// Grab an image 10 times a second, optionally applying the filter,
// and display the results
- (void) snap: (NSTimer *) timer
{
    if (useFilter)
    {
        // Create the sepia filter at 75%
        CIFilter *sepiaFilter =
            [CIFilter filterWithName:@"CISepiaTone"
                 keysAndValues: @"inputImage", helper.ciImage, nil];
        [sepiaFilter setDefaults];
        [sepiaFilter setValue:[NSNumber numberWithFloat:0.75f]
             forKey:@"inputIntensity"];

        // Apply the filter and display the results
        CIImage *sepiaImage = [sepiaFilter valueForKey:kCIOutputImageKey];
        if (sepiaImage)
            imageView.image = [UIImage imageWithCIImage:sepiaImage];
    }
    else
        imageView.image = [UIImage imageWithCIImage:helper.ciImage];
}

- (void) loadView
{
    [super loadView];

    // Add basic functionality
    if ([CameraImageHelper numberOfCameras] > 1)
        self.navigationItem.leftBarButtonItem =
            BARBUTTON(@"Switch", @selector(switch:));
```

From the Library of Eric Berna

```
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Toggle Filter", @selector(toggleFilter:));

    // Create an image view to host the results
    imageView = [[UIImageView alloc]
        initWithFrame:[[UIScreen mainScreen] bounds]];
    imageView.contentMode = UIViewContentModeScaleAspectFill;
    RESIZABLE(imageView);
    [self.view addSubview:imageView];

    // Establish a new camera session
    helper = [CameraImageHelper helperWithCamera:kCameraFront];
    [helper startRunningSession];

    // Update ten times a second
    [NSTimer scheduledTimerWithTimeInterval:0.1f target:self
        selector:@selector(snap:) userInfo:nil repeats:YES];
}
@end
```

> **Get This Recipe's Code**
>
> To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from the book, go to the folder for Chapter 7 and open the project for this recipe.

Recipe: Core Image Face Detection

Face detection is one of iOS 5's newer and splashier features. It's something that is both very easy to implement and quite hard to get right, and that all comes down to basic geometry. Retrieving the feature set involves creating a new face-type detector and querying it for the detected features. This works by returning an array of zero or more `CIFaceFeature` objects, corresponding to the number of faces found in the scene.

```
- (NSArray *) featuresInImage
{
    NSDictionary *detectorOptions = [NSDictionary
        dictionaryWithObject:CIDetectorAccuracyLow
        forKey:CIDetectorAccuracy];

    CIDetector *detector = [CIDetector
        detectorOfType:CIDetectorTypeFace context:nil
        options:detectorOptions];

    uint orientation = detectorEXIF(helper.isUsingFrontCamera, NO);
    NSDictionary *imageOptions =
        [NSDictionary dictionaryWithObject:
            [NSNumber numberWithInt:orientation]
```

From the Library of Eric Berna

```
            forKey:CIDetectorImageOrientation];
    return [detector featuresInImage:ciImage options:imageOptions];
}
```

What makes this a particularly hard task comes down to several issues. The detector depends on knowing the proper orientation in order to interpret the CI image passed to it. It cannot detect faces that are sideways or upside down. That's why this method includes a call to a `detectorEXIF` function. The problem is that, at least at the time this book was being written, the required detector orientation doesn't always match the real-world image orientation, causing workarounds like this:

```
uint detectorEXIF(BOOL isUsingFrontCamera, BOOL shouldMirrorFlip)
{
    if (isUsingFrontCamera || deviceIsLandscape())
        return currentEXIFOrientation(isUsingFrontCamera,
            shouldMirrorFlip);

    // Only back camera portrait or upside down here.
    // Detection happens but the geometry is messed.
    int orientation = currentEXIFOrientation(!isUsingFrontCamera,
        shouldMirrorFlip);
    return orientation;
}
```

As you can tell, this is hacky, buggy, and a disaster waiting to happen.

The second problem lies in the geometry. Apple has not provided APIs that map coordinates into images while respecting the image orientation information. The coordinate (0,0) may refer to the top left, bottom left, top right, or bottom right. What's more, mirroring means the coordinate may be offset from the left side or the right, from the bottom or the top. The math is left up to you. Enter Listing 7-2. These methods help you convert point and rectangle results from their raw data coordinates into normalized image coordinates.

The `ExifOrientation` enumeration used here is not built into iOS. I defined it to match the natural EXIF values shown in Table 7-2. Notice that there are no cases for `kRightTop` and `kLeftBottom`. These are the two EXIF orientations not currently supported by the Core Image detector. Should Apple update their implementation, you will want to add cases to support both.

Listing 7-2 **Converting Geometry from EXIF to Image Coordinates**

```
CGPoint pointInEXIF(ExifOrientation exif, CGPoint aPoint, CGRect rect)
{
    switch(exif)
    {
        case kTopLeft:
            return CGPointMake(aPoint.x,
                rect.size.height - aPoint.y);
```

From the Library of Eric Berna

```
        case kTopRight:
            return CGPointMake(rect.size.width - aPoint.x,
                rect.size.height - aPoint.y);
        case kBottomRight:
             return CGPointMake(rect.size.width - aPoint.x,
                aPoint.y);
        case kBottomLeft:
            return CGPointMake(aPoint.x, aPoint.y);

        case kLeftTop:
            return CGPointMake(aPoint.y, aPoint.x);
        case kRightBottom:
            return CGPointMake(rect.size.width - aPoint.y,
                rect.size.height - aPoint.x);

        default:
            return aPoint;
    }
}

CGSize sizeInEXIF(ExifOrientation exif, CGSize aSize)
{
    switch(exif)
    {
        case kTopLeft:
        case kTopRight:
        case kBottomRight:
        case kBottomLeft:
            return aSize;

        case kLeftTop:
        case kRightBottom:
            return CGSizeMake(aSize.height, aSize.width);
    }
}

CGRect rectInEXIF(ExifOrientation exif, CGRect inner, CGRect outer)
{
    CGRect rect;
    rect.origin = pointInEXIF(exif, inner.origin, outer);
    rect.size = sizeInEXIF(exif, inner.size);

    switch(exif)
```

From the Library of Eric Berna

```
    {
        case kTopLeft:
            rect = CGRectOffset(rect, 0.0f, -inner.size.height);
            break;
        case kTopRight:
            rect = CGRectOffset(rect, -inner.size.width,
                -inner.size.height);
            break;
        case kBottomRight:
            rect = CGRectOffset(rect, -inner.size.width, 0.0f);
            break;
        case kBottomLeft:
            break;

        case kLeftTop:
            break;
        case kRightBottom:
            rect = CGRectOffset(rect, -inner.size.width,
                -inner.size.height);
            break;
        default:
            break;
    }

    return rect;
}
```

Recipe 7-7 demonstrates how to perform facial detection in your application. This method retrieves detected features, outlines the detected face bounds with a shaded rectangle, and highlights three specific features (left eye, right eye, and mouth) with circles. Figure 7-6 shows the detection in action.

Each member of the `CIFaceFeature` array returned by the detection can report several features of geometric interest. These include the rectangular bounds around the face, and positions for the left and right eyes and the mouth. These subfeatures are not always created, so you should first test whether positions are available before using them. Recipe 7-7 demonstrates both test and use.

Recipe 7-7 also handles several workarounds due to Core Image's portrait EXIF eccentricities. Should Apple update the way Core Image detects faces by the time this book is released, please adjust the code accordingly for the two remaining orientations.

From the Library of Eric Berna

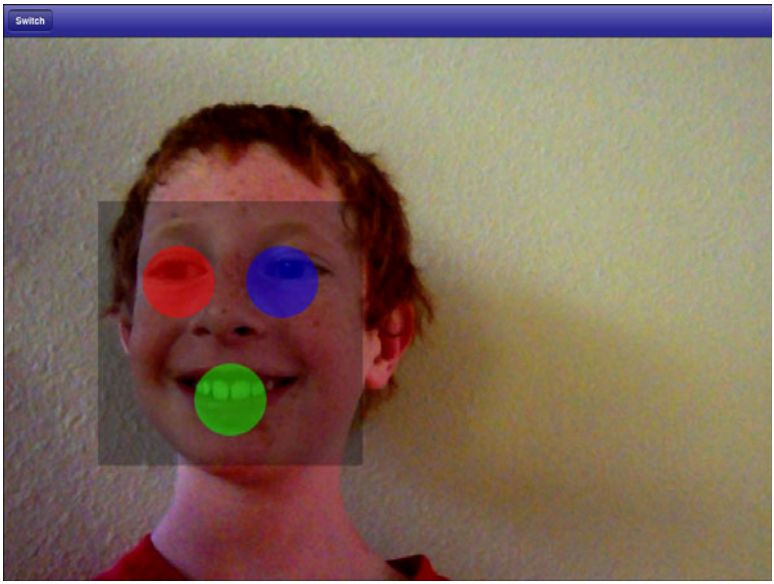


Figure 7-6 Automatically detecting faces and their features.

Recipe 7-7 **Detecting Faces**

```
- (void) snap: (NSTimer *) timer
{
    ciImage = helper.ciImage;
    UIImage *baseImage = [UIImage imageWithCIImage:ciImage];
    CGRect imageRect = (CGRect){.size = baseImage.size};

    NSDictionary *detectorOptions = [NSDictionary
        dictionaryWithObject:CIDetectorAccuracyLow
        forKey:CIDetectorAccuracy];

    CIDetector *detector = [CIDetector
        detectorOfType:CIDetectorTypeFace
        context:nil options:detectorOptions];

    ExifOrientation detectOrientation =
        detectorEXIF(helper.isUsingFrontCamera, NO);

    NSDictionary *imageOptions = [NSDictionary
        dictionaryWithObject:[NSNumber numberWithInt:detectOrientation]
        forKey:CIDetectorImageOrientation];
    NSArray *features =
        [detector featuresInImage:ciImage options:imageOptions];
```

From the Library of Eric Berna

```
UIGraphicsBeginImageContext(baseImage.size);
[baseImage drawInRect:imageRect];

for (CIFaceFeature *feature in features)
{
    CGRect rect = rectInEXIF(detectOrientation,
        feature.bounds, imageRect);
    if (deviceIsPortrait() && helper.isUsingFrontCamera) // workaround
    {
        rect.origin = CGPointFlipHorizontal(rect.origin, imageRect);
        rect.origin =
            CGPointOffset(rect.origin, -rect.size.width, 0.0f);
    }

    [[[UIColor blackColor] colorWithAlphaComponent:0.3f] set];
    UIBezierPath *path = [UIBezierPath bezierPathWithRect:rect];
    [path fill];

    if (feature.hasLeftEyePosition)
    {
        [[[UIColor redColor] colorWithAlphaComponent:0.5f] set];
        CGPoint position = feature.leftEyePosition;
        CGPoint pt = pointInEXIF(detectOrientation, position,
            imageRect);

        if (deviceIsPortrait() && helper.isUsingFrontCamera)
            pt = CGPointFlipHorizontal(pt, imageRect); // workaround

        UIBezierPath *path = [UIBezierPath bezierPathWithArcCenter:pt
            radius:30.0f startAngle:0.0f endAngle:2 * M_PI
            clockwise:YES];
        [path fill];
    }

    if (feature.hasRightEyePosition)
    {
        [[[UIColor blueColor] colorWithAlphaComponent:0.5f] set];
        CGPoint position = feature.rightEyePosition;
        CGPoint pt = pointInEXIF(detectOrientation, position,
            imageRect);
        if (deviceIsPortrait() && helper.isUsingFrontCamera)
            pt = CGPointFlipHorizontal(pt, imageRect); // workaround

        UIBezierPath *path = [UIBezierPath bezierPathWithArcCenter:pt
            radius:30.0f startAngle:0.0f endAngle:2 * M_PI
            clockwise:YES];
```

From the Library of Eric Berna

```
            [path fill];
        }

        if (feature.hasMouthPosition)
        {
            [[[UIColor greenColor] colorWithAlphaComponent:0.5f] set];
            CGPoint position = feature.mouthPosition;
            CGPoint pt = pointInEXIF(detectOrientation, position,
                imageRect);
            if (deviceIsPortrait() && helper.isUsingFrontCamera)
                pt = CGPointFlipHorizontal(pt, imageRect); // workaround

            UIBezierPath *path = [UIBezierPath bezierPathWithArcCenter:pt
                radius:30.0f startAngle:0.0f endAngle:2 * M_PI
                clockwise:YES];

            [path fill];
        }

    }

    imageView.image = UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();
}
```

> **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 7 and open the project for this recipe.

Extracting Faces

Recipe 7-7 showed how to use face detection to add features in and around a given face. Surprisingly, that's not always why you want to use face detection. More often, you want to fetch the data at the face's coordinates and reuse that data in another context.

Consider Figure 7-7. The application that created this presentation used face detection to extract face data and reposition it behind the cutout in the two-dollar bill. It doesn't matter where the person's face is in-frame. So long as it's found, the Core Image feature's bounds allow the application to extract the image and use it in some other way.

Extracting image data forms the opposite of the approach used in Recipe 7-7. Instead of drawing on top of an existing image, the following method returns an image cut from the bounds that you pass to it. From there, you can reposition it or process it further for custom recognition and so on.

From the Library of Eric Berna



Figure 7-7 Extracting a face from an image allows you to reposition just the face to any view location.

```
- (UIImage *) subImageWithBounds:(CGRect) rect
{
    UIGraphicsBeginImageContext(rect.size);

    CGRect destRect = CGRectMake(-rect.origin.x, -rect.origin.y,
        self.image.size.width, self.image.size.height);
    [self.image drawInRect:destRect];

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

    return newImage;
}
```

Recipe: Working with Bitmap Representations

Although Cocoa Touch provides excellent resolution-independent tools for working with many images, there are times you need to reach down to the bits that underlie a picture and access data on a bit-by-bit basis. For example, you might apply edge-detection or blurring routines that calculate their results by convolving matrices against actual byte values.

Figure 7-8 shows the result of Canny edge detection on an iPhone image. The Canny operator in its most basic form is one of the first algorithms taught in image processing classes. Canny edge detection combines a smoothing filter with a search for maximal changes. Edges, where adjacent pixels demonstrate the greatest difference in intensity, produce the highest response to the mask. The version used to produce the image shown here uses a hardwired 3×3 mask—that is, the simplest possible version of the effect.

From the Library of Eric Berna

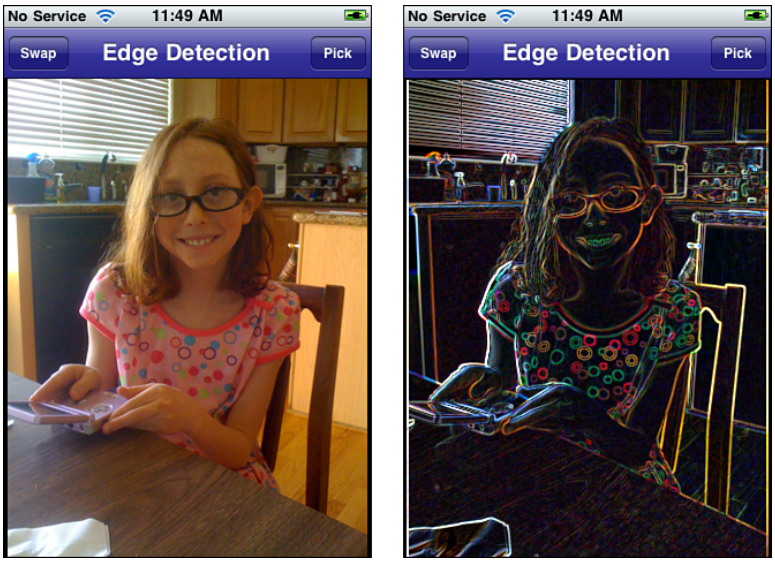


Figure 7-8 Applying edge detection to an image produces a result that outlines areas where byte values experience the greatest changes.

As a rule, you really don't want to do manual image processing on your entire image. Recipe 7-8 will, because it's a good example of how heavy-duty processing can block your code and require a secondary thread. For most image processing, try to use Core Image where possible, or use tricks to apply processing only to a small part of the live image data you have available. For example, you might only want to sample every second or fifth or tenth line, or you might look only at the center of the live view.

Manual image processing is expensive in both processor time and memory overhead. Use it cautiously and wisely in your real-world deployment.

Drawing into a Bitmap Context

To get started with image processing, draw an image into a bitmap context and then retrieve bytes as a `uint8 *` buffer. (The uint8 type is equivalent to unsigned char, namely 8 unsigned bits.) This code draws the image and retrieves the bits from the context. This implementation is from a small `UIImage` category, which is why the code refers to `self` as the image.

```
- (uint8 *) createBitmap
{
    // Create bitmap data for the given image
    CGContextRef context = CreateARGBBitmapContext(self.size);
    if (context == NULL) return NULL;
```

From the Library of Eric Berna

```
    CGRect rect =
        CGRectMake(0.0f, 0.0f, self.size.width, self.size.height);
    CGContextDrawImage(context, rect, self.CGImage);
    uint8 *data = CGBitmapContextGetData(context);
    CGContextRelease(context);

    return data;
}
```

This routine relies on a special bitmap context that allocates memory for the bitmap data. Here is the function that creates that context. It produces an ARGB bitmap context (Alpha-Red-Green-Blue), one byte per channel, 256 levels per unsigned byte.

```
CGContextRef CreateARGBBitmapContext (CGSize size)
{
    // Create the new color space
    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    if (colorSpace == NULL)
    {
        fprintf(stderr, "Error allocating color space\n");
        return NULL;
    }

    // Allocate memory for the bitmap data
    void *bitmapData = malloc(size.width * size.height * 4);
    if (bitmapData == NULL)
    {
        fprintf (stderr, "Error: Memory not allocated!");
        CGColorSpaceRelease(colorSpace);
        return NULL;
    }

    // Build an 8-bit per channel context
    CGContextRef context = CGBitmapContextCreate (bitmapData,
        size.width, size.height, 8, size.width * 4, colorSpace,
        kCGImageAlphaPremultipliedFirst);
    CGColorSpaceRelease(colorSpace );
    if (context == NULL)
    {
        fprintf (stderr, "Error: Context not created!");
        free (bitmapData);
        return NULL;
    }

    return context;
}
```

From the Library of Eric Berna

Once the image bytes are available, you can access them directly. The following functions return offsets for any point `(x,y)` inside an ARGB bitmap using width `w`. The height is not needed for these calculations; the width of each row allows you to determine a two-dimensional point in what is really a one-dimensional buffer. Notice how the data is interleaved. Each 4-byte sequence contains a level for alpha, red, green, and then blue. Each byte ranges from 0 (0%) to 255 (100%). Convert this to a float and divide by 255.0 to retrieve the ARGB value.

```
NSUInteger alphaOffset(NSUInteger x, NSUInteger y, NSUInteger w)
    {return y * w * 4 + x * 4 + 0;}
NSUInteger redOffset(NSUInteger x, NSUInteger y, NSUInteger w)
    {return y * w * 4 + x * 4 + 1;}
NSUInteger greenOffset(NSUInteger x, NSUInteger y, NSUInteger w)
    {return y * w * 4 + x * 4 + 2;}
NSUInteger blueOffset(NSUInteger x, NSUInteger y, NSUInteger w)
    {return y * w * 4 + x * 4 + 3;}
```

Applying Image Processing

It's relatively easy then to convolve an image by recovering its bytes and applying some image-processing algorithm. This routine uses the basic Canny edge detection mentioned earlier. It calculates both the vertical and horizontal edge results for each color channel, and then scales the sum of those two results into a single value that falls within `[0, 255]`. The output alpha value preserves the original level.

```
- (UIImage *) convolveImageWithEdgeDetection
{
    // Dimensions
    int theheight = floor(self.size.height);
    int thewidth =  floor(self.size.width);

    // Get input and create output bits
    uint8 *inbits = (UInt8 *)[self createBitmap];
    uint8 *outbits =
        (uint8 *)malloc(theheight * thewidth * 4);

    // Basic Canny Edge Detection
    int matrix1[9] = {-1, 0, 1, -2, 0, 2, -1, 0, 1};
    int matrix2[9] = {-1, -2, -1, 0, 0, 0, 1, 2, 1};

    int radius = 1;

    // Iterate through each available pixel (leaving a radius-sized
    // boundary)
    for (int y = radius; y < (theheight - radius); y++)
        for (int x = radius; x < (thewidth - radius); x++)
```

From the Library of Eric Berna

```
        {
            int sumr1 = 0, sumr2 = 0;
            int sumg1 = 0, sumg2 = 0;
            int sumb1 = 0, sumb2 = 0;
            int offset = 0;
            for (int j = -radius; j <= radius; j++)
                for (int i = -radius; i <= radius; i++)
                {
                    sumr1 += inbits[redOffset(x+i, y+j, thewidth)] *
                        matrix1[offset];
                    sumr2 += inbits[redOffset(x+i, y+j, thewidth)] *
                        matrix2[offset];

                    sumg1 += inbits[greenOffset(x+i, y+j, thewidth)] *
                        matrix1[offset];
                    sumg2 += inbits[greenOffset(x+i, y+j, thewidth)] *
                        matrix2[offset];

                    sumb1 += inbits[blueOffset(x+i, y+j, thewidth)] *
                        matrix1[offset];
                    sumb2 += inbits[blueOffset(x+i, y+j, thewidth)] *
                        matrix2[offset];
                    offset++;
                }

            // Assign the outbits
            int sumr = MIN(((ABS(sumr1) + ABS(sumr2)) / 2), 255);
            int sumg = MIN(((ABS(sumg1) + ABS(sumg2)) / 2), 255);
            int sumb = MIN(((ABS(sumb1) + ABS(sumb2)) / 2), 255);

            outbits[redOffset(x, y, thewidth)] = (uint8) sumr;
            outbits[greenOffset(x, y, thewidth)] = (uint8)
            sumg;
            outbits[blueOffset(x, y, thewidth)] = (uint8) sumb;
            outbits[alphaOffset(x, y, thewidth)] =
            (uint8) inbits[alphaOffset(x, y, thewidth)];
        }

    // Release the original bitmap. imageWithBits frees outbits
    free(inbits);

    return [UIImage imageWithBits:outbits
        withSize:CGSizeMake(thewidth, theheight)];
}
```

From the Library of Eric Berna

The `imageWithBits:withSize` method allows you to convert bits into a bitmap context and then pull that data into a new image.

```
+ (UIImage *) imageWithBits: (uint8 *) bits
    withSize: (CGSize) size
{
    // Create a color space
    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    if (colorSpace == NULL)
    {
        fprintf(stderr, "Error allocating color space\n");
        free(bits);
        return nil;
    }

    // Create the bitmap context
    CGContextRef context = CGBitmapContextCreate(bits, size.width,
        size.height, 8, size.width * 4, colorSpace,
        kCGImageAlphaPremultipliedFirst);
    if (context == NULL)
    {
        fprintf (stderr, "Error: Context not created!");
        free (bits);
        CGColorSpaceRelease(colorSpace );
        return nil;
    }

    // Create the image ref
    CGColorSpaceRelease(colorSpace );
    CGImageRef imageRef = CGBitmapContextCreateImage(context);
    free(CGBitmapContextGetData(context)); // This does the free
    CGContextRelease(context);

    // Return image using image ref
    UIImage *newImage = [UIImage imageWithCGImage:imageRef];
    CFRelease(imageRef);

    return newImage;
}
```

Image Processing Realities

iOS devices are not number-crunching powerhouses. Routines outside of Core Image, such as the Canny edge detection shown in this section, can slow down applications significantly. Use them judiciously. Recipe 7-8 demonstrates how to balance image-processing demands with iOS limitations. It follows three main rules of iOS implementation:

From the Library of Eric Berna

- Provide meaningful feedback to the user when dealing with unavoidable delays.
- Perform processor-heavy functionality on a secondary thread (here via an `NSOperation`).
- Only ever perform GUI updates on the main thread.

In this recipe, an alert view displays throughout the processing interval. Even on a very fast and capable iPad 2, the manual edge detection processing routine can take several seconds. The alert is dismissed and the processed image displayed once the (blocking) image convolution finishes. This is done so on the main queue, allowing the updates to safely affect the GUI.

Recipe 7-8 **Image Processing with a Heads-Up Alert Display**

```
- (void) process: (id) sender
{
    // Prepare the interface
    self.navigationItem.rightBarButtonItem = nil;
    UIAlertView *alertView =
        [[UIAlertView alloc]
            initWithTitle:@"\n\nProcessing\nPlease wait."
            message:nil delegate:self cancelButtonTitle:nil
            otherButtonTitles:nil];
    [alertView show];

    // Create a background queue for processing
    NSOperationQueue *queue = [[NSOperationQueue alloc] init];
    [queue addOperationWithBlock:
     ^{
         UIImage *theImage = helper.currentImage;
         UIImage *processed =
            [theImage convolveImageWithEdgeDetection];

         // Update the image on the main thread using the main queue
         [[NSOperationQueue mainQueue] addOperationWithBlock:^{
             imageView.image = processed;
             [alertView dismissWithClickedButtonIndex:-1 animated:YES];
             self.navigationItem.rightBarButtonItem =
                 BARBUTTON(@"Process", @selector(process:));
         }];
     }];
}
```

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 7 and open the project for this recipe.

From the Library of Eric Berna

Recipe: Sampling a Live Feed

You can adapt the bitmap access introduced in Recipe 7-8 to create real-time responses for the user. Recipe 7-9 samples just the center of each image to retrieve a prevailing color. It extracts a 128-by-128-pixel swatch, and then performs some basic statistical image processing on it. The navigation bar's color continuously updates to display this most popular color from the center of the sampled feed.

This method converts each pixel from RGB to HSB, allowing the algorithm to retrieve a characteristic hue. It increases the histogram bucket for the hue and accumulates the saturation and brightness—that accumulation will eventually be divided by the number of samples in that bucket to create an average saturation and brightness for a given hue.

The most popular hue in the sample wins. The hue with the greatest hit count (that is, the mode) forms the basis of the new color, with the average saturation and brightness rounding out the creation. That color is assigned to tint the navigation bar.

Recipe 7-9 **Analyzing Bitmap Samples**

```
#define SAMPLE_LENGTH 128
- (void) pickColor
{
    // Retrieve the center 128x128 sample as bits
    UIImage *currentImage = helper.currentImage;
    CGRect sampleRect =
        CGRectMake(0.0f, 0.0f, SAMPLE_LENGTH, SAMPLE_LENGTH);
    sampleRect = CGRectCenteredInRect(sampleRect,
        (CGRect){.size = currentImage.size});
    UIImage *sampleImage =
        [currentImage subImageWithBounds:sampleRect];
    unsigned char *bits = [sampleImage createBitmap];

    // Create the histogram and average sampling buckets
    int bucket[360];
    CGFloat sat[360], bri[360];

    for (int i = 0; i < 360; i++)
    {
        bucket[i] = 0; // histogram sample
        sat[i] = 0.0f; // average saturation
        bri[i] = 0.0f; // average brightness
    }

    // Iterate over each sample pixel, accumulating hsb info
    for (int y = 0; y < SAMPLE_LENGTH; y++)
        for (int x = 0; x < SAMPLE_LENGTH; x++)
        {
            CGFloat r = ((CGFloat)bits[redOffset(x, y,
```

From the Library of Eric Berna

```
                SAMPLE_LENGTH)] / 255.0f);
            CGFloat g = ((CGFloat)bits[greenOffset(x, y,
                SAMPLE_LENGTH)] / 255.0f);
            CGFloat b = ((CGFloat)bits[blueOffset(x, y,
                SAMPLE_LENGTH)] / 255.0f);

           // Convert from RGB to HSV
            CGFloat h, s, v;
            rgbtohsb(r, g, b, &h, &s, &v);
            int hue = (hue > 359.0f) ? 0 : (int) h;

            // Collect metrics on a per-hue basis
            bucket[hue]++;
            sat[hue] += s;
            bri[hue] += v;
        }

    // Retrieve the hue mode
    int max = -1;
    int maxVal = -1;
    for (int i = 0; i < 360; i++)
    {
        if (bucket[i]  > maxVal)
        {
            max = i;
            maxVal = bucket[i];
        }
    }

    // Create a color based on the mode hue, average sat & bri
    float h = max / 360.0f;
    float s = sat[max]/maxVal;
    float br = bri[max]/maxVal;

    CGFloat red, green, blue;
    hsbtorgb((CGFloat) max, s, br, &red, &green, &blue);

    UIColor *hueColor = [UIColor colorWithHue:h saturation:s
        brightness:br alpha:1.0f];

    // Display the selected hue
    self.navigationController.navigationBar.tintColor = hueColor;

    free(bits);
}
```

From the Library of Eric Berna

> **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 7 and open the project for this recipe.

Converting to HSB

Recipe 7-8 depends on converting its colors to HSB from RGB using the following function, which is adapted from my Ph.D. advisor Jim Foley's seminal textbook on computer graphics:

```
void rgbtohsb(CGFloat r, CGFloat g, CGFloat b, CGFloat *pH,
    CGFloat *pS, CGFloat *pV)
{
    CGFloat h,s,v;

    // From Foley and Van Dam
    CGFloat max = MAX(r, MAX(g, b));
    CGFloat min = MIN(r, MIN(g, b));

    // Brightness
    v = max;

    // Saturation
    s = (max != 0.0f) ? ((max - min) / max) : 0.0f;

    if (s == 0.0f) {
        // No saturation, so undefined hue
        h = 0.0f;
    } else {
        // Determine hue distances from...
        CGFloat rc = (max - r) / (max - min); // from red
        CGFloat gc = (max - g) / (max - min); // from green
        CGFloat bc = (max - b) / (max - min); // from blue

        if (r == max) h = bc - gc; // between yellow & magenta
        else if (g == max) h = 2 + rc - bc; // between cyan & yellow
        else h = 4 + gc - rc; // between magenta & cyan

        h *= 60.0f; // Convert to degrees
        if (h < 0.0f) h += 360.0f; // Make non-negative
    }

    if (pH) *pH = h;
    if (pS) *pS = s;
    if (pV) *pV = v;
}
```

From the Library of Eric Berna

Recipe: Building Thumbnails from Images

Thumbnails play an important role in any application that uses images. Often you need to resize an image to fit into a smaller space. Sure, you can load up a `UIImageView` with the full-sized original and resize its frame, but you can save a lot of memory by redrawing that image into fewer bytes. Thumbnails can use one of three approaches, as demonstrated in Figure 7-9:

- Resize the image while retaining its proportions, fitting it so every part of the image remains visible. Depending on the image's aspect ratio, you'll need to either letterbox or pillarbox some extra area, matting the image with transparent pixels.
- Punch out part of the image to match the available space. The example in Figure 7-9 chooses a centered subimage and crops any elements that fall outside the pixel area.
- Fill the image by matching the height and width to the available space. Every pixel gets used, but the image will get cropped, either horizontally or vertically. This corresponds to the full-screen film presentation shown on nonwidescreen TVs, which tend to lose detail at either side of the movie.

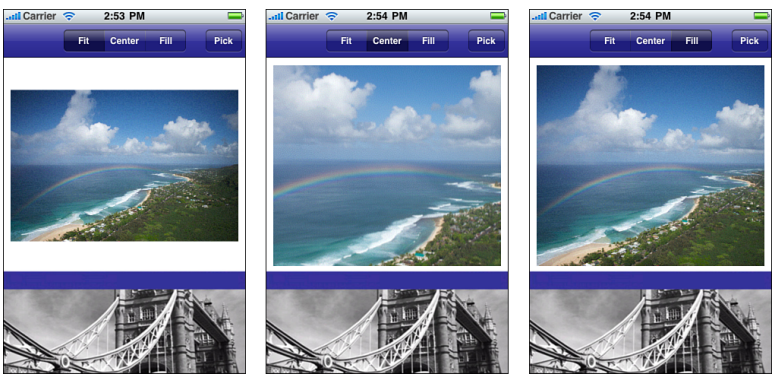


Figure 7-9 These screenshots represent three ways to create image thumbnails. Fitting (left) preserves original aspect ratios, padding the image as needed with extra space. Centering (center) uses the original image pixels, cropping from the center out. Filling (right) ensures that every available pixel is filled, cropping only those portions that fall outside the frame.

Recipe 7-10 shows how to create these three thumbnail effects using a `UIImage` category. The methods in this code allow you to pass a target size. They return a new thumbnail using the fit, center, or fill technique, respectively.

From the Library of Eric Berna

Recipe 7-10 **Creating Thumbnails**

```
// Calculate a size that fits in another size while retaining its
// original proportions
CGSize CGSizeFitInSize(CGSize sourceSize, CGSize destSize)
{
        CGFloat destScale;
        CGSize newSize = sourceSize;

        if (newSize.height && (newSize.height > destSize.height))
        {
               destScale = destSize.height / newSize.height;
               newSize.width *= destScale;
               newSize.height *= destScale;
        }

        if (newSize.width && (newSize.width >= destSize.width))
        {
               destScale = destSize.width / newSize.width;
               newSize.width *= destScale;
               newSize.height *= destScale;
        }

        return newSize;
}

// Proportionately resize, completely fit in view, no cropping
- (UIImage *) fitInSize: (CGSize) viewsize
{
    // calculate the fitted size
    CGSize size = CGSizeFitInSize(self.size, viewsize);

    UIGraphicsBeginImageContext(viewsize);

    // Calculate any matting needed for image spacing
    float dwidth = (viewsize.width - size.width) / 2.0f;
    float dheight = (viewsize.height - size.height) / 2.0f;

    CGRect rect = CGRectMake(dwidth, dheight, size.width, size.height);
    [self drawInRect:rect];

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

    return newImage;
}


// No resize, may crop
```

From the Library of Eric Berna

```
- (UIImage *) centerInSize: (CGSize) viewsize
{
    CGSize size = self.size;

    UIGraphicsBeginImageContext(viewsize);

    // Calculate the offset to ensure that the image center is set
    // to the view center
    float dwidth = (viewsize.width - size.width) / 2.0f;
    float dheight = (viewsize.height - size.height) / 2.0f;

    CGRect rect = CGRectMake(dwidth, dheight, size.width, size.height);
    [self drawInRect:rect];

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

    return newImage;
}

// Fill every view pixel with no black borders,
// resize and crop if needed
- (UIImage *) fillSize: (CGSize) viewsize
{
    CGSize size = self.size;

    // Choose the scale factor that requires the least scaling
    CGFloat scalex = viewsize.width / size.width;
    CGFloat scaley = viewsize.height / size.height;
    CGFloat scale = MAX(scalex, scaley);

    UIGraphicsBeginImageContext(viewsize);

    CGFloat width = size.width * scale;
    CGFloat height = size.height * scale;

    // Center the scaled image
    float dwidth = ((viewsize.width - width) / 2.0f);
    float dheight = ((viewsize.height - height) / 2.0f);

    CGRect rect = CGRectMake(dwidth, dheight,
        size.width * scale, size.height * scale);
    [self drawInRect:rect];

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

    return newImage;
}
```

From the Library of Eric Berna

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 7 and open the project for this recipe.

Taking View-based Screenshots

At times you need to take a shot of a view or window in its current state. Listing 7-3 details how you can draw views into image contexts and retrieve `UIImage` instances. This code works by using Quartz Core's `renderInContext` call for `CALayer` instances. It produces a screenshot not only of the view but of all the subviews that view owns.

There are, of course, limits. You cannot screenshot the entire window (the status bar will be missing in action) and, using this code in particular, you cannot screenshot videos or the camera previews. OpenGL ES views also may not be captured.

Listing 7-3 **Screenshotting a View**

```
UIImage *imageFromView(UIView *theView)
{
        UIGraphicsBeginImageContext(theView.frame.size);
        [theView.layer renderInContext:UIGraphicsGetCurrentContext()];
        UIImage *newImage = UIGraphicsGetImageFromCurrentImageContext();
        UIGraphicsEndImageContext();
        return newImage;
}

UIImage *screenShot()
{
        UIWindow *window = [[UIApplication sharedApplication] keyWindow];
        return imageFromView(window);
}
```

Drawing into PDF Files

You can draw directly into PDF documents, just as if you were drawing into an image context. As you can see in Listing 7-4, you can use Quartz drawing commands such as `UIImage`'s `drawInRect:` and `CALayer`'s `renderInContext:` methods, along with your full arsenal of other Quartz functions.

You'll typically draw each image into a new PDF page. You must create at least one page in your PDF document, as shown in this listing. In addition to simple rendering, you can add live links to your PDFs using functions such as `UIGraphicsSetPDFContextURLForRect()`, which links to an external URL from the rectangle you define, and `UIGraphicsSetPDFContextDestinationForRect()`, which links

From the Library of Eric Berna

internally within your document. Create your destination points using `UIGraphicsAddPDFContextDestinationAtPoint()`.

Each PDF document is built with a custom dictionary. Pass `nil` if you want to skip the dictionary, or assign values for Author, Creator, Title, Password, AllowsPrinting, and so forth. The keys for this dictionary are listed in Apple's `CGPDFContext` reference documentation.

Listing 7-4 **Drawing View Contents into a PDF File**

```
- (void) saveImage: (UIImage *) image
    toPDFFileNamed: (NSString *) filename
{
    NSString *pdfPath = [NSHomeDirectory()
        stringByAppendingFormat:@"/Documents/%
    CGRect theBounds = CGRectMake(0.0f, 0.0f,
        image.size.width, image.size.height);
    UIGraphicsBeginPDFContextToFile(pdfPath, theBounds, nil);
    {
        UIGraphicsBeginPDFPage();
        [image drawInRect:theBounds];
    }
    UIGraphicsEndPDFContext();
}
```

Creating New Images from Scratch

In addition to loading images from files and from the Web, Cocoa Touch allows you to create new images on the fly. This blends `UIKit` functions with standard Quartz 2D graphics to build new `UIImage` instances.

So why would you build new images from scratch? The answers are many. You might create a thumbnail by shrinking a full-size picture into a new image. You could programmatically lay out a labeled game piece. You might generate a semitransparent backsplash for custom alert views. You can also add effects to existing images, such as the reflection discussed in Chapter 6, "Assembling Views and Animations," or you might just want to customize an image in some other way. Each of these examples builds a new image in code, whether that image is based on another or built entirely from new elements.

Cocoa Touch provides a simple way to build new images. As this code shows, you just create a new image context, draw to it, and then transform the context into a `UIImage` object:

```
UIImage *theImage;
UIGraphicsBeginImageContext(CGSizeMake(40.0f, 40.0f));
{
    CGContextRef context = UIGraphicsGetCurrentContext();
```

From the Library of Eric Berna

```
    // Draw to the context here

    UIImage *theImage = UIGraphicsGetImageFromCurrentImageContext();
}
UIGraphicsEndImageContext();
```

The drawing commands you use may consist of a combination of `UIKit` calls (such as `drawAtPoint:` and `drawInRect:`) and Core Graphics Quartz calls. The following method, which is taken from Chapter 5, "Working with View Controllers," draws a filled rounded rectangle. Its grey level (brightness) is passed as an argument.

```
- (UIImage*) buildSwatch: (int) aBrightness
{
    CGRect rect = CGRectMake(0.0f, 0.0f, 40.0f, 40.0f);
    UIGraphicsBeginImageContext(rect.size);

    UIBezierPath *path = [UIBezierPath
        bezierPathWithRoundedRect:rect cornerRadius:4.0f];
    [[[UIColor blackColor]
        colorWithAlphaComponent:(float) aBrightness / 10.0f] set];
    [path fill];

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

    return image;
}
```

Recipe: Displaying Images in a Scrollable View

Image display is all about memory. Treat large and small image display as separate problems. In Cocoa Touch, the `UIWebView` class easily handles memory-intense data. You might load a larger image into a web view using a method such as the following. This approach works well with bulky PDF images. `UIWebViews` offer a complete package of image presentation, including built-in scrolling and resizing.

```
- (void) loadImageIntoWebView: (NSString *) path
{
    // Automatically fit the image to the view
    self.webView.scalesPageToFit = YES;

    // Load the image by creating a request
    NSURL *fileURL = [NSURL fileURLWithPath:path];
    NSURLRequest *request = [NSURLRequest requestWithURL:fileURL];
    [self.webView loadRequest:request];
}
```

From the Library of Eric Berna

With smaller images, say less than half a megabyte in size when compressed, you can load them directly to `UIImageViews` and add them to your interface. Apple recommends that `UIImage` images never exceed 1024 by 1024 pixels because most device GPUs cannot handle single textures larger than that size.

The problem with basic image views is that they are static. Unlike web views, they do not respond to user scrolls and pinches. Embedding into a `UIScrollView` solves this problem. Scroll views provide those user interactions, allowing users to manipulate any image placed on the scroll view surface.

Recipe 7-11 demonstrates how to do this. It adds a weather map to an interface scroll view, as shown in Figure 7-10. Then it calculates a pair of minimum values based on the core size of the image—namely the least degree of zoom that allows the image to be fully seen in the scroll view. It assigns this value to the scroll view's `minimumZoomScale`. The maximum scale is set arbitrarily to four times the image size. These settings allow full user interaction with the image while limiting that interaction to a reasonable scope.

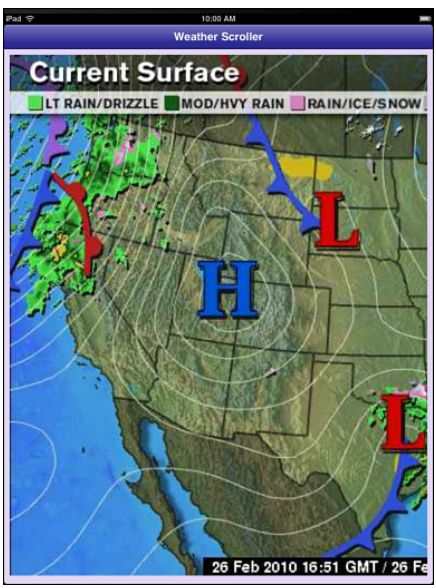


Figure 7-10 This live weather map is downloaded from a World Wide Web URL and layered onto a scroll view that allows users to scale and pan through the image.

The delegate method shown in the recipe identifies which view responds to zooming. For this recipe, that corresponds to the single image view placed onto the scroll view. Scroll views do not automatically know anything about any subviews you add to them. Defining this delegate method binds the zoom to your image.

From the Library of Eric Berna

Recipe 7-11 **Embedding an Image onto a Scroller**

```
@implementation TestBedViewController
- (UIView *)viewForZoomingInScrollView:(UIScrollView *)scrollView
{
    // Specify which view responds to zoom events
    return imageView;
}

- (void) loadView
{
    [super loadView];

    // Create the Scroll View
    scrollView = [[UIScrollView alloc] init];
    scrollView.delegate = self;
    scrollView.maximumZoomScale = 4.0f;
    [self.view addSubview:scrollView];

    // Create the embedded Image View
    imageView = [[UIImageView alloc] init];
    imageView.contentMode = UIViewContentModeCenter;
    [scrollView addSubview:imageView];

    // Use an operation queue to asynchronously load the data
    NSString *map = @"http://maps.weather.com/images/maps/\
        current/curwx_720x486.jpg";
    NSOperationQueue *queue = [[NSOperationQueue alloc] init];
    [queue addOperationWithBlock:
     ^{
         // Load the weather data
         NSURL *weatherURL = [NSURL URLWithString:map];
         NSData *imageData = [NSData dataWithContentsOfURL:weatherURL];

         // Update the image on the main thread using the main queue
         [[NSOperationQueue mainQueue] addOperationWithBlock:^{
             // Download the image data and set the image
             UIImage *weatherImage = [UIImage imageWithData:imageData];
             imageView.userInteractionEnabled = YES;
             imageView.image = weatherImage;
             imageView.frame = (CGRect){.size = weatherImage.size};

             // Adjust the scroll view zoom scale accordingly
             float scalex = scrollView.frame.size.width /
                 weatherImage.size.width;
             float scaley = scrollView.frame.size.height /
                 weatherImage.size.height;
```

From the Library of Eric Berna

```
            scrollView.zoomScale = MIN(scalex, scaley);
            scrollView.minimumZoomScale = MIN(scalex, scaley);
            scrollView.contentSize = weatherImage.size;
        }];
    }];
}
@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 7 and open the project for this recipe.

Creating a Multi-Image Paged Scroll

Scroll views aren't just about zooming. The `UIScrollView`'s paging property allows you to place images (or other views, for that matter) in a scroll view and move through them one view-width at a time. The key lies in ensuring that each image loaded exactly matches the width of the scroll view frame for horizontal presentations or its height for vertical ones. Set the `pagingEnabled` property to `YES`. This allows users to flick their way from one image to another. Make sure to update the page setup whenever the orientation changes to allow the scroll view's pages to match the reoriented content.

```
// Adjust the page setup when the orientation changes
- (void) setUpPages
{
    // Calculate the width for the current orientation
    float baseWidth = self.view.frame.size.width;

    // Match the scroller content size to that
    scrollView.contentSize =
        CGSizeMake(NPAGES * baseWidth, scrollView.frame.size.height);

    // Use page tags to update each image frame
    for (int i = 0; i < NPAGES; i++)
    {
        UIView *pageView = [scrollView viewWithTag:(900 + i)];
        pageView.frame = CGRectMake(i * baseWidth, 0.0f,
            baseWidth, BASE_HEIGHT);
    }

    // Refresh the content offset to the current page
    scrollView.contentOffset =
        CGPointMake(pageNumber * baseWidth, 0.0f);
}
```

From the Library of Eric Berna

Summary

This chapter introduced many ways to handle images, including picking, reading, modifying, and saving. You saw recipes that showed you how to use the iOS's built-in editor selection process and how to snap images with the camera. You learned about AVFoundation video and Core Image filters. You also read about adding images to the `UIScrollView` class and how to send pictures as e-mail attachments. Before moving on from this chapter, here are some thoughts about the recipes you saw here:

- The built-in image picker is a memory hog. Develop your code around that basic fact of life and consider whether your user is better served using AVFoundation.
- On the iPad, always present your image picker using a popover. The iPad offers a much larger screen and its use of popovers helps limit the visual transitions you present to your user.
- Keep in mind, as well, that your iPad interface needs to work in all orientations. Refresh your image presentations whenever the orientation changes, using the kinds of algorithms discussed in this chapter to update their size using the most recent screen geometry.
- Use PDF document creation meaningfully to add value to your application. Placing your generated PDF documents into the Documents folder or offering them as e-mail attachments allows users to move those PDFs to destinations where they can be shared with and viewed by others.
- Always provide user feedback when working with long processing delays. Most image manipulation is slow. The simulator always outperforms iOS devices, so test your applications on the device as well as the simulator and provide a mechanism such as the HUD display used in this chapter that lets users know that ongoing operations may take some time. There is also a large discrepancy in the processing performance of early iOS units versus the later-generation iPods, iPhones, and iPads. Due to camera quality, there is also a big difference in image quality and size between early units and later generations.
- The OpenCV computer vision library has been widely ported to iOS. Consider making use of its features directly rather than reinventing the wheel. For high performance, you might also consider vImage.
- Sending image e-mail attachments from in-program is a handy OS feature. Make sure that you check whether e-mail is available on your device before attempting to use the controller and be aware that sending images can be very, very slow.
- Thumbnails use far less memory than loading all images at once. Consider precomputing icon versions of your pictures in addition to using the thumbnail-sizing routines shown in this chapter.

From the Library of Eric Berna

8

Gestures and Touches

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

Touches

Cocoa Touch implements direct manipulation in the simplest way possible. It sends touch events to the view you're working with. As an iOS developer, you tell the view how to respond to each touch. You can even add gesture recognizers to your view, allowing iOS to perform basic recognition tasks to collect swipes, drags, and taps for you.

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

At the most basic level, touches and their information are stored in `UITouch` objects. Each object represents a single touch event. Your applications can receive these either in its view class or in its view controller class; both implement touch handlers via inheritance from the `UIResponder` class. You may choose between these two class types to decide where you process and respond to touches. Trying to implement low-level gesture control in the wrong class has tripped up many new iOS developers.

Handling touches in views may seem counterintuitive. You probably expect to separate the way an interface looks (its view) from the way it responds to touches (its controller).

From the Library of Eric Berna

Using views for direct touch interaction may seem to contradict Model-View-Controller design orthogonality, but it can be necessary and help promote encapsulation.

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

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

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

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

Phases

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

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

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

From the Library of Eric Berna

Touches and Responder Methods

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

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

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

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

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

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

Touching Views

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

Just because a touch is passed to a view doesn't mean that a view has to respond. Each view can use a "hit test" to choose whether to handle a touch or to let that touch fall

From the Library of Eric Berna

through to views beneath it. As you see in the recipes that follow, you can use clever response strategies to decide when your view should respond, particularly when you're using irregular art with partial transparency.

Multitouch

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

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

Gesture Recognizers

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

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

- **Taps**—Taps correspond to single or multiple finger taps onscreen. Users can tap with one or more fingers; you specify how many fingers you require as a gesture recognizer property and how many taps you want to detect. You can create a tap recognizer that works with single finger taps, or more complex recognizers that look for, for example, two-fingered triple-taps.
- **Swipes**—Swipes are short, single- or multi-touch gestures that move in a single cardinal direction: up, down, left, or right. They cannot move too far off course from that primary direction. You set the direction or directions you want your recognizer to work with. The recognizer returns the detected direction as a property.
- **Pinches**—To pinch or unpinch, a user must move two fingers together or apart in a single movement. The recognizer returns a scale factor indicating the degree of pinching.
- **Rotations**—To rotate, a user moves two fingers at once either in a clockwise or counterclockwise direction, producing an angular rotation as the main returned property.

From the Library of Eric Berna

- **Pan**—Pans occur when users drag their fingers across the screen. The recognizer determines the change in translation produced by that drag.
- **Long press**—To create a long press, the user touches the screen and holds his or her finger (or fingers) there for a specified period of time. You can specify how many fingers must be used before the recognizer triggers.

Recipe: Adding a Simple Direct Manipulation Interface

Your design focus moves from the `UIViewController` to the `UIView` when you work with direct manipulation. The view, or more precisely the `UIResponder`, forms the heart of direct manipulation development. Create touch-based interfaces by customizing methods that derive from the `UIResponder` class.

Recipe 8-1 centers on touches in action. This example creates a child of `UIImageView` called `DragView` and adds touch responsiveness to the class. Being an image view, it's important to enable user interaction (that is, set `setUserInteractionEnabled` to `YES`). This property affects all the view's children as well as the view itself.

The recipe works by updating a view's center to match the movement of an onscreen touch. When a user first touches any `DragView`, the object stores the start location as an offset from the view's origin. As the user drags, the view moves along with the finger—always maintaining the same origin offset so that the movement feels natural. Movement occurs by updating the object's center. Recipe 8-1 calculates x and y offsets and adjusts the view center by those offsets after each touch movement.

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

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

Recipe 8-1 **Creating a Draggable View**

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

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

From the Library of Eric Berna

```
    return self;
}

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

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

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

> **Get This Recipe's Code**
>
> To 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 8 and open the project for this recipe.

Recipe: Adding Pan Gesture Recognizers

Gesture recognizers allow you to achieve the same kind of interaction shown in Recipe 8-1 without working quite so directly with touch handlers. Pan gesture recognizers detect dragging gestures. They allow you to assign a callback that triggers whenever iOS senses panning.

Recipe 8-2 mimics Recipe 8-1's behavior by adding a recognizer to the view when it is first initialized. As iOS detects the user dragging on a `DragView` instance, the `handlePan:` callback updates the view's center to match the distance dragged.

Recipe 8-2 **Using a Pan Gesture Recognizer to Drag Views**

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

From the Library of Eric Berna

```
@implementation DragView
- (id) initWithImage: (UIImage *) anImage
{
    if (self = [super initWithImage:anImage])
    {
        self.userInteractionEnabled = YES;
        UIPanGestureRecognizer *pan = [[UIPanGestureRecognizer alloc]
            initWithTarget:self action:@selector(handlePan:)];
        self.gestureRecognizers = [NSArray arrayWithObject: pan];
    }
    return self;
}

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

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

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

This code uses what might seem like an odd way of calculating distance. It stores the original view location in an instance variable (`previousLocation`) and then calculates the offset from that point each time the view updates with a pan detection callback. It does this because the iOS SDK assumes you'll be updating views using affine transforms, not by moving view centers, as done here. This workaround creates a dx/dy offset pair and applies that offset to the view's center.

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

From the Library of Eric Berna

For this recipe to work iteratively—that is, to apply new transforms to existing changes—you will have to store previous state, just as this code does. Unlike this recipe, Recipe 8-3, which follows next, demonstrates how to layer several changes on top of each other and use multiple gesture recognizers at once.

> **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 8 and open the project for this recipe.

Recipe: Using Multiple Gesture Recognizers at Once

Recipe 8-3 builds off the ideas presented in Recipe 8-2, but with several differences. First, it introduces multiple recognizers that work in parallel. To achieve this, the code creates an array of three separate recognizers—rotation, pinch, and pan—and adds them all to the `DragView`'s `gestureRecognizers` property. Further, it assigns the `DragView` as the delegate for each recognizer. This allows the `DragView` to implement the `gestureRecognizer:shouldRecognizeSimultaneouslyWithGestureRecognizer:` delegate method, enabling these recognizers to work simultaneously. Until this method is added to return `YES` as its value, only one recognizer will take charge at a time. Using parallel recognizers allows you to, for example, both zoom and rotate in response to a user's pinch gesture.

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

Second, Recipe 8-3 extends the view's state to include scale and rotation instance variables. These items keep track of previous transformation values and permit the code to build compound affine transforms. These compound transforms, which are established in Recipe 8-3's `updateTransformWithOffset:` method, combine translation, rotation, and scaling into a single result. Unlike the previous recipe, this recipe uses transforms uniformly to apply changes to its objects, which is the standard practice for recognizers.

Finally, this recipe introduces a hybrid approach to gesture recognition. Instead of adding a `UITapGestureRecognizer` to the view's recognizer array, Recipe 8-3 demonstrates how you can add the kind of basic touch method used in Recipe 8-1 to catch a triple-tap. In this example, a triple-tap resets the view back to the identity transform. This undoes any manipulation previously applied to the view and reverts it to its original position, orientation, and size. As you can see, the touches began, moved, ended, and cancelled methods work seamlessly alongside the gesture recognizer callbacks, which is the point of including this extra detail in this recipe. Adding a tap recognizer would have worked just as well.

From the Library of Eric Berna

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

Recipe 8-3 **Recognizing Gestures in Parallel**

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

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

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

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

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

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

From the Library of Eric Berna

```
    self.transform = CGAffineTransformRotate(self.transform, theta);
    self.transform = CGAffineTransformScale(self.transform, scale, scale);
}

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

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

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

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

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

    self.userInteractionEnabled = YES;

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

    // Add gesture recognizer suite
    UIRotationGestureRecognizer *rot =
        [[UIRotationGestureRecognizer alloc]
            initWithTarget:self action:@selector(handleRotation:)];
    UIPinchGestureRecognizer *pinch = [[UIPinchGestureRecognizer alloc]
        initWithTarget:self action:@selector(handlePinch:)];
    UIPanGestureRecognizer *pan = [[UIPanGestureRecognizer alloc]
```

From the Library of Eric Berna

```
        initWithTarget:self action:@selector(handlePan:)];
    self.gestureRecognizers =
        [NSArray arrayWithObjects: rot, pinch, pan, nil];
    for (UIGestureRecognizer *recognizer in self.gestureRecognizers)
        recognizer.delegate = self;

    return self;
}
@end
```

Resolving Gesture Conflicts

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

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

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

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

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

Get This Recipe's Code

To 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 8 and open the project for this recipe.

From the Library of Eric Berna

Recipe: Constraining Movement

One problem with the simple approach of the earlier recipes in this chapter is that it's entirely possible to drag a view offscreen to the point where the user cannot see or easily recover it. Those recipes use unconstrained movement. There is no check to test whether the object remains in view and is touchable. Recipe 8-4 fixes this problem by constraining a view's movement to within its parent.

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

Figure 8-1 shows a sample interface. The subviews (flowers) are constrained into the black rectangle in the center of the interface and cannot be dragged off-view. Recipe 8-4's code is general and can adapt to parent bounds and child views of any size.

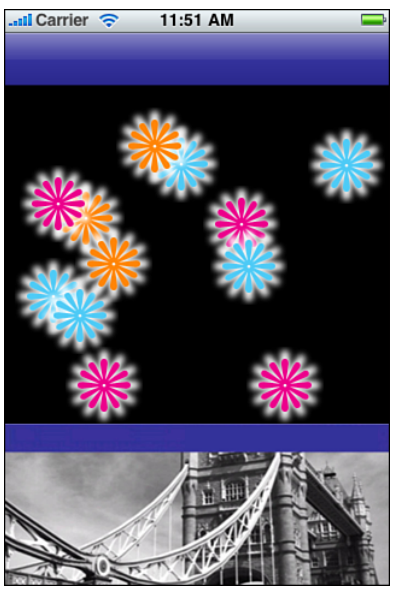


Figure 8-1 The movement of these flowers is bounded into the black rectangle.

Recipe 8-4 **Bounded Movement**

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

From the Library of Eric Berna

```
        previousLocation.y + translation.y);

    // Bound movement into parent bounds
    float halfx = CGRectGetMidX(self.bounds);
    newcenter.x = MAX(halfx, newcenter.x);
    newcenter.x = MIN(self.superview.bounds.size.width - halfx,
        newcenter.x);

    float halfy = CGRectGetMidY(self.bounds);
    newcenter.y = MAX(halfy, newcenter.y);
    newcenter.y = MIN(self.superview.bounds.size.height - halfy,
        newcenter.y);

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

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 8 and open the project for this recipe.

Recipe: Testing Touches

Most onscreen view elements for direct manipulation interfaces are not rectangular. This complicates touch detection because parts of the actual view rectangle may not correspond to actual touch points. Figure 8-2 shows the problem in action. The screenshot on the right shows the interface with its touch-based subviews. The shot on the left shows the actual view bounds for each subview. The light gray areas around each onscreen circle fall within the bounds, but touches to those areas should not "hit" the view in question.

iOS senses user taps throughout the entire view frame. This includes the undrawn area, such as the corners of the frame outside the actual circles of Figure 8-2, just as much as the primary presentation. That means that unless you add some sort of hit test, users may attempt to tap through to a view that's "obscured" by the clear portion of the `UIView` frame.

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

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

This adds the backsplashes shown in Figure 8-2 (left) without affecting the actual onscreen art. In this case, the art consists of a centered circle with a transparent background. Unless you add some sort of test, all taps to any portion of this frame are captured by the view in question. Enabling background colors offers a convenient debugging aid to visualize the true extent of each view; don't forget to comment out the background color assignment in production code.

From the Library of Eric Berna

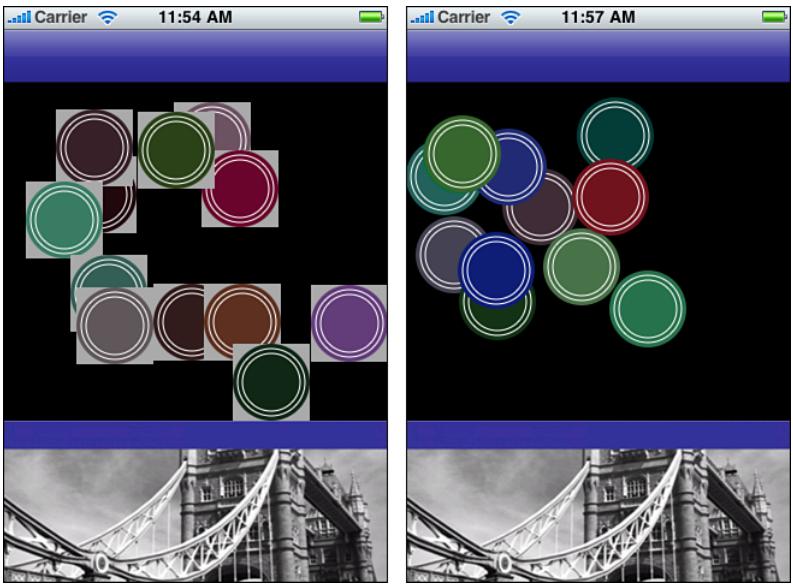


Figure 8-2 The application should ignore touches to the gray areas that surround each circle (left). The actual interface (right) uses zero alpha values to hide the parts of the view that are not used.

Recipe 8-5 adds a simple hit test to the views, determining whether touches fall within the circle. This test overrides the standard `UIView`'s `pointInside:withEvent:` method. This method returns either `YES` (the point falls inside the view) or `NO` (it does not). The test here uses basic geometry, checking whether the touch lies within the circle's radius. You can provide any test that works with your onscreen views. As you see in Recipe 8-6, that test can be expanded for much finer control.

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

Recipe 8-5 **Providing a Circular Hit Test**

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

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

From the Library of Eric Berna

```
    pt.y = (point.y - HALFSIDE) / HALFSIDE;

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

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

Get This Recipe's Code

To 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 8 and open the project for this recipe.

Recipe: Testing Against a Bitmap

Unfortunately, most views don't fall into the simple geometries that make the hit test from Recipe 8-5 so straightforward. The flowers shown in Figure 8-1, for example, offer irregular boundaries and varied transparencies. For complicated art, it helps to test touches against a bitmap. Bitmaps provide byte-by-byte information about the contents of an image-based view, allowing you to test whether a touch hits a solid portion of the image or should pass through to any views below.

Recipe 8-6 extracts an image bitmap from a `UIImageView`. It assumes that the image used provides a pixel-by-pixel representation of the view in question. When you distort that view (normally by resizing a frame or applying a transform), update the math accordingly. `CGPoints` can be transformed via `CGPointApplyAffineTransform()` to handle scaling and rotation changes. Keeping the art at a 1:1 proportion to the actual view pixels simplifies lookup and avoids any messy math. You can recover the pixel in question, test its alpha level, and determine whether the touch has hit a solid portion of the view.

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

Note

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

From the Library of Eric Berna

Recipe 8-6 **Testing Touches Against Bitmap Alpha Levels**

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

// Return the bitmap from a provided image
unsigned char *getBitmapFromImage (UIImage *image)
{

    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    if (colorSpace == NULL)
    {
        fprintf(stderr, "Error allocating color space\n");
        return NULL;
    }

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

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

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

    return data;
}

- (id) initWithImage: (UIImage *) anImage
```

From the Library of Eric Berna

```
{
    if (self = [super initWithImage:anImage])
    {
        self.userInteractionEnabled = YES;
        bytes = getBitmapFromImage(anImage);
    }
    return self;
}

- (void) dealloc
{
    free(bytes);
    [super dealloc];
}

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

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 8 and open the project for this recipe.

Recipe: Adding Persistence to Direct Manipulation Interfaces

Persistence represents a key iOS design touch point. After users leave a program, Apple strongly recommends that, upon returning, they encounter a state that matches as closely to where they left off as possible. Even with new multitasking features, users should expect to resume their interactions where they left off. Adding persistence to a direct manipulation interface, in the simplest approach, involves storing a representation of the onscreen data when an application suspends or terminates and then restoring that state on startup.

Storing State

Every view knows its position because you can query its frame or center. This enables you to easily recover and store positions for each onscreen flower. The flower type (green, pink, or blue) is another matter. For each view to report its current flower, the `DragView` class must store that value, too. Adding a string instance variable enables the view to return the image name used. Extending the `DragView` interface lets you do that.

From the Library of Eric Berna

```
@interface DragView : UIImageView
{
    CGPoint startLocation;
}
@property (nonatomic, strong) NSString *whichFlower;
@end
```

Adding this extra property lets the view controller that owns the flowers store both a list of colors and a list of locations to its defaults file. Here, a simple loop collects both values from each draggable view and then stores them:

```
- (void) updateDefaults
{
    NSMutableArray *colors =  [[NSMutableArray alloc] init];
    NSMutableArray *locs = [[NSMutableArray alloc] init];

    for (DragView *dv in [self.view subviews])
    {
        [colors addObject:dv.whichFlower];

        // Alternatively use [NSValue valueWithCGRect]
        [locs addObject:NSStringFromCGRect(dv.frame)];
    }

    [[NSUserDefaults standardUserDefaults] setObject:colors
        forKey:@"colors"];
    [[NSUserDefaults standardUserDefaults] setObject:locs
        forKey:@"locs"];
    [[NSUserDefaults standardUserDefaults] synchronize];
}
```

Defaults, as you can see, work like a dictionary. Just assign an object to a key and iOS updates the preferences file associated with your application ID. Defaults are stored in Library/Preferences inside your application's sandbox. Calling the `synchronize` method updates those defaults immediately instead of waiting for the program to terminate. This ensures that any changes you make are instantly transmitted to your preferences file; doing so comes at a slight cost from updating the file.

The `NSStringFromCGRect()` function provides a tight way to store frame information as a string. To recover the rectangle, issue `CGRectFromString()`. Each call takes one argument: a `CGRect` in the first case, an `NSString` object in the second. The `UIKit` framework provides functions that translate points and sizes as well as rectangles to and from strings.

This `updateDefaults` method, which saves the current state to disk, should be called in the application delegate's `applicationWillResignActive:` methods, just before the program resigns control. The defaults stored reflect the most recent application state.

From the Library of Eric Berna

```
- (void)applicationWillResignActive:(UIApplication *)application
{
    [self.tbvc updateDefaults];
}
```

Alternatively, given the small quantity of data involved, you could update defaults after each user touch has been processed and the gesture finished.

Recovering State

To bring views back to life, re-create them in a standard initialization method such as the `loadView`, `viewDidAppear:`, or `viewDidLoad` method of your view controller. (Persistence awareness can also reside in the view controller's `init` method if you're not working with views.) Your methods should find any previous state information and update the interface to match that state.

When querying user defaults, Recipe 8-7 checks whether state data is unavailable (for example, the value returned is `nil`). When state data goes missing, the method creates new views at random points.

> **Note**
>
> When working with large data sources, you may want to initialize and populate your saved object array in the `UIViewController`'s `init` method and then draw the saved objects in `loadView` or `viewDidLoad`. Where possible, use threading when working with many objects to avoid too much processing on the main thread. This can make the program laggy or unresponsive by blocking GUI updates. Never perform any actual GUI tasks outside the primary thread. If it's a `UIKit` call, keep it on the main thread. If you stick strictly to Quartz, however, most calls are thread-safe.

Recipe 8-7 **Checking for Previous State**

```
- (void) loadFlowersInView: (UIView *) backdrop
{
    // Attempt to read in previous colors and locations
    NSMutableArray *colors = [[NSUserDefaults standardUserDefaults]
        objectForKey:@"colors"];
    NSMutableArray *locs = [[NSUserDefaults standardUserDefaults]
        objectForKey:@"locs"];

    // Add the flowers to random points on the screen
    for (int i = 0; i < MAXFLOWERS; i++)
    {
        NSString *whichFlower = [[NSArray
            arrayWithObjects:@"blueFlower.png", @"pinkFlower.png",
            @"orangeFlower.png", nil] objectAtIndex:(random() % 3)];
        if (colors && ([colors count] == MAXFLOWERS))
            whichFlower = [colors objectAtIndex:i];
```

From the Library of Eric Berna

```
        DragView *dragger = [[DragView alloc] initWithImage:
            [UIImage imageNamed:whichFlower]];
        dragger.center = randomPoint();
        dragger.userInteractionEnabled = YES;
        dragger.whichFlower = whichFlower;
        if (locs && ([locs count] == MAXFLOWERS))
            dragger.frame = CGRectFromString([locs objectAtIndex:i]);

        [backdrop addSubview:dragger];
    }
}
```

> **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 8 and open the project for this recipe.

Recipe: Persistence Through Archiving

Recipe 8-7 created persistence via the user defaults system. It stored descriptions of the onscreen views and built those views from the recovered description. Recipe 8-8 takes things to the next level. Instead of storing descriptions, it archives the objects themselves, or at least as much of the objects as is necessary to reconstruct them at launch time.

Two classes—`NSKeyedArchiver` and `NSKeyedUnarchiver`—provide an elegant solution for archiving objects into a file for later retrieval. These archive classes provide an object persistence API that allows you to restore objects between successive application sessions. The example you're about to see uses the simplest archiving approach. It stores a single root object, which in this case is an array of `DragViews` (that is, the flowers).

To create an archivable object class, you must define a pair of methods. The first, `encodeWithCoder:`, stores any information needed to rebuild the object. In this case, that is the view's frame and its flower. Both are stored as `NSString` objects. The second method, `initWithCoder:`, recovers that information and initializes objects using saved information. Here are the two methods defined for the `DragView` class, allowing objects of this class to be encoded and retrieved from an archive:

```
- (void) encodeWithCoder: (NSCoder *)coder
{
    [coder encodeCGRect:self.frame forKey:@"viewFrame"];
    [coder encodeObject:self.whichFlower forKey:@"flowerType"];
}

- (id) initWithCoder: (NSCoder *)coder
{
    if (self = [super initWithImage:nil])
    {
        self.frame = [coder decodeCGRectForKey:@"viewFrame"];
```

From the Library of Eric Berna

```
        self.whichFlower = [coder decodeObjectForKey:@"flowerType"];
        self.image = [UIImage imageNamed:self.whichFlower];
        self.userInteractionEnabled = YES;
    }
    return self;
}
```

Each element is stored with a key name. Keys let you recover stored data in any order. Special `UIKit` extensions to the `NSCoder` class add storage methods for points, sizes, rectangles, affine transforms, and edge insets. This example takes advantage of the rectangle method for encoding and decoding the view frame.

Data is saved to an actual file. You supply an archive path to that file. This example stores its data in the Library folder in the sandbox in a file called flowers.archive. Be aware that any application state files belong in the Library. Reserve the application's Documents folder for end-user documents only. The user's Documents folder should always be ready for possible iTunes management. Although that access remains a developer option (by setting the `UIFileSharingEnabled` key in the application's Info.plist property list file), best practices demand that you use the Library folder for such material and not the user's Documents folder.

```
#define DATAPATH [NSString stringWithFormat: \
    @"%@/Library/flowers.archive", NSHomeDirectory()]
```

or

```
#define DATAPATH [[NSSearchPathForDirectoriesInDomains( \
    NSLibraryDirectory, NSUserDomainMask, YES) objectAtIndex:0] \
    stringByAppendingPathComponent:@"flowers.archive"]
```

So for this direct manipulation interface, how do you actually perform the archiving and unarchiving? Recipe 8-8 shows the exact calls, which in this case are implemented in the view controller. Here are two custom methods that collect the `DragViews` and archive them to the file and then retrieve the views from the file.

Notice that the latter method returns a Boolean value. This indicates whether the views could be read in correctly. On failure, a fallback method generates a new set of subviews. It's assumed either that the data was corrupted or that this is the first time the application was run. Either way, the application generates new data to populate the backdrop.

Recipe 8-8 **Archiving Interfaces**

```
- (void) archiveInterface
{
    NSArray *flowers = [self.view subviews];
    [NSKeyedArchiver archiveRootObject:flowers toFile:DATAPATH];
}

- (BOOL) unarchiveInterface
{
```

From the Library of Eric Berna

```
    NSArray *flowers = [NSKeyedUnarchiver
        unarchiveObjectWithFile:DATAPATH];
    if (!flowers) return NO;

    for (UIView *aView in flowers)
        [self.view addSubview:aView];
    return YES;
}
```

Get This Recipe's Code

To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from the book, go to the folder for Chapter 8 and open the project for this recipe.

Recipe: Adding Undo Support

Undo support provides another important component of direct manipulation interfaces. For a simple GUI, this involves little more than returning each object to a previous onscreen position. Cocoa Touch offers the `NSUndoManager` class to provide a way to reverse user actions.

Creating an Undo Manager

By default, every application window provides a shared undo manager. You can use this shared manager or create your own. The easiest way to hook into the shared manager is to wait until the view has been presented in `viewDidAppear:` and then point a local instance variable to that shared manager:

```
- (void) viewDidAppear:(BOOL)animated
{
    undoManager = self.view.window.undoManager;
    [undoManager setLevelsOfUndo:999];
}
```

The manager can store an arbitrary number of undo actions. You specify how deep that stack goes. The bigger the stack, the more memory you will use. Many applications allow three, five, or ten levels of undo when memory is tight. Extremely lightweight actions allow the kind of "unlimited" undo shown in this recipe. Each action can be complex, involving groups of undo activities, or the action can be simple as in the example shown here. These undos do one thing: move a view to a previous location.

Child-View Undo Support

All children of the `UIResponder` class can find the nearest undo manager in the responder chain. This means that if you use the window's undo manager in your view controller, each `DragView` automatically knows about that manager through its `undoManager`

From the Library of Eric Berna

property. This is enormously convenient because you can add undo support in your main view controller, and all your child views basically pick up that support for free.

Working with Navigation Bars

When working with undo managers and the navigation bar, child views should store a pointer to their view controller; Apple has not opened the API that lets views retrieve the view controller that owns them, neither directly nor as a subview. Knowing their view controller lets children coordinate with any bar button items in the enclosing navigation presentation. You only want an Undo button to appear when items are available on the undo stack.

```
@interface DragView : UIImageView
{
    CGPoint previousLocation;
}
@property (nonatomic, strong) NSString *whichFlower;
@property (nonatomic, strong) UIViewController *viewController;
@end
```

Upon adding an undo item to the manager, you may want to display an Undo button as this example does. The Undo button calls the manager's undo method, which in turn uses the target, action, and objects set stored at the top of the undo stack to perform the actual reversion. When the undo manager has no more undos to perform, the Undo button should hide.

```
- (void) checkUndoAndUpdateNavBar
{
    // Do not interrupt any ongoing operations
    if ([self.undoManager isUndoing])
    {
        [self performSelector:@selector(checkUndoAndUpdateNavBar)
            withObject:nil afterDelay:0.1f];
        return;
    }

    // Don't show the undo button if the undo stack is empty
    if (!self.undoManager.canUndo)
        self.navigationItem.leftBarButtonItem = nil;
    else
        self.navigationItem.leftBarButtonItem =
            BARBUTTON(@"Undo", @selector(undo));
}

- (void) undo
{
    [self.undoManager undo];
}
```

From the Library of Eric Berna

Notice that the checking method waits for the undo manager to finish any ongoing actions before proceeding to update the navigation bar. Unfortunately, Apple has not yet implemented an undo API with a completion handler.

Registering Undos

Here is the simplest call to register an undo. It stores the object location at the start of a touch sequence, specifying that upon undo, the object should reset its position to this start location. This call is made from the child view, and not from the view controller. This approach tells the undo manager how to reset an object to its previous attributes.

```
[self.undoManager registerUndoWithTarget:self
    selector:@selector(resetPosition)
    object:NSStringFromCGPoint(self.center)];
```

The preferred alternative approach uses an invocation instead of a target and selector. The invocation records a message for reverting state—that is, it stores a way that it can jump back to the previous state.

```
[[self.undoManager prepareWithInvocationTarget:self]
    setPosition:previousLocation fromPosition:newLocation];
```

With invocations, you can use a method with any number of arguments and argument types. This invocation simplifies adding redo support, which is why it is preferred. Regardless of the approach used, prepare your undos before applying changes to your object's state.

There are several ways to approach the undo registration process in a direct manipulation interface. Placing a call to a setter/unsetter method when the gesture finishes provides the easiest solution, as shown in Recipe 8-9. It guards against adding new items to the undo stack by waiting for the gesture recognizer to enter its `UIGestureRecognizerStateEnded` state.

When using direct manipulation without touch recognizers, be aware that if users touch an object and release without moving it, undo results may be imperceptible. You may want to add a check into the touches-ended routine to make sure that an object was actually moved. If not, remove the last item from the undo stack by issuing `undo`. This is not an issue when you're using gesture recognizers, as shown in Recipe 8-9.

Recipe 8-9 lists the actual undo code. The `setPosition:fromPosition:` method provides both a set and reset solution for the undo manager. Upon registration, it stores the new and old positions of a view into the undo stack. Upon undo, it animates the view back to the original position, providing a visual connection between the new value and the old. Although redo support is not used in this recipe, the `setPosition:fromPosition:` method is redo compliant. When called by the undo manager, the repeat `prepareWithInvocationTarget:` call gets added to the redo stack.

The delayed selector in this method, `checkUndoAndUpdateNavBar:`, triggers after the animation has completed. This allows the `setPosition:fromPosition:` method to finish before any checks are made against the undo stack. The stack will not decrease its count until after the registered method returns.

From the Library of Eric Berna

Recipe 8-9 **Creating a Custom Undo Routine**

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

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

- (void) setPosition: (CGPoint) position fromPosition: previousPosition
{
    // Prepare undo-redo first
    [[self.undoManager prepareWithInvocationTarget:self]
        setPosition:previousPosition fromPosition:position];

    // With no completion handlers, add a slight delay to the GUI update
    [self.viewController
        performSelector:@selector(checkUndoAndUpdateNavBar)
        withObject:nil afterDelay:0.2f];

    // Make the change
    [UIView animateWithDuration:0.1f animations:
        ^{self.center = position;}];
}

// Catch the completion of the event to prepare the undo manager
- (void) handlePan: (UIPanGestureRecognizer *) uigr
{
    CGPoint translation = [uigr translationInView:self.superview];
    CGPoint newcenter = CGPointMake(previousLocation.x + translation.x,
        previousLocation.y + translation.y);

    // Bound movement into parent bounds
    float halfx = CGRectGetMidX(self.bounds);
    newcenter.x = MAX(halfx, newcenter.x);
    newcenter.x = MIN(self.superview.bounds.size.width - halfx,
        newcenter.x);

    float halfy = CGRectGetMidY(self.bounds);
    newcenter.y = MAX(halfy, newcenter.y);
    newcenter.y = MIN(self.superview.bounds.size.height - halfy,
        newcenter.y);

    // Set new location
    self.center = newcenter;
```

From the Library of Eric Berna

```
    // Test for end state, update undo stack and nav bar
    if (uigr.state == UIGestureRecognizerStateEnded)
    {
        [self setPosition:self.center fromPosition:previousLocation];
        [self.viewController
            performSelector:@selector(checkUndoAndUpdateNavBar)];
    }
}
```

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 8 and open the project for this recipe.

Adding Shake-Controlled Undo Support

First introduced in the 3.0 SDK, shake-to-undo support offers a "whimsical" feature that automatically produces an undo/redo menu. It's supposed to play off the real-world Etch-a-Sketch toy, which users shake to erase. When users shake their iOS device, this menu appears, connected to the current undo manager. The menu allows users to undo the previous action or redo an action that has been undone. Figure 8-3 shows the shake-to-undo menu.

Shake-to-edit is a "high concept" feature that's not entirely practical in application. Its discoverability is dismal and its unintentional use problematic ("Why is my device asking 'Undo edit?'"). Training your users to shake the phone rather than press an Undo button presents a real-world hurdle as well. Even if a user is trained, it's a pain to keep shaking the device to process a series of undo events. If you plan to include this feature in your applications (which I really don't recommend), consider using it to enhance and extend an existing undo setup rather than replace it.

Adding support for shake-to-edit takes just a few steps. Here is an item-by-item list of the changes you need to make to offer this feature in your application.

Add an Action Name for Undo and Redo (Optional)

Action names provide the word or words that appear after "Undo" and "Redo," as shown in Figure 8-3. Here, the action name is set to "movement." The undo menu option is therefore Undo Movement. Extend the `setPosition:` method to provide this name by adding this line right after you prepare the invocation target:

```
if (![self.undoManager isUndoing])
    [self.undoManager setActionName:@"movement"];
```

From the Library of Eric Berna

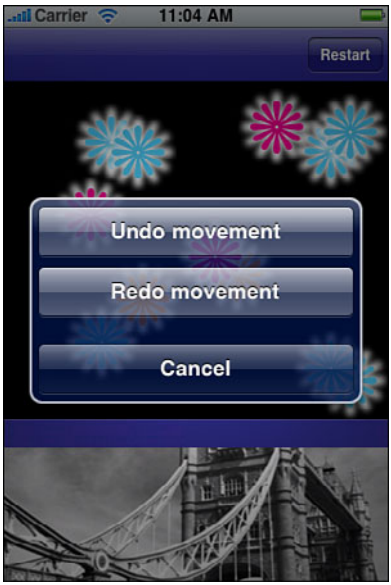


Figure 8-3 Shake-to-undo provides an undo/redo menu for users.

Provide Shake-To-Edit Support

Locate the `applicationDidFinishLaunching:` method of your application delegate. In that method add this line. Setting the `applicationSupportsShakeToEdit` property explicitly adds shake-to-edit support to the application as a whole.

```
application.applicationSupportsShakeToEdit = YES;
```

Force First Responder

For a view controller to handle undo/redo, it must always be first responder. Because each application may be handling several undo manager clients, the application must match each undo manager to a particular view controller. Only the first responder receives undo/redo calls.

Because the undo manager typically lives inside a `UIViewController` instance, make sure to add the following methods to your view controller. These ensure that it becomes first responder whenever it appears onscreen and that its undo manager is used.

```
- (BOOL)canBecomeFirstResponder {
    return YES;
}
```

From the Library of Eric Berna

```
- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
    [self becomeFirstResponder];
}

- (void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];
    [self resignFirstResponder];
}
```

Recipe: Drawing Touches Onscreen

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

Recipe 8-10 combines gestures with `drawRect` to create touch-based painting. As a user touches the screen, the `TouchTrackerView` class builds a Bezier path that follows those points. At each touch movement, the `touchesMoved:withEvent:` method calls `setNeedsDisplay`. This, in turn, triggers a call to `drawRect:`, where the view draws the accumulated Bezier path to reflect the traced gesture. Figure 8-4 shows the interface with a user-created path.

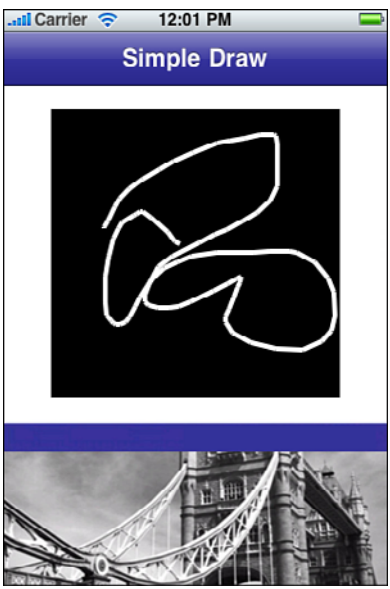


Figure 8-4 A simple painting tool for iOS requires little more than collecting touches along a path and painting that path with `UIKit`/Quartz 2D calls.

From the Library of Eric Berna

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

Recipe 8-10 **Touch-Based Painting in a `UIView`**

```
@interface TouchTrackerView : UIView
{
    UIBezierPath *path;
}
@property (retain) UIBezierPath *path;
@end

@implementation TouchTrackerView
@synthesize path;

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

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

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

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

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

From the Library of Eric Berna

```
- (void) drawRect:(CGRect)rect
{
    // Draw the path
    [path stroke];
}
@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 8 and open the project for this recipe.

Recipe: Smoothing Drawings

Depending on the device in use and the amount of simultaneous processing involved, capturing user gestures may produce results that are rougher than desired. Figure 8-5 demonstrates the kind of angularity that derives from granular input. Touch events are often limited by CPU demands. Using a real-time smoothing algorithm can offset those limitations by interpolating between points using basic splining.

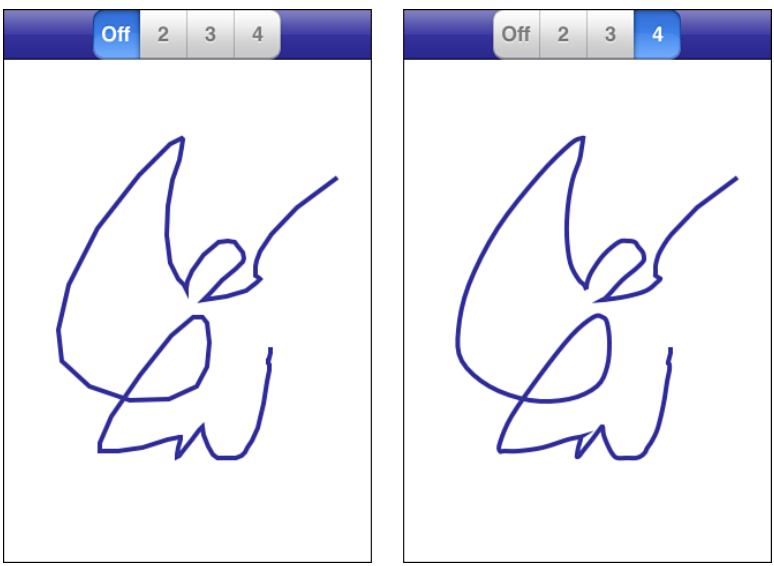


Figure 8-5 Catmull-Rom smoothing can be applied in real time to improve arcs between touch events. The images shown here are based on an identical gesture input, with and without smoothing applied.

Catmull-Rom splines offer one of the simplest approaches to create continuous curves between key points. This algorithm ensures that each initial point you provide remains

From the Library of Eric Berna

part of the final curve, so the resulting path retains the original path's shape. You choose the number of points to interpolate between each of your reference points. The trade-off lies between processing power and greater smoothing. The more points you add, the more CPU resources you'll consume. As you can see when using the sample code that accompanies this chapter, a little smoothing goes a long way, even on the iPad 2. The iPad 2 is so responsive that it's hard to draw a particularly jaggy line in the first place.

Recipe 8-11 demonstrates how to extract points from an existing Bezier path and then apply splining to create a smoothed result. Catmull-Rom uses four points at a time to calculate intermediate values between the second and third points in that sequence using a granularity you specify between those points.

Recipe 8-11 provides an example of just one kind of real-time geometric processing you might add to your applications. There are many other algorithms out there in the world of computational geometry that can be applied in a similar manner.

Recipe 8-11 **Creating Smoothed Bezier Paths Using Catmull-Rom Splining**

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

// Get points from Bezier Curve
void getPointsFromBezier(void *info, const CGPathElement *element)
{
    NSMutableArray *bezierPoints =
        (__bridge NSMutableArray *)info;

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

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

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

NSArray *pointsFromBezierPath(UIBezierPath *bpath)
{
    NSMutableArray *points = [NSMutableArray array];
```

From the Library of Eric Berna

```
    CGPathApply(bpath.CGPath,
        (__bridge void *)points, getPointsFromBezier);
    return points;
}

UIBezierPath *smoothedPath(UIBezierPath *bpath, int granularity)
{
    NSArray *points = pointsFromBezierPath(bpath);
    if (points.count < 4) return bpath;

    // This only copies lineWidth. You may want to copy more
    UIBezierPath *smoothedPath = [UIBezierPath bezierPath];
    smoothedPath.lineWidth = bpath.lineWidth;

    // Draw out the first 3 points (0..2)
    [smoothedPath moveToPoint:POINT(0)];
    for (int index = 1; index < 3; index++)
        [smoothedPath addLineToPoint:POINT(index)];

    // Points are interpolated between p1 and p2,
    // starting with 2..3, and moving from there
    for (int index = 4; index < points.count; index++)
    {
        CGPoint p0 = POINT(index - 3);
        CGPoint p1 = POINT(index - 2);
        CGPoint p2 = POINT(index - 1);
        CGPoint p3 = POINT(index);

        // now add n points starting at p1 + dx/dy up until p2
        // using Catmull-Rom splines
        for (int i = 1; i < granularity; i++)
        {
            float t = (float) i * (1.0f / (float) granularity);
            float tt = t * t;
            float ttt = tt * t;

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

From the Library of Eric Berna

```
        [smoothedPath addLineToPoint:p2];
    }

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

    return smoothedPath;
}
```

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 8 and open the project for this recipe.

Recipe: Detecting Circles

In a direct manipulation interface like iOS, you'd imagine that most people could get by just pointing to items onscreen. And yet, circle detection remains one of the most requested gestures. Developers like having people circle items onscreen with their fingers. In the spirit of providing solutions that readers have requested, Recipe 8-12 offers a relatively simple circle detector, which is shown in Figure 8-6.

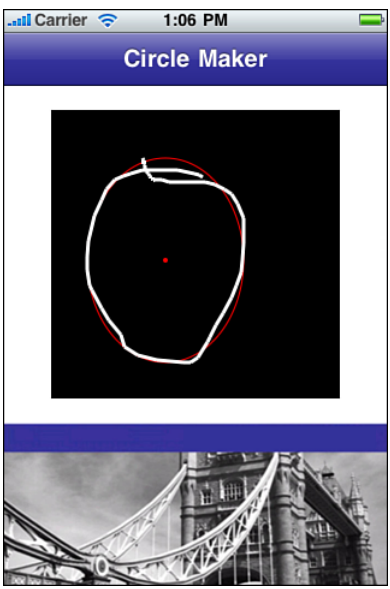


Figure 8-6 The dot and the outer ellipse show the key features of the detected circle.

From the Library of Eric Berna

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

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

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

Recipe 8-12 **Detecting Circles**

```
// Return center of the given rect
CGPoint getRectCenter(CGRect rect)
{
    return CGPointMake(CGRectGetMidX(rect), CGRectGetMidY(rect));
}

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

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

    return dot;
}

// Return distance between two points
float distance (CGPoint p1, CGPoint p2)
{
```

From the Library of Eric Berna

```
    float dx = p2.x - p1.x;
    float dy = p2.y - p1.y;

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

// Return a point with respect to a given origin
CGPoint pointWithOrigin(CGPoint pt, CGPoint origin)
{
    return CGPointMake(pt.x - origin.x, pt.y - origin.y);
}

// Calculate and return least bounding rectangle
CGRect boundingRect(NSArray *points)
{
    CGRect rect = CGRectZero;
    CGRect ptRect;

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

    return rect;
}

#define DX(p1, p2)    (p2.x - p1.x)
#define DY(p1, p2)    (p2.y - p1.y)
#define SIGN(NUM) (NUM < 0 ? (-1) : 1)
#define DEBUG NO

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

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

From the Library of Eric Berna

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

// Test 2: The number of direction changes should be
// limited to near 4
int inflections = 0;
for (int i = 2; i < (points.count - 1); i++)
{
    float dx = DX(POINT(i), POINT(i-1));
    float dy = DY(POINT(i), POINT(i-1));
    float px = DX(POINT(i-1), POINT(i-2));
    float py = DY(POINT(i-1), POINT(i-2));

    if ((SIGN(dx) != SIGN(px)) || (SIGN(dy) != SIGN(py)))
        inflections++;
}

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

// Test 3: The start and end points must be between
// some number of points of each other
float tolerance = [[[UIApplication sharedApplication]
    keyWindow] bounds].size.width / 3.0f;
if (distance(POINT(0), POINT(points.count - 1)) > tolerance)
{
    if (DEBUG) NSLog(@"Start and end points too far apart.");
    return CGRectZero;
}

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

From the Library of Eric Berna

```
        float transitTolerance = distance - 2 * M_PI;
        if (transitTolerance < 0.0f) // fell short of 2 PI
        {
            if (transitTolerance < - (M_PI / 4.0f))
            {
                if (DEBUG) NSLog(@"Transit was too short");
                return CGRectZero;
            }
        }

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

    return circle;
}
@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 8 and open the project for this recipe.

Creating a Custom Gesture Recognizer

It takes little work to transform the code shown in Recipe 8-12 into a custom recognizer. Subclassing `UIGestureRecognizer` enables you to build your own circle recognizer that you can add to views in your applications.

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

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

All recognizers start in the possible state (`UIGestureRecognizerStatePossible`), and then for continuous gestures pass through a series of changed states (`UIGestureRecognizerStateChanged`). Discrete recognizers either succeed in recognizing a gesture (`UIGestureRecognizerStateRecognized`) or fail (`UIGestureRecognizerStateFailed`), as demonstrated in Listing 8-1. The recognizer

From the Library of Eric Berna

sends actions to its target each time you update state *except* when the state is set to `Possible` or `Failed`.

The rather long comments you see in Listing 8-1 belong to Apple, courtesy of the subclass header file. I've included them here because they help explain the roles of the key methods that override their superclass. The `reset` method returns the recognizer back to its quiescent state, allowing it to prepare itself for its next recognition challenge.

The touches began (and so on) methods are called at similar points as their `UIResponder` analogs, allowing you to perform your tests at the same touch life-cycle points. This example waits to check for success or failure until the touches ended callback, and uses the same `testForCircle` method defined in Recipe 8-12.

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

Listing 8-1 **Creating a Gesture Recognizer Subclass**

```
@implementation CircleRecognizer

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

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

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

    if (touches.count > 1)
    {
```

From the Library of Eric Berna

```
        self.state = UIGestureRecognizerStateFailed;
        return;
    }

    points = [NSMutableArray array];
    firstTouchDate = [NSDate date];
    UITouch *touch = [touches anyObject];
    [points addObject: [NSValue valueWithCGPoint:
        [touch locationInView:self.view]]];
}

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

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

Recipe: Using Multitouch

Enabling multitouch interaction in your `UIViews` lets iOS recover and respond to more than one finger touch at a time. Set the `UIView` property `multipleTouchEnabled` to `YES` or override `isMultipleTouchEnabled` for your view. When multitouch is enabled, each touch callback returns an entire set of touches. When that set's count exceeds 1, you know you're dealing with multitouch.

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

When multitouch was first explored on the iPhone, developers did not dream of the freedom and flexibility that multitouch combined with multiple users offered. Adding

From the Library of Eric Berna

multitouch to your games and other applications really opens up not just expanded gestures but also new ways of creating profoundly exciting multiuser experiences, especially on larger screens like the iPad. I encourage you to include multitouch support in your applications wherever it is practical and meaningful.

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

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

Recipe 8-13 adds multitouch to a `UIView` by setting its `multipleTouchEnabled` property and traces the lines that each finger draws. It does this by keeping track of each touch's physical address in memory but without pointing to or retaining the touch per Apple's recommendations. This is, obviously, an oddball approach, but it has worked reliably throughout the history of the SDK. That's because each `UITouch` object persists at a single address throughout the touch-move-release life cycle. By using the physical address as a key, you can distinguish each touch, even as new touches are added or old touches are removed from the screen. Be aware that new touches can start their life cycle via `touchesBegan:withEvent:` independently of others as they move, end, or cancel. Your code should reflect that reality.

Beyond the multitouch additions, this recipe mirrors Recipe 8-10. Each touch grows a separate Bezier path, which is painted in the view's `drawRect` method.

> **Note**
>
> At least one developer has mentioned to me that historically Apple has not always lived up to its guarantee that touches remain the same object throughout their existence.

Recipe 8-13 **Adding Basic Multitouch**

```
@implementation TouchTrackerView
- (void) touchesBegan:(NSSet *) touches withEvent:(UIEvent *) event
{
    // Create a new path dictionary if one does not yet exist
    if (!touchPaths)
        touchPaths = [NSMutableDictionary dictionary];
```

From the Library of Eric Berna

```
    // Register each touch with a new Bezier path
    for (UITouch *touch in touches)
    {
        // Register by touch memory location
        NSString *key = [NSString stringWithFormat:@"%d", touch];
        CGPoint pt = [touch locationInView:self];

        // Establish the new path
        UIBezierPath *path = [UIBezierPath bezierPath];
        path.lineWidth = 4;
        [path moveToPoint:pt];

        [touchPaths setObject:path forKey:key];
    }
}

// Continue tracking each touch path
- (void) touchesMoved:(NSSet *) touches withEvent:(UIEvent *) event
{
    for (UITouch *touch in touches)
    {
        NSString *key = [NSString stringWithFormat:@"%d", touch];
        UIBezierPath *path = [touchPaths objectForKey:key];
        if (!path) break;

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

    [self setNeedsDisplay];
}

// Upon finishing a touch, remove the path
- (void) touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    for (UITouch *touch in touches)
    {
        NSString *key = [NSString stringWithFormat:@"%d", touch];
        [touchPaths removeObjectForKey:key];
    }

    [self setNeedsDisplay];
}

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

From the Library of Eric Berna

```
}

// Draw out each path
- (void) drawRect:(CGRect)rect
{
    for (UIBezierPath *path in [touchPaths allValues])
        [path stroke];

}

// Ensure that multi-touch is enabled
- (id) initWithFrame:(CGRect)frame
{
    if (self = [super initWithFrame:frame])
        self.multipleTouchEnabled = YES;

    return self;
}
@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 8 and open the project for this recipe.

Retaining Touch Paths

Recipe 8-13 erased old paths at the end of their life cycle. That worked well for application bookkeeping but failed when it came to creating a standard drawing application, where you expect to iteratively add elements to a picture. You can easily adapt its drawing code base to continue adding traces into a composite picture without erasing old items, as is shown in Recipe 8-14.

To extend your multitouch path dictionary beyond the lifetime of a single touch, be prepared to create new keys for your existing Bezier paths. That's an important step because of one simple fact: memory gets re-used. A touch that maps to a given address as its key may be overwritten once the user removes that touch from the screen and a new touch begins. You can fix this minor stumbling block by creating new unique keys that won't conflict with other items in your touch dictionary.

The following recipe shows how you can build a unique ID (here, called a "nonce") using a built-in Core Foundation UUID (universally unique identifier) function. This creates a key that is guaranteed not to conflict with any existing key.

The updated touches-ended method reassigns the Bezier path for each touch to a new random unique nonce. This allows the interface to retain all traced paths without interfering with the basic touch-tracking mechanism for building those paths.

From the Library of Eric Berna

These paths accumulate until the application invokes the `clear` method shown here. Clearing empties the `touchPaths` dictionary and redraws the screen.

Recipe 8-14 **Accumulating User Tracings for a Composite Picture**

```
NSString *nonce()
{
    CFUUIDRef theUUID = CFUUIDCreate(NULL);
    NSString *nonceString = (__bridge_transfer NSString *)
        CFUUIDCreateString(NULL, theUUID);
    CFRelease(theUUID);
    return nonceString;
}

- (void) touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    // Move all completed touches out of memory named keys
    for (UITouch *touch in touches)
    {
        NSString *key = [NSString stringWithFormat:@"%d", touch];
        UIBezierPath *path = [touchPaths objectForKey:key];
        [touchPaths removeObjectForKey:key];
        [touchPaths setObject:path forKey:nonce()];
    }

    [self setNeedsDisplay];
}

- (void) clear
{
    [touchPaths removeAllObjects];
    [self setNeedsDisplay];
}
```

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 8 and open the project for this recipe.

Note

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

From the Library of Eric Berna

One More Thing: Dragging from a Scroll View

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

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

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

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

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

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

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

To make this happen, Recipe 8-15 manually adds that directional-movement detection, outside of the built-in gesture detection. In the end, that working-outside-the-box approach provides a major coding win. That's because once the swipe has been detected, the underlying pan gesture recognizer continues to operate. This allows the user to keep moving the swiped object without having to raise his or her finger and retouch the object in question.

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

From the Library of Eric Berna

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

Recipe 8-15 **Dragging Items Out of Scroll Views**

```
@implementation DragView

#define DX(p1, p2)    (p2.x - p1.x)
#define DY(p1, p2)    (p2.y - p1.y)

#define SWIPE_DRAG_MIN 16
#define DRAGLIMIT_MAX 8

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

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

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

From the Library of Eric Berna

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

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

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

    // Handle touch based on recognizer state

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

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

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

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

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

From the Library of Eric Berna

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

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

> **Get This Recipe's Code**
>
> To 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 8 and open the project for this recipe.

Summary

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

- Be concrete. iOS devices have perfectly good touch screens. Why not let your users drag items around the screen or trace lines with their fingers? It adds to the reality and the platform's interactive nature.

From the Library of Eric Berna

- Users typically have five fingers per hand. iPads, in particular, offer a lot of screen real estate. Don't limit yourself to a one-finger interface when it makes sense to expand your interaction into multitouch territory for one or more users, screen space allowing.
- A solid grounding in Quartz graphics and Core Animation will be your friend. Using `drawRect:`, you can build any kind of custom `UIView` presentation you'd like, including text, Bezier curves, scribbles, and so forth.
- If Cocoa Touch doesn't provide the kind of specialized gesture recognizer you're looking for, write your own. It's not that hard, although it helps to be as thorough as possible when considering the states your custom touch might pass through.
- Use multitouch whenever possible, especially when you can expand your application to invite more than one user to touch the screen at a time. Don't limit yourself to one-person, one-touch interactions when a little extra programming will open doors of opportunity for multiuser use.
- Explore! This chapter only touched lightly on the ways you can use direct manipulation in your applications. Use this material as a jumping-off point to explore the full vocabulary of the `UITouch` class.

From the Library of Eric Berna

9

Building and Using Controls

The `UIControl` class provides the basis for many iOS interactive elements, including buttons, text fields, sliders, and switches. These onscreen objects have more in common than just deriving from their ancestor class. Controls all use similar layout paradigms and target-action approaches. Learning to create a single control, no matter how specialized, teaches you how all controls work. Controls may appear visually unique and specialized, but a common development approach runs through them. This chapter introduces controls and their use. You discover how to build and customize controls in a variety of ways. From the prosaic to the obscure, this chapter introduces a range of control recipes you can reuse in your programs.

The `UIControl` Class

In iOS, controls generally refer to the members of a library of prebuilt onscreen objects designed for user interaction. Controls include buttons and text fields, sliders and switches, along with other Apple-supplied objects. A control's role is to transform user interactions into callbacks. Users touch and manipulate controls and in doing so communicate with your application.

The `UIControl` class lies at the root of the control class tree. All controls define a visual interface and implement ways to dispatch messages when users interact with that interface. Controls send those messages using target-action. When you define a new onscreen control, you tell it who receives messages, what messages to send, and when to send those messages.

Kinds of Controls

The standard members of the `UIControl` family include buttons, segmented controls, switches, sliders, page controls, and text fields. Each of these controls can be found in Interface Builder's Object Library (Command-Control-Option-3, View > Utilities > Show Object Library), as shown in Figure 9-1.

From the Library of Eric Berna

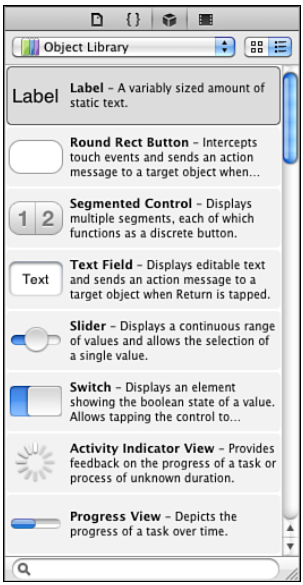


Figure 9-1 Interface Builder includes its available controls in the Object Library.

Control Events

Controls respond primarily to three kinds of events: those based on touch, those based on value, and those based on edits. Table 9-1 lists the full range of event types available to controls.

Table 9-1 **`UIControl` Event Types**

Event	Type	Use
`UIControlEvent TouchDown`	Touch	A touch down event anywhere within a control's bounds.
`UIControlEvent TouchUpInside`	Touch	A touch up event anywhere within a control's bounds. This is the most common event type used for buttons.
`UIControlEvent TouchUpOutside`	Touch	A touch up event that falls strictly outside a control's bounds.
`UIControlEvent TouchDragEnter`	Touch	Events corresponding to drags that cross into or out from the control's bounds.

From the Library of Eric Berna

Table 9-1 **`UIControl` Event Types**

Event	Type	Use
`UIControlEvent TouchDragInside` `UIControlEvent TouchDragOutside`	Touch	Drag events limited to inside the control bounds or to just outside the control bounds.
`UIControlEvent TouchDownRepeat`	Touch	A repeated touch down event with a `tapCount` above 1 (for example, a double-tap).
`UIControlEvent TouchCancel`	Touch	A system event that cancels the current touch. See Chapter 8, "Gestures and Touches," for more details about touch phases and life cycles.
`UIControlEvent AllTouchEvents`	Touch	A mask that corresponds to all the touch events listed so far, used to catch any touch event.
`UIControlEvent ValueChanged`	Value	A user-initiated event that changes the value of a control such as moving a slider's thumb or toggling a switch.
`UIControlEvent EditingDidBegin` `UIControlEvent EditingDidEnd`	Editing	Touches inside or outside a `UITextField`. A touch inside begins the editing session. A touch outside ends it.
`UIControlEvent EditingChanged`	Editing	An editing change to the contents of the `UITextField`.
`UIControlEvent EditingDidEndOn Exit`	Editing	An event that ends an editing session but not necessarily a touch outside its bounds.
`UIControlEvent AllEditingEvents`	Editing	A mask of all editing events.
`UIControlEvent Application Reserved`	Application	Application-specific event range (rarely if ever used).
`UIControlEvent SystemReserved`	System	System-specific event range (rarely if ever used).
`UIControlEvent AllEvents`	Touch, value, editing, application, system	A mask of all touch, value, editing, application, and system events.

For the most part, events break down along the following lines. Buttons use touch events; the single `UIControlEventTouchUpInside` event accounts for nearly all button interaction and is the default event created by Interface Builder connections. Value events (for example, `UIControlEventValueChanged`) correspond to user-initiated adjustments to

From the Library of Eric Berna

segmented controls, switches, sliders, and page controls. When users switch, slide, or tap those objects, the control value changes. `UITextField` objects trigger editing events. Users cause these events by tapping into (or out from) the text field, or by changing the text field contents.

As with all iOS GUI elements, you can lay out controls in Xcode's Interface Builder screen or build them directly in code. This chapter discusses some IB approaches but focuses more intently on code-based solutions. IB layout, once mastered, remains pretty much the same regardless of the item involved. You place an object into the interface, customize it with inspectors, and connect it to other IB objects.

Buttons

`UIButton` instances provide simple onscreen buttons. Users can tap them to trigger a callback via target-action programming. You specify how the button looks, what art it uses, and what text it displays.

iOS offers two ways to build buttons. You can use a precooked button, which includes several predesigned styles, or build a custom button from scratch. The current iOS SDK offers the following precooked types. As you can see, the buttons available are not general purpose. They were added to the SDK primarily for Apple's convenience, not yours. That's because, as a rule, Apple does not add UI features that they do not primarily consume themselves. Nonetheless, you can use these in your programs as needed if you follow Apple's Human Interface Guidelines (HIG). Figure 9-2 shows each button.

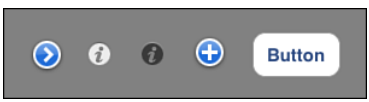


Figure 9-2 iOS SDK offers five precooked button types, which you can access in Interface Builder or build directly into your applications. From left to right, these are the Detail Disclosure button, the Info Light and Info Dark buttons, the Contact Add button, and the Rounded Rectangle.

- **Detail Disclosure**—This is the same round, blue circle with the chevron you see when you add a detail disclosure accessory to table cells. Detail disclosures are used in tables to lead to a screen that shows details about the currently selected cell.
- **Info Light and Info Dark**—These two buttons offer a small circled *i* like you see on a Macintosh's Dashboard widget and are meant to provide access to an information or settings screen. These are used in the Weather and Stocks application to flip the view from one side to the other.
- **Contact Add**—This round, blue circle has a white + in its center and can be seen in the Mail application for adding new recipients to a mail message.

From the Library of Eric Berna

- **Rounded Rectangle**—This button provides a simple onscreen rounded rectangle that surrounds the button text. In its default state it is not an especially attractive button (that is, it's not very "Apple" looking), but it is simple to program and use in your applications.

To use a precooked button in code, allocate it, set its frame, and add a target. Don't worry about adding custom art or creating the overall look of the button. The SDK takes care of all that. For example, here's how to build a simple rounded rectangle button:

```
UIButton *button = [UIButton buttonWithType:UIButtonTypeRoundedRect];
[button setFrame:CGRectMake(0.0f, 0.0f, 80.0f, 30.0f)];
[button setCenter:CGPointMake(160.0f, 208.0f)];
[button setTitle:@"Beep" forState:UIControlStateNormal];
[button addTarget:self action:@selector(playSound)
     forControlEvents:UIControlEventTouchUpInside];
[contentView addSubview:button];
```

To build one of the other standard button types, omit the title line. Rounded rectangles is the only precooked button type that uses a title.

Most buttons use the "touch up inside" trigger, where the user's touch ends inside the button's bounds. iOS UI standards allow a user to cancel a button press by pulling his or her finger off a button before releasing the finger from the screen. The `UIControlEventTouchUpInside` event choice mirrors that standard.

When using a precooked button, you *must* conform to Apple's Human Interface Guidelines on how these buttons can be used. Adding a detail disclosure, for example, to lead to an information page can get your application rejected from the App Store. It might seem a proper extrapolation of the button's role, but if it does not meet the exact wording of how Apple expects the button to be used, it may not pass review. (Obviously, this depends on the reviewer, but you'll be hard pressed to defend an application that violates the Human Interface Guidelines.) To avoid potential issues, you may want to use rounded rectangle and custom buttons wherever possible.

Adding Buttons in Interface Builder

Buttons appear in the Interface Builder library as Rounded Rect Button objects. To use them, drag them into your interface. You can then change them to another button type via the attribute inspector (View > Utility > Show Attributes Inspector, Command-Option-4). A button-type pop-up appears at the top of the inspector, as shown in Figure 9-3. Use this pop-up menu to select the button type.

If your button uses text (such as the word "Button" in Figure 9-2), you can enter that text in the Title field. The Image and Background pull-downs let you choose a primary and background image for the button.

From the Library of Eric Berna

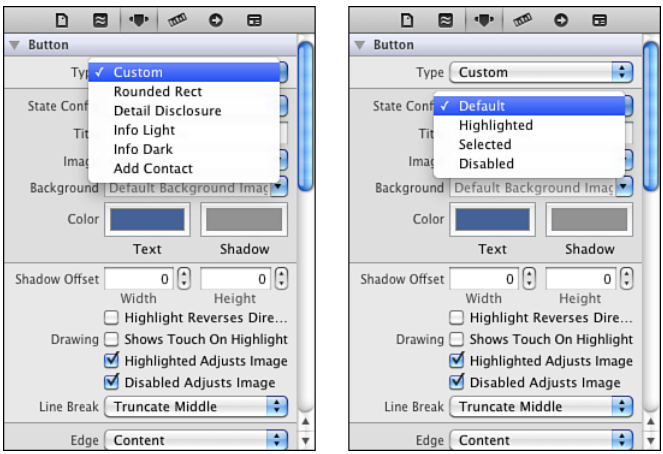


Figure 9-3 Choose your button type from the Type pop-up in the attributes inspector (left). Changes in the Button section apply to the current configuration (right).

Each button provides four configuration settings, which can be seen in Figure 9-3 (right). The four button states are Default (the button in its normal state), Highlighted (when a user is currently touching the button), Selected (an "on" version of the button, for buttons that support toggled states), and Disabled (when the button is unavailable for user interaction).

Changes in the Object Attributes > Button > State Configuration section (that is, the darkened rectangle below the configuration pop-up) apply to the currently selected configuration. You might, for example, use a different button text color for a button in its default state versus its disabled state.

To preview each state, locate the three check boxes in Object Attributes > Control > Content. The Highlighted, Selected, and Enabled options let you set the button state. After previewing, and before you compile, make sure you returned the button to the actual state it needs to be in when you first run the application.

Art

Apart from the precooked button types (disclosure, info, and add contact), you'll want to create buttons using custom art. Figure 9-4 shows a variety of buttons built around the Rounded Rect and Custom button classes.

Figure 9-4 shows that when working with Rounded Rect buttons, you are not limited to just text (Button A). You can add an image along with text (Button B), use an image instead of text (Button F), or even replace the background rounded rectangle style with custom art (Button E), although this latter case does not make a lot of sense in the day-to-day design process; just use a custom button from the get-go.

From the Library of Eric Berna

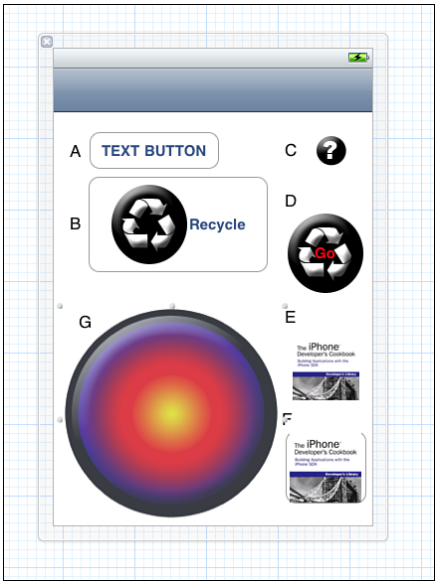


Figure 9-4 These examples show a variety of custom art options for both Rounded Rect and Custom buttons.

Custom buttons have no built-in look. You can make buttons with any size you like (Buttons C and G) and add text (Button D) using the attributes inspector. What Figure 9-4 does not show is that these three buttons also represent other custom design decisions.

Button D uses the same art from Button B. Being a custom button, its text is centered and not displayed on a rounded backsplash. Beyond that, there's no big difference between the B layout and the D layout. The button relies on the default highlighting provided by Interface Builder and the `UIButton` class.

Button C represents a button created for highlighting on touch. Its relatively small size allows it to work with Button Attributes > Button > Shows Touch On Highlight. When touched, the button reveals a glowing halo. This halo is approximately 55 by 55 pixels in size. Buttons larger than about 40 by 40 pixels cannot effectively use this visual pop.

What can't be seen in this static screenshot is that Button G was built to display an alternate image when pushed. Setting a second image in Button Attributes > Button > Highlighted State Configuration lets a button change its look on touch. For Button G, that image shows the same button but pushed into an indented position.

Connecting Buttons to Actions

When you Control-drag (right-drag) from a button to an IB object such as the File's Owner view controller in the XIB editor, IB presents a pop-up menu of actions to

From the Library of Eric Berna

choose from. These actions are polled from the target object's available IBActions. Connecting to an action creates a target-action pair for the button's touch up inside event.

Alternatively, as Figure 9-5 shows, you can Control-click (right-click) the button, scroll down to Touch Up Inside, and drag from the unfilled dot to the target you want to connect to (in this case, the File's Owner object). The same pop-up menu appears with its list of available actions. Select the one you want to use to finish defining the target-action callback.

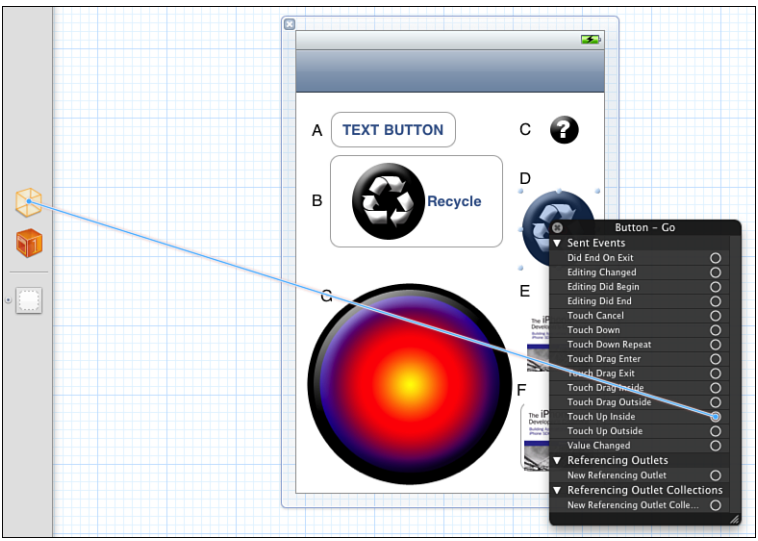


Figure 9-5 Control-clicking (right-clicking) a `UIControl` in Interface Builder reveals a table of events that you can connect to a target. Available actions appear in a pop-up menu after you drag out the connection.

> **Note**
>
> As Figure 9-5 demonstrates, you can create, edit, and interact with XIB files in the latest versions of Xcode 4, just as you did prior to iOS 5. Although new Xcode projects default to storyboard layout, you may add new interface files using either the File Template Library or File > New > New File > iOS > User Interface.

Buttons That Are Not Buttons

In Interface Builder, you also encounter buttons that look like views and act like views but are not, in fact, views. Bar button items (`UIBarButtonItem`) store the properties of toolbar and navigation bar buttons but are not buttons themselves.

From the Library of Eric Berna

Building Custom Buttons in Xcode

When using the `UIButtonTypeCustom` style, you supply all button art. The number of images depends on how you want the button to work. For a simple pushbutton, you might add a single background image and vary the label color to highlight when the button is pushed. For a toggle-style button, you might use four images: for the "off" state in a normal presentation, the "off" state when highlighted (that is, pressed), and two more for the "on" state. You choose and design the interaction details, making sure to add local state (the Boolean `isOn` instance variable in Recipe 9-1) to extend a simple pushbutton to a toggle.

Recipe 9-1 builds a button that toggles on and off, demonstrating the detail that goes into building custom buttons. When tapped, the button switches its art from green (on) to red (off), or from red to green. This allows your (noncolorblind) users to instantly identify a current state. The displayed text reinforces the state setting. Figure 9-6 (left) shows the button created by this recipe.

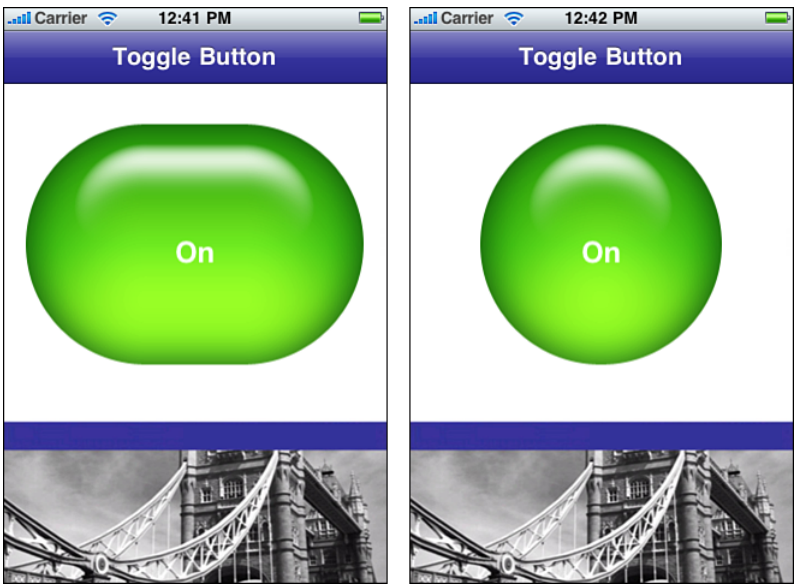


Figure 9-6 Use `UIImage` stretching to resize art for arbitrary button widths. Set the left cap width to specify where the stretching can take place.

The `UIImage` resizable image calls in this recipe play an important role in button creation. Resizable images enable you to create buttons of arbitrary width, turning circular art into lozenge-shaped buttons. You specify the caps (that is, the art that should not be stretched). In this case, the cap is 110 pixels wide on the left and right. If you were to change the button width from the 300 pixels used in this recipe to 220, the button loses the middle stretch, as shown in Figure 9-6 (right).

From the Library of Eric Berna

Recipe 9-1 **Building a `UIButton` That Toggles On and Off**

```
#define CAPWIDTH    110.0f
#define INSETS      (UIEdgeInsets){0.0f, CAPWIDTH, 0.0f, CAPWIDTH}
#define BASEGREEN   [[UIImage imageNamed:@"green.png"] \
    resizableImageWithCapInsets:INSETS]
#define ALTGREEN    [[UIImage imageNamed:@"green2.png"] \
    resizableImageWithCapInsets:INSETS]
#define BASERED     [[UIImage imageNamed:@"red.png"] \
    resizableImageWithCapInsets:INSETS]
#define ALTRED      [[UIImage imageNamed:@"red2.png"] \
    resizableImageWithCapInsets:INSETS]

- (void) toggleButton: (id) sender
{
    if ((isOn = !isOn))
    {
        [button setTitle:@"On"
             forState:UIControlStateNormal];
        [button setTitle:@"On"
             forState:UIControlStateHighlighted];
        [button setBackgroundImage:BASEGREEN
             forState:UIControlStateNormal];
        [button setBackgroundImage:ALTGREEN
             forState:UIControlStateHighlighted];
    }
    else
    {
        [button setTitle:@"Off"
            forState:UIControlStateNormal];
        [button setTitle:@"Off"
            forState:UIControlStateHighlighted];
        [button setBackgroundImage:BASERED
            forState:UIControlStateNormal];
        [button setBackgroundImage:ALTRED
            forState:UIControlStateHighlighted];
    }
}

- (void) loadView
{
    [super loadView];
    self.view.backgroundColor = [UIColor whiteColor];

    // Create a button sized to our art
    button = [UIButton buttonWithType:UIButtonTypeCustom];
    button.frame = CGRectMake(0.0f, 0.0f, 300.0f, 233.0f);
```

From the Library of Eric Berna

```
    // Set up the button alignment properties
    button.contentVerticalAlignment =
        UIControlContentVerticalAlignmentCenter;
    button.contentHorizontalAlignment =
        UIControlContentHorizontalAlignmentCenter;

    // Set the font and color
    [button setTitleColor:[UIColor whiteColor]
        forState:UIControlStateNormal];
    [button setTitleColor:[UIColor lightGrayColor]
        forState:UIControlStateHighlighted];
    button.titleLabel.font = [UIFont boldSystemFontOfSize:24.0f];

    // Add action
    [button addTarget:self action:@selector(toggleButton:)
        forControlEvents: UIControlEventTouchUpInside];

    // Place the button into the view and initialize its art
    [self.view addSubview:button];
    [self toggleButton:button];
}
```

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 9 and open the project for this recipe.

Multiline Button Text

The button's `titleLabel` property allows you to modify title attributes, including its font and line break mode. Here, the font is set to a very large value (basically ensuring that the text needs to wrap to display correctly) and used with word wrap and centered alignment:

```
button.titleLabel.font = [UIFont boldSystemFontOfSize:36.0f];
[button setTitle:@"Lorem Ipsum Dolor Sit" forState:
    UIControlStateNormal];
button.titleLabel.textAlignment = UITextAlignmentCenter;
button.titleLabel.lineBreakMode = UILineBreakModeWordWrap;
```

By default, button labels stretch from one end of your button to the other. This means that text may extend farther out than you might otherwise want, possibly beyond the edges of your button art. To fix this problem, you can force carriage returns in word wrap mode by embedding new line literals (that is, `\n`) into the text. This allows you to control how much text appears on each line of the button title.

From the Library of Eric Berna

Adding Animated Elements to Buttons

When working with buttons, you can creatively layer art in front of or behind them. Use the standard `UIView` hierarchy to do this, making sure to disable user interaction for any view that might otherwise obscure your button (`setUserInteractionEnabled:NO`). Figure 9-7 shows what happens when you combine semitranslucent button art with an animated `UIImageView` behind it. The image view contents "leak" through to the viewer, enabling you to add live animation elements to the button.

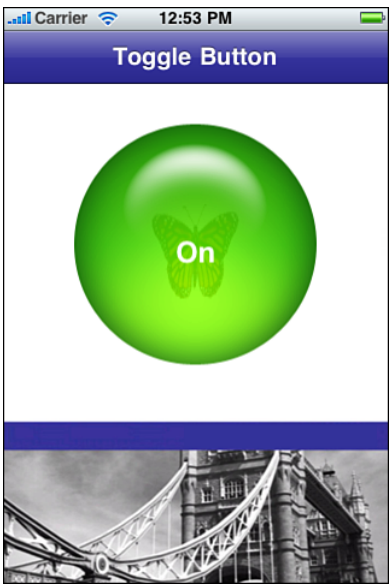


Figure 9-7 Combine semitranslucent button art with animated `UIImageViews` to build eye-catching UI elements. In this concept, the butterfly flaps "within" the button.

Recipe: Animating Button Responses

There's more to `UIControl` instances than frames and target-action. All controls inherit from the `UIView` class. This means you can use `UIView` animation blocks when working with controls just as you would with standard views. Recipe 9-2 builds a toggle switch that zooms itself whenever a user touches it, returning to its original size when the touch leaves the control.

This recipe creates a livelier interaction element that helps focus greater attention on the control in question.

From the Library of Eric Berna

Recipe 9-2 **Adding `UIView` Animation Blocks to Controls**

```
- (void) zoomButton: (id) sender
{
    // Slightly enlarge the button
    [UIView animateWithDuration:0.2f animations:^{
         button.transform = CGAffineTransformMakeScale(1.1f, 1.1f);}];
}

- (void) relaxButton: (id) sender
{
    // Return the button to its normal size
    [UIView animateWithDuration:0.2f animations:^{
         button.transform = CGAffineTransformIdentity;}];
}

- (void) toggleButton: (UIButton *) button
{
    if (isOn = !isOn)
    {
        [button setTitle:@"On" forState:UIControlStateNormal];
        [button setTitle:@"On"
            forState:UIControlStateHighlighted];
        [button setBackgroundImage:BASEGREEN
            forState:UIControlStateNormal];
        [button setBackgroundImage:ALTGREEN
            forState:UIControlStateHighlighted];
    }
    else
    {
        [button setTitle:@"Off" forState:UIControlStateNormal];
        [button setTitle:@"Off"
            forState:UIControlStateHighlighted];
        [button setBackgroundImage:BASERED
            forState:UIControlStateNormal];
        [button setBackgroundImage:ALTRED
            forState:UIControlStateHighlighted];
    }

    [self relaxButton:button];
}

- (void) viewDidLoad
{
    // Create a button sized to our art
    UIButton *button =
        [UIButton buttonWithType:UIButtonTypeCustom];
```

From the Library of Eric Berna

```
    button.frame = CGRectMake(0.0f, 0.0f, 300.0f, 233.0f);
    button.tag = BUTTONTAG;

    // Set the font and color
    [button setTitleColor:[UIColor whiteColor]
        forState:UIControlStateNormal];
    [button setTitleColor:[UIColor lightGrayColor]
        forState:UIControlStateHighlighted];
    button.titleLabel.font = [UIFont boldSystemFontOfSize:24.0f];

    // Add actions to zoom and relax the button and to toggle it
    [button addTarget:self action:@selector(zoomButton:)
        forControlEvents: UIControlEventTouchDown |
        UIControlEventTouchDragInside | UIControlEventTouchDragEnter];
    [button addTarget:self action:@selector(relaxButton:)
        forControlEvents: UIControlEventTouchDragExit |
        UIControlEventTouchCancel | UIControlEventTouchDragOutside];
    [button addTarget:self action:@selector(toggleButton:)
        forControlEvents: UIControlEventTouchUpInside];

    // For tracking the two states
    isOn = NO;
    [self toggleButton:button];

    // Place the button into the view
    [self.view addSubview:button];
}
```

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 9 and open the project for this recipe.

Recipe: Adding a Slider With a Custom Thumb

`UISlider` instances provide a control allowing users to choose a value by sliding a knob (called its "thumb") between its left and right extent. You'll have seen sliders in the iPod/Music application, where the class is used to control volume.

Slider values default to 0.0 for the minimum and 1.0 for the maximum, although you can easily customize this in the Interface Builder attributes inspector or by setting the `minimumValue` and `maximumValue` properties. If you want to stylize the ends of the control, you can add in a related pair of images (`minimumValueImage` and `maximumValueImage`) that reinforce those settings. For example, you might show a snowman on one end and a steaming cup of tea on the other for a slider that controls temperature settings. You can also set the color of the track before and after the thumb. Adjust the `minimumTrackTintColor` and `maximumTrackTintColor` properties.

From the Library of Eric Berna

The slider's `continuous` property controls whether a slider continually sends value updates as a user drags the thumb. When set to `NO` (the default is `YES`), the slider only sends an action event when the user releases the thumb.

Customizing `UISlider`

In addition to setting minimum and maximum images, the `UISlider` class lets you directly update its thumb component. You can set a thumb to whatever image you like by calling `setThumbImage:forState:`. Recipe 9-3 takes advantage of this option to dynamically build thumb images on the fly, as shown in Figure 9-8. The indicator bubble appears above the user's finger as part of the custom-built thumb. This bubble provides instant feedback both textually (the number inside the bubble) and graphically (the shade of the bubble reflects the slider value, moving from black to white as the user drags).

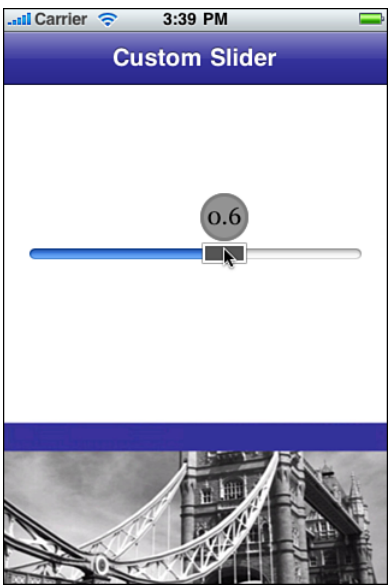


Figure 9-8 Core Graphics/Quartz calls enable this slider's thumb image to dim or brighten based on the current slider value. The text inside the thumb bubble mirrors that value.

This kind of dynamically built feedback could be based on any kind of data. You might grab values from onboard sensors or make calls out to the Internet just as easily as you use the user's finger movement with a slider. No matter what live update scheme you use, dynamic updates are certainly graphics intensive—but it's not as expensive as you might fear. The Core Graphics calls are fast, and the memory requirements for the thumb-sized images are minimal.

From the Library of Eric Berna

This particular recipe assigns two thumb images to the slider. The bubble appears only when the slider is in use, for its `UIControlStateHighlighted`. In its normal state, namely `UIControlStateNormal`, only the smaller rectangular thumb appears. Users can tap on the thumb to review the current setting. The context-specific feedback bubble mimics the letter highlights on the standard iOS keyboard.

To accommodate these changes in art, the slider updates its frame at the start and end of each gesture. On being touched (`UIControlEventTouchDown`), the frame expands by 60 pixels in height. This extra space provides enough room to show the expanded thumb during interaction.

When the finger is removed from the screen (`UIControlEventTouchUpInside` or `UIControlEventTouchUpOutside`), the slider returns to its previous dimensions. This restores space to other onscreen objects, ensuring that the slider will not activate unless a user directly touches it.

Adding Efficiency

This recipe stores a previous value for the slider to minimize the overall computational burden on iOS. It updates the thumb with a new custom image when the slider has changed by at least 0.1, or 10% in value. You can omit this check, if you want, and run the recipe with full live updating. When tested, this provided reasonably fast updates, even on a first-generation iPod touch unit. It also avoids any issues at the ends of the slider—namely when the thumb gets caught at 0.9 and won't update properly to 1.0. In this recipe, a hard-coded workaround for values above 0.98 handles that particular situation by forcing updates.

Appearance Proxies

Tired of tinting every button, navigation bar, or slider in your application by hand? Proxies allow you to customize the appearance of all members of a view class. Instead of updating an instance's properties, you perform the same updates to the proxy. These calls set the shared appearance for slider instances:

```
[[UISlider appearance] setMinimumTrackTintColor:[UIColor blackColor]];
[[UISlider appearance] setMaximumTrackTintColor:[UIColor grayColor]];
```

At times you may only want to apply appearance proxies on a container-by-container basis. For example, you might want all navigation bar buttons to appear blue but not affect bar buttons in toolbars. In that case, you use a container-aware appearance proxy:

```
[[UIBarButtonItem appearanceWhenContainedIn:
    [UINavigationBar class], nil] setTintColor:[UIColor blueColor]];
```

The container list specifies a Boolean AND condition. For example, this proxy applies only to navigation bars that appear in popover controllers:

```
[[UIBarButtonItem appearanceWhenContainedIn:
    [UINavigationBar class], [UIPopoverController class], nil]
    setTintColor:[UIColor blueColor]];
```

From the Library of Eric Berna

Properties applied to actual instances "win." In the following code, the left button ("Hello") defaults to purple but the right button ("World") overrides that default to appear in green:

```
UIBarButtonItem *hello = BARBUTTON(@"Hello", nil);
UIBarButtonItem *world = BARBUTTON(@"World", nil);
world.tintColor = [UIColor greenColor];

UINavigationItem *navigationItem =
    [[UINavigationItem alloc] initWithTitle@""];
navigationItem.leftBarButtonItem = hello;
navigationItem.rightBarButtonItem = world;

[[UIBarButtonItem appearanceWhenContainedIn:
    [UINavigationBar class], nil]
    setTintColor:[UIColor purpleColor]];
```

The `UI_APPEARANCE_SELECTOR` tag marks all properties that can be affected by appearance proxies. You'll find these marks in the `UIKit` header files. Here's an example:

```
@property(nonatomic,retain) UIColor *tintColor
    __OSX_AVAILABLE_STARTING(__MAC_NA,__IPHONE_5_0)
    UI_APPEARANCE_SELECTOR;
```

Beware of guessing. Setting an appearance proxy for a view's `backgroundColor` property does not throw either a compile-time or runtime error but it can seriously mess up your application's presentation. Background color is not authorized as an appearance selector. At the time this book was being written, the following classes are the only ones that support appearance proxies:

- `UIBarButtonItem`
- `UIBarItem`
- `UINavigationBar`
- `UIProgressView`
- `UISearchBar`
- `UISegmentedControl`
- `UISlider`
- `UISwitch`
- `UITabBar`
- `UITabBarItem`
- `UIToolbar`

From the Library of Eric Berna

Recipe 9-3 **Building Dynamic Slider Thumbs**

```
// Draw centered text into the context
void centerText(CGContextRef context, NSString *fontname,
    float textsize, NSString *text, CGPoint point, UIColor *color)
{
    CGContextSaveGState(context);
    CGContextSelectFont(context, [fontname UTF8String],
        textsize, kCGEncodingMacRoman);

    // Retrieve the text width without actually drawing anything
    CGContextSaveGState(context);
    CGContextSetTextDrawingMode(context, kCGTextInvisible);
    CGContextShowTextAtPoint(context, 0.0f, 0.0f,
        [text UTF8String], text.length);
    CGPoint endpoint = CGContextGetTextPosition(context);
    CGContextRestoreGState(context);

    // Query for size to recover height. Width is less reliable
    CGSize stringSize = [text sizeWithFont:
        [UIFont fontWithName:fontname size:textsize]];

    // Draw the text
    [color setFill];
    CGContextSetShouldAntialias(context, true);
    CGContextSetTextDrawingMode(context, kCGTextFill);
    CGContextSetTextMatrix (context,
        CGAffineTransformMake(1, 0, 0, -1, 0, 0));
    CGContextShowTextAtPoint(context, point.x - endpoint.x / 2.0f,
        point.y + stringSize.height / 4.0f,
        [text UTF8String], text.length);
    CGContextRestoreGState(context);
}

// Create a thumb image using a grayscale/numeric level
UIImage *thumbWithLevel (float aLevel)
{
    float INSET_AMT = 1.5f;
    CGRect baseRect = CGRectMake(0.0f, 0.0f, 40.0f, 100.0f);
    CGRect thumbRect = CGRectMake(0.0f, 40.0f, 40.0f, 20.0f);

    UIGraphicsBeginImageContext(baseRect.size);
    CGContextRef context = UIGraphicsGetCurrentContext();

    // Create a filled rect for the thumb
    [[UIColor darkGrayColor] setFill];
    CGContextAddRect(context,
```

From the Library of Eric Berna

```
        CGRectInset(thumbRect, INSET_AMT, INSET_AMT));
    CGContextFillPath(context);

    // Outline the thumb
    [[UIColor whiteColor] setStroke];
    CGContextSetLineWidth(context, 2.0f);
    CGContextAddRect(context,
        CGRectInset(thumbRect, 2.0f * INSET_AMT, 2.0f * INSET_AMT));
    CGContextStrokePath(context);

    // Create a filled ellipse for the indicator
    CGRect ellipseRect = CGRectMake(0.0f, 0.0f, 40.0f, 40.0f);
    [[UIColor colorWithWhite:aLevel alpha:1.0f] setFill];
    CGContextAddEllipseInRect(context, ellipseRect);
    CGContextFillPath(context);

    // Label with a number
    NSString *numstring = [NSString stringWithFormat:@"%0.1f", aLevel];
    UIColor *textColor = (aLevel > 0.5f) ?
        [UIColor blackColor] : [UIColor whiteColor];
    centerText(context, @"Georgia", 20.0f, numstring,
        CGPointMake(20.0f, 20.0f), textColor);

    // Outline the indicator circle
    [[UIColor grayColor] setStroke];
    CGContextSetLineWidth(context, 3.0f);
    CGContextAddEllipseInRect(context,
        CGRectInset(ellipseRect, 2.0f, 2.0f));
    CGContextStrokePath(context);

    // Build and return the image
    UIImage *theImage = UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();
    return theImage;
}

// Return a base thumb image without the bubble
UIImage *simpleThumb()
{
    float INSET_AMT = 1.5f;
    CGRect baseRect = CGRectMake(0.0f, 0.0f, 40.0f, 100.0f);
    CGRect thumbRect = CGRectMake(0.0f, 40.0f, 40.0f, 20.0f);

    UIGraphicsBeginImageContext(baseRect.size);
    CGContextRef context = UIGraphicsGetCurrentContext();
```

From the Library of Eric Berna

```
    // Create a filled rect for the thumb
    [[UIColor darkGrayColor] setFill];
    CGContextAddRect(context,
        CGRectInset(thumbRect, INSET_AMT, INSET_AMT));
    CGContextFillPath(context);

    // Outline the thumb
    [[UIColor whiteColor] setStroke];
    CGContextSetLineWidth(context, 2.0f);
    CGContextAddRect(context,
        CGRectInset(thumbRect, 2.0f * INSET_AMT, 2.0f * INSET_AMT));
    CGContextStrokePath(context);

    // Retrieve the thumb
    UIImage *theImage = UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();
    return theImage;
}

// Update the thumb images as needed
- (void) updateThumb: (UISlider *) aSlider
{
    // Only update the thumb when registering significant
    // changes, i.e. 10%
    if ((slider.value < 0.98f) &&
       (ABS(slider.value - previousValue) < 0.1f)) return;

    // create a new custom thumb image for the highlighted state
    UIImage *customimg = thumbWithLevel(slider.value);
    [slider setThumbImage: customimg
        forState: UIControlStateHighlighted];
    previousValue = slider.value;
}

// Expand the slider to accommodate the bigger thumb
- (void) startDrag: (UISlider *) aSlider
{
    slider.frame = CGRectInset(slider.frame, 0.0f, -30.0f);
}

// At release, shrink the frame back to normal
- (void) endDrag: (UISlider *) aSlider
{
    slider.frame = CGRectInset(slider.frame, 0.0f, 30.0f);
}

- (void) loadView
```

From the Library of Eric Berna

```
{
    [super loadView];

    // Set global UISlider appearance attributes
    [[UISlider appearance]
        setMinimumTrackTintColor:[UIColor blackColor]];
    [[UISlider appearance]
        setMaximumTrackTintColor:[UIColor grayColor]];
    // Initialize slider settings
    previousValue = -99.0f;

    // Create slider
    slider = [[UISlider alloc]
        initWithFrame:(CGRect){.size=CGSizeMake(200.0f, 40.0f)}];
    [slider setThumbImage:simpleThumb()
        forState:UIControlStateNormal];
    slider.value = 0.0f;

    // Create the callbacks for touch, move, and release
    [slider addTarget:self action:@selector(startDrag:)
        forControlEvents:UIControlEventTouchDown];
    [slider addTarget:self action:@selector(updateThumb:)
        forControlEvents:UIControlEventValueChanged];
    [slider addTarget:self action:@selector(endDrag:)
        forControlEvents:UIControlEventTouchUpInside |
        UIControlEventTouchUpOutside];

// Present the slider
[self.view addSubview:slider];
[self performSelector:@selector(updateThumb:)
    withObject:slider afterDelay:0.1f];
}
```

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 9 and open the project for this recipe.

Recipe: Creating a Twice-Tappable Segmented Control

The `UISegmentedControl` class presents a multiple-button interface, where users can choose one choice out of a group. The control provides two styles of use. In its normal radio-button-style mode, a button once selected remains selected. Users can tap on other buttons, but they cannot generate a new event by retapping their existing choice. The alternative momentary style lets users tap on each button as many times as desired but

From the Library of Eric Berna

stores no state about a currently selected item. It provides no highlights to indicate the most recent selection.

Recipe 9-4 builds a hybrid approach. It allows users to see their currently selected option and to reselect that choice if needed. This is not the way segmented controls normally work. There are times, though, when you want to generate a new result on reselection (as in momentary mode) while visually showing the most recent selection (as in radio button mode).

Unfortunately, "obvious" solutions to create this desired behavior don't work. You cannot add target-action pairs that detect `UIControlEventTouchUpInside`. `UIControlEventValueChanged` is the only control event generated by `UISegmentedControl` instances. (You can easily test this yourself by adding a target-action pair for touch events.)

Here is where subclassing comes in to play. It's relatively simple to create a new class based on `UISegmentedControl` that does respond to that second tap. Recipe 9-4 defines that class. Its code works by detecting when a touch has occurred, operating independently of the segmented control's internal touch handlers that are subclassed from `UIControl`.

Segment switches remain unaffected; they'll continue to update and switch back and forth as users tap them. Unlike the parent class, here touches on an already-touched segment continue to do something. In this case, they request that the object's delegate produce the `performSegmentAction` method.

Don't add target-action pairs to your segmented controllers the way you'd normally do. Because all touch down events are detected, target-actions for value-changed events would add a second callback and trigger twice whenever you switched segments. Instead, implement the delegate callback and let object delegation handle the updates.

Recipe 9-4 **Creating a Segmented Control Subclass That Responds to a Second Tap**

```
@class DoubleTapSegmentedControl;

@protocol DoubleTapSegmentedControlDelegate <NSObject>
- (void) performSegmentAction: (DoubleTapSegmentedControl *) aDTSC;
@end

@interface DoubleTapSegmentedControl : UISegmentedControl
{
    id __weak <DoubleTapSegmentedControlDelegate> delegate;
}
@property (nonatomic, weak) id delegate;
@end

@implementation DoubleTapSegmentedControl
@synthesize delegate;

- (void) touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
```

From the Library of Eric Berna

```
    [super touchesBegan:touches withEvent:event];
    if (self.delegate)
        [self.delegate performSegmentAction:self];
}
@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 9 and open the project for this recipe.

Recipe: Subclassing `UIControl`

`UIKit` provides many prebuilt controls that you can use directly in your applications. There are buttons and switches and sliders and more. But why stop there? You don't have to limit yourself to Apple-supplied items. Why not create your own?

Recipe 9-5 demonstrates how to subclass `UIControl` to build new controls from scratch. This example creates a simple color picker. When run, it lets the user select a color by touching or dragging within the control. As the user traces left and right, the color changes its hue. Up and down movements adjust the color's saturation. The brightness and alpha levels for the color are fixed at 100%.

This is a really simple control to work with, because there's not much interaction involved other than retrieving the x and y coordinates of the touch. It provides a basic example that demonstrates most of the development issues involved in subclassing `UIControl`.

So why build custom controls? There are several reasons that motivate doing so. First, you can match your own design style. Elements that you place into your interface can and should match your application's aesthetics. If Apple's prebuilt switches, sliders, and other GUI elements don't provide a natural fit into your interface, a custom-built control can satisfy your application's needs without sacrificing cohesive design.

Second, you can create controls that Apple didn't think of providing. From selecting ratings by swiping through a series of stars, or choosing a color from a set of pop-up crayons, custom controls allow you to interact with the user beyond the prepackaged buttons and switches in the SDK. It's easy to build unique eye-catching interactive elements by subclassing `UIControl`.

Finally, custom controls allow you to add features that you cannot access directly or through subclassing. Take Apple's `UISwitch` control, for example. It limits your choices of font, of background color, of size, and of text. With relatively little work, you can build your own switches from the ground up, allowing you to adjust their presentation exactly as you wish.

Keep in mind when you do so that it helps to keep your items visually distinct so you don't run afoul of Human Interface Guideline issues. When you do use lookalike items, you may want to add a note to Apple when submitting apps to the App Store. Make it

From the Library of Eric Berna

clear that you have created a new class rather than using private APIs or otherwise accessing their objects in a manner that's not App Store safe. Even then, you may be rejected for creating items that could potentially "confuse" the end user.

Creating `UIControl`s

The process of building a custom `UIControl` generally involves four distinct steps. As Recipe 9-5 demonstrates, you begin by subclassing `UIControl` to create a new custom class. In that class, you lay out the visual look of the control in your initialization. Next, you build methods to track and interpret touches, and finish by generating events and visual feedback.

Nearly all controls offer a value of some kind. For example, switches have "`isOn`," sliders have a floating point "value," and text fields offer "text." The kinds of values you provide with a custom control are arbitrary. They can be integers, floats, strings, or even (as in Recipe 9-5) a color.

In the case of Recipe 9-5, the control layout is basically a colored rectangle. More complex controls require more complex layout, but even a simple layout like the one shown here can function to provide all the touch interaction space and feedback needed.

Tracking Touches

`UIControl` instances use an embedded method set to work with touches. These methods allow the control to track touches throughout their interaction with the control object:

- **`beginTrackingWithTouch:withEvent:`**—Gets called when a touch enters a control's bounds
- **`continueTrackingWithTouch:withEvent:`**—Follows the touch with repeated calls as the touch remains within the control bounds
- **`endTrackingWithTouch:withEvent:`**—Handles the last touch for the event
- **`cancelTrackingWithEvent:`**—Manages a touch cancellation

Add your custom control logic by implementing any or all these methods in a `UIControl` subclass. Recipe 9-5 uses the begin and continue versions to locate the user touch and track it until the touch is lifted or otherwise leaves the control.

Dispatching Events

Controls use target-action pairs to communicate changes triggered by events. When you build a new control, you must decide what kind of events your object will generate and add code to trigger those events.

Add a dispatch message to your custom control by calling `sendActionsForControlEvents:`. This method lets you send an event (for example, `UIControlEventValueChanged`) to the specified target. Controls transmit these updates by messaging the `UIApplication` singleton. As Apple notes, the application acts as the centralized dispatch point for all messages.

From the Library of Eric Berna

Always try to create as complete a control vocabulary as possible, no matter how simple your class. Most control development cannot anticipate exactly how the class will be used in the future. Overdesigning your events provides the flexibility for future use. Recipe 9-5 dispatches a wide range of events for what is, after all, a very simple control.

Where you dispatch your events depends a lot on the control you end up building. Switch controls, for example, are really only interested in touch up events, which is when their value changes. Sliding controls, in contrast, center on touch movement and require continuing updates as the control tracks finger movement. Adjust your coding accordingly, and be mindful of presenting appropriate visual changes during all parts of your touch cycle.

Recipe 9-5 **Building a Custom Color Control**

```
@implementation ColorControl
@synthesize value; // value is a color

- (id)initWithFrame:(CGRect)frame {
    if (!(self = [super initWithFrame:frame]))
        return nil;

    value = nil;
    self.backgroundColor = [UIColor grayColor];

    return self;
}

- (void) updateColorFromTouch: (UITouch *) touch
{
    // Calculate hue and saturation
    CGPoint touchPoint = [touch locationInView:self];
    float hue = touchPoint.x / self.frame.size.width;
    float saturation = touchPoint.y / self.frame.size.height;

    // Update the color value and change background color
    self.value = [UIColor colorWithHue:hue
        saturation:saturation brightness:1.0f alpha:1.0f];
    self.backgroundColor = self.value;
    [self sendActionsForControlEvents:UIControlEventValueChanged];
}

// Continue tracking touch in control
- (BOOL) continueTrackingWithTouch: (UITouch *) touch
    withEvent: (UIEvent *) event
{
    // Test if drag is currently inside or outside
    CGPoint touchPoint = [touch locationInView:self];
```

From the Library of Eric Berna

```
    if (CGRectContainsPoint(self.frame, touchPoint))
        [self sendActionsForControlEvents:
            UIControlEventTouchDragInside];
    else
        [self sendActionsForControlEvents:
            UIControlEventTouchDragOutside];

    // Update color value
    [self updateColorFromTouch:touch];

    return YES;
}

// Start tracking touch in control
- (BOOL) beginTrackingWithTouch: (UITouch *) touch
    withEvent: (UIEvent *) event
{
    // Touch Down
    [self sendActionsForControlEvents:UIControlEventTouchDown];

    // Update color value
    [self updateColorFromTouch:touch];

    return YES;
}

// End tracking touch
- (void) endTrackingWithTouch: (UITouch *)touch
    withEvent: (UIEvent *)event
{
    // Test if touch ended inside or outside
    CGPoint touchPoint = [touch locationInView:self];
    if (CGRectContainsPoint(self.bounds, touchPoint))
        [self sendActionsForControlEvents:UIControlEventTouchUpInside];
    else
        [self sendActionsForControlEvents:UIControlEventTouchUpOutside];

    // Update color value
    [self updateColorFromTouch:touch];
}

// Handle touch cancel
- (void)cancelTrackingWithEvent: (UIEvent *) event
{
    [self sendActionsForControlEvents:UIControlEventTouchCancel         oxx];
}
@end
```

From the Library of Eric Berna

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 9 and open the project for this recipe.

Working with Switches and Steppers

The `UISwitch` object offers a simple ON/OFF toggle that lets users choose a Boolean value. (The switch internationalizes to 1/0 for most non-English localizations.) The switch object contains a single (settable) value property, called `on`. This returns either `YES` or `NO`, depending on current state of the control. You can programmatically update a switch's value by changing the property value directly or calling `setOn:animated:`, which offers a way to animate the change.

```
- (void) didSwitch: (UISwitch *) theSwitch
{
    self.title = [NSString stringWithFormat:@"%
        theSwitch.on ? @"On" : @"Off"];
}

- (void) viewDidAppear: (BOOL) animated
{
    // Create the switch
    UISwitch *theSwitch = [[[UISwitch alloc] init] autorelease];

    // Trigger on value changes
    [theSwitch addTarget:self action:@selector(didSwitch:)
       forControlEvents:UIControlEventValueChanged];

    [self.view addSubview:theSwitch];

    // Initialize to "off"
    theSwitch.on = NO;
    self.title = @"Off";
}
```

In this example, when the switch updates, it changes the view controller's title. Interface Builder offers relatively few options for working with a switch. You can enable it and set its initial value, but beyond that there's not much to customize. A switch produces a value-changed event when a user adjusts it.

Note

Do not name `UISwitch` instances as `switch`. Recall that `switch` is a reserved C word; it is used for conditional statements. This simple oversight has tripped up many iOS developers.

From the Library of Eric Berna

The `UIStepper` class provides an alternative to sliders and switches. Sliders offer a continuous range of values; switches offer a simple Boolean on/off choice. Steppers fall somewhat in the middle. Instances present a pair of buttons, one labeled - and the other labeled +. These iteratively increment or decrement its `value` property.

You generally want to assign a range to the control by setting its `minimumValue` and `maximumValue` to some reasonable bounds so the control ties in more tightly to actual application features such as volume, speed, and other measurable amounts. You do not have to do so, but there are few use cases where you want to allow user input for unbounded variables. You can make the stepper "wrap" by setting its `wraps` property to `YES`. When the value exceeds the maximum or falls below the minimum, the `value` wraps around from min to max or max to min, depending on the button pressed.

By default, the stepper autorepeats. That is, it continues to change as long as the user holds one of its buttons. You can disable this by setting the `autorepeat` property to `NO`. The amount the value changes at each tap is controlled by the `stepValue` property. You don't want to ever set this to 0 or a negative number; otherwise, you'll raise a runtime exception.

Recipe: Building a Star Slider

Rating sliders allow users to grade items such as movies, software, and so forth by dragging their fingers across a set of images. It's a common task for touch-based interfaces but one that's not well served by a simple `UISlider` instance, with its floating-point values. Instead, a picker like the one built in Recipe 9-6 limits a user's choice to a discrete set of elements, producing a bounded integer value between zero and the maximum number of items shown. As a user's finger touches each star, the control's value updates and a corresponding event is spawned, allowing your application to treat the star slider like any other `UIControl` subclass.

The art is arbitrary. The example shown in Figure 9-9 uses stars, but there's no reason to limit yourself to stars. Use any art you like, so long as you provide both "on" and "off" images. You might consider hearts, diamonds, smiles, and so on.



Figure 9-9 Recipe 9-6 creates a custom star slider control that animates each star upon selection. A simple animation block causes the star to zoom out and back as the control's value updates.

From the Library of Eric Berna

In addition to simple sliding, Recipe 9-6 adds animation elements as well. Upon achieving a new value, the rightmost star uses a simple animation block to zoom out and back, providing lively feedback to the user in addition to the highlighted visuals. Because the user's finger lays on top of the stars in real use (rather than in the simulator-based screenshot shown in Figure 9-9), the animation must use exaggerated transforms to provide feedback that extends beyond expected finger sizes. Here, the art is quite small and the zoom goes to 150% of the original size, but you can easily adapt both in your applications to match your needs.

Apart from the minimal layout and feedback elements, Recipe 9-6 follows the same kind of custom `UIControl` subclass approach used by Recipe 9-5, tracking touches through their life cycle and spawning events at opportune times. The minimal code needed to add the star elements and feedback in this recipe demonstrates how simple `UIControl` subclassing really is.

Recipe 9-6 **Building a Discrete Valued Star Slider**

```
@implementation StarSlider
- (id) initWithFrame: (CGRect) aFrame
{
    if (self = [super initWithFrame:aFrame])
    {
        // Lay out five stars, with spacing between, and at the ends
        float minimumWidth = WIDTH * 8.0f;
        float minimumHeight = 34.0f;

        // This control uses a minimum 260x34 sized frame
        self.frame = CGRectMake(0.0f, 0.0f,
            MAX(minimumWidth, aFrame.size.width),
            MAX(minimumHeight, aFrame.size.height));

        // Add stars — initially assuming fixed width
        float offsetCenter = WIDTH;
        for (int i = 1; i <= 5; i++)
        {
            UIImageView *imageView = [[UIImageView alloc]
                initWithFrame:CGRectMake(0.0f, 0.0f, WIDTH, WIDTH)];
            imageView.image = OFF_ART;
            imageView.center = CGPointMake(offsetCenter,
                self.frame.size.height / 2.0f);
            offsetCenter += WIDTH * 1.5f;
            [self addSubview:imageView];
        }
    }

    // Place on a contrasting background
    self.backgroundColor =
        [[UIColor blackColor] colorWithAlphaComponent:0.25f];
```

From the Library of Eric Berna

```
    return self;
}

// Handle the value update for the touch point
- (void) updateValueAtPoint: (CGPoint) p
{
    int newValue = 0;
    UIImageView *changedView = nil;

    // Iterate through stars to check against touch point
    for (UIImageView *eachItem in [self subviews])
        if (p.x < eachItem.frame.origin.x)
            eachItem.image = OFF_ART;
        else
        {
            changedView = eachItem; // last item touched
            eachItem.image = ON_ART;
            newValue++;
        }

    // Handle value change
    if (self.value != newValue)
    {
        self.value = newValue;
        [self sendActionsForControlEvents:
            UIControlEventValueChanged];

        // Animate the new value with a zoomed pulse
        [UIView animateWithDuration:0.15f
            animations:^{changedView.transform =
                CGAffineTransformMakeScale(1.5f, 1.5f);}
            completion:^(BOOL done){[UIView
                animateWithDuration:0.1f
                animations:^{changedView.transform =
                    CGAffineTransformIdentity;}];}];
    }
}

- (BOOL)beginTrackingWithTouch:(UITouch *)touch
    withEvent:(UIEvent *)event
{
    // Establish touch down event
    CGPoint touchPoint = [touch locationInView:self];
    [self sendActionsForControlEvents:UIControlEventTouchDown];

    // Calculate value
    [self updateValueAtPoint:touchPoint];
```

From the Library of Eric Berna

```
    return YES;
}

- (BOOL)continueTrackingWithTouch:(UITouch *)touch
    withEvent:(UIEvent *)event
{
    // Test if drag is currently inside or outside
    CGPoint touchPoint = [touch locationInView:self];
    if (CGRectContainsPoint(self.frame, touchPoint))
        [self sendActionsForControlEvents:
            UIControlEventTouchDragInside];
    else
        [self sendActionsForControlEvents:
            UIControlEventTouchDragOutside];

    // Calculate value
    [self updateValueAtPoint:[touch locationInView:self]];
    return YES;
}

- (void) endTrackingWithTouch: (UITouch *)touch
    withEvent: (UIEvent *)event
{
    // Test if touch ended inside or outside
    CGPoint touchPoint = [touch locationInView:self];
    if (CGRectContainsPoint(self.bounds, touchPoint))
        [self sendActionsForControlEvents:
            UIControlEventTouchUpInside];
    else
        [self sendActionsForControlEvents:
            UIControlEventTouchUpOutside];
}

- (void)cancelTrackingWithEvent: (UIEvent *) event
{
    // Cancelled touch
    [self sendActionsForControlEvents:UIControlEventTouchCancel];
}
@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 9 and open the project for this recipe.

From the Library of Eric Berna

Recipe: Building a Touch Wheel

This next recipe creates a touch wheel, like the ones used on older model iPods. Touch wheels provide infinitely scrollable input. Users can rotate their finger clockwise or counterclockwise, and the object's value increases or decreases accordingly. Each complete turn around the wheel (that is, a traversal of 360 degrees) corresponds to a value change of 1.0. Clockwise changes are positive; counterclockwise changes are negative. The value accumulates on each touch, although it can be reset; simply assign the control's `value` property back to 0.0. This property is not a standard part of `UIControl` instances, even though many controls use values.

This recipe computes user changes by casting out vectors from the control's center. The code adds differences in the angle as the finger moves, updating the current value accordingly. For example, three spins around the touch wheel add or subtract 3 to or from the current value, depending on the direction of movement.

This basic wheel defined in Recipe 9-7 tracks touch rotation but does little else. The original iPod scroll wheel offered five click points: in the center circle and at the four cardinal points of the wheel. Adding click support and the associated button-like event support (for `UIControlEventTouchUpInside`) are left as an exercise for the reader.

Recipe 9-7 **Building a Touch Wheel Control**

```
@implementation ScrollWheel

// Layout the wheel
- (id) initWithFrame: (CGRect) aFrame
{
    if (self = [super initWithFrame:aFrame])
    {
        // This control uses a fixed 200x200 sized frame
        self.frame = CGRectMake(0.0f, 0.0f, 200.0f, 200.0f);
        self.center = CGPointMake(CGRectGetMidX(aFrame),
            CGRectGetMidY(aFrame));

        // Add the touchwheel art
        UIImageView *iv = [[UIImageView alloc]
            initWithImage:[UIImage imageNamed:@"wheel.png"]];
        [self  addSubview:iv];
    }
    return self;
}

- (BOOL)beginTrackingWithTouch:(UITouch *)touch
    withEvent:(UIEvent *)event
{
    CGPoint p = [touch locationInView:self];
```

From the Library of Eric Berna

```
    // Center point of view in own coordinate system
    CGPoint cp = CGPointMake(self.bounds.size.width / 2.0f,
        self.bounds.size.height / 2.0f);

    // First touch must touch the gray part of the wheel
    if (!pointInsideRadius(p, cp.x, cp)) return NO;
    if (pointInsideRadius(p, 30.0f, cp)) return NO;

    // Set the initial angle
    self.theta = getangle([touch locationInView:self], cp);

    // Establish touch down
    [self sendActionsForControlEvents:UIControlEventTouchDown];

    return YES;
}

- (BOOL)continueTrackingWithTouch:(UITouch *)touch
    withEvent:(UIEvent *)event
{

    CGPoint p = [touch locationInView:self];

    // Center point of view in own coordinate system
    CGPoint cp = CGPointMake(self.bounds.size.width / 2.0f,
        self.bounds.size.height / 2.0f);

    // Touch updates
    if (CGRectContainsPoint(self.frame, p))
        [self sendActionsForControlEvents:
            UIControlEventTouchDragInside];
    else
        [self sendActionsForControlEvents:
            UIControlEventTouchDragOutside];

    // Falls outside too far, with boundary of 50 pixels?
    if (!pointInsideRadius(p, cp.x + 50.0f, cp)) return NO;

    float newtheta = getangle([touch locationInView:self], cp);
    float dtheta = newtheta - self.theta;

    // correct for edge conditions
    int ntimes = 0;
    while ((ABS(dtheta) > 300.0f)  && (ntimes++ < 4))
        if (dtheta > 0.0f) dtheta -= 360.0f; else dtheta += 360.0f;
```

From the Library of Eric Berna

```
    // Update current values
    self.value -= dtheta / 360.0f;
    self.theta = newtheta;

    // Send value changed alert
    [self sendActionsForControlEvents:UIControlEventValueChanged];

    return YES;
}

- (void) endTrackingWithTouch: (UITouch *)touch
    withEvent: (UIEvent *)event
{
    // Test if touch ended inside or outside
    CGPoint touchPoint = [touch locationInView:self];
    if (CGRectContainsPoint(self.bounds, touchPoint))
        [self sendActionsForControlEvents:
            UIControlEventTouchUpInside];
    else
        [self sendActionsForControlEvents:
            UIControlEventTouchUpOutside];
}

- (void)cancelTrackingWithEvent: (UIEvent *) event
{
    // Cancel
    [self sendActionsForControlEvents:UIControlEventTouchCancel];
}
@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 9 and open the project for this recipe.

Adding a Page Indicator Control

The `UIPageControl` class provides a line of dots that indicates which item of a multipage view is currently displayed. The dots at the bottom of the SpringBoard home page present an example of this kind of control in action. Sadly, the `UIPageControl` class is a disappointment in action. Its instances are awkward to handle, hard to tap, and will generally annoy your users. So when using it, make sure you add alternative navigation options so that the page control acts more as an indicator and less as a control.

From the Library of Eric Berna

Figure 9-10 shows a page control with three pages. Taps to the left or right of the bright-colored current page indicator trigger `UIControlEventValueChanged` events, launching whatever method you set as the control's action. You can query the control for its new value by calling `currentPage` and set the available page count by adjusting the `numberOfPages` property. SpringBoard limits the number of dots representing pages to nine, but your application can use a higher number, particularly in landscape mode.



Figure 9-10 The `UIPageControl` class offers an interactive indicator for multipage presentations. Taps to the left or right of the active dot enable users to select new pages—at least in theory. The page control is hard to tap, requires excessive user precision, and offers poor response performance.

Listing 9-1 uses a `UIScrollView` instance to display three pages of images. Users can scroll through the pictures using swipes, and the page indicator updates to reflect the current page shown. Similarly, users can tap on the page control and the scroller animates the selected page into place. This two-way relationship is built by adding a target-action callback to the page control and a delegate callback to the scroller. Each callback updates the other object, providing a tight coupling between the two.

From the Library of Eric Berna

Listing 9-1 **Using the `UIPageControl` Indicator**

```
@implementation TestBedViewController
- (void) pageTurn: (UIPageControl *) aPageControl
{
    // Animate to the new page
    float width = self.view.frame.size.width;
    int whichPage = aPageControl.currentPage;
    [UIView animateWithDuration:0.3f
        animations:^{sv.contentOffset =
            CGPointMake(width * whichPage, 0.0f);}];
}

- (void) scrollViewDidScroll: (UIScrollView *) aScrollView
{
    // Update the page control to match the current scroll
    CGPoint offset = aScrollView.contentOffset;
    float width = self.view.frame.size.width;
    pageControl.currentPage = offset.x / width;
}

- (void) loadView
{
    [super loadView];

    float width = self.view.frame.size.width;

    // Create the scroll view and set its content size and delegate
    sv = [[UIScrollView alloc] initWithFrame:
        CGRectMake(0.0f, 0.0f, width, width)];
    sv.contentSize = CGSizeMake(NPAGES * width, sv.frame.size.height);
    sv.pagingEnabled = YES;
    sv.delegate = self;

    // Load in all the pages
    for (int i = 0; i < NPAGES; i++)
    {
        NSString *filename =
            [NSString stringWithFormat:@"image%d.png", i+1];
        UIImageView *iv = [[UIImageView alloc] initWithImage:
             [UIImage imageNamed:filename]];
        iv.frame = CGRectMake(i * width, 0.0f, width, width);
        [sv addSubview:iv];
    }

    // Place the scroll view onscreen
    [self.view addSubview:sv];
```

From the Library of Eric Berna

```
    // Update the page control attributes and add a target
    pageControl.numberOfPages = 3;
    pageControl.currentPage = 0;
    [pageControl addTarget:self action:@selector(pageTurn:)
        forControlEvents:UIControlEventValueChanged];
}
@end
```

Recipe: Creating a Customizable Paged Scroller

Listing 9-1 introduced a basic paged scroller but didn't add any dynamic interaction to the equation. That example started and ended with three pages. In real life, page controls are far more useful when you can add and delete pages on the fly. Recipe 9-8 does exactly that. It adds buttons that build and remove views for the `UIScrollView`.

This approach uses not two but four separate controls to produce the add-and-remove interface of Figure 9-11. The four buttons include an add button built using the standard Contacts Add button style, a delete button that mimics that style, a confirm button that looks like an "X," which is built to fit over the delete button, and a full-screen, completely clear cancel button.

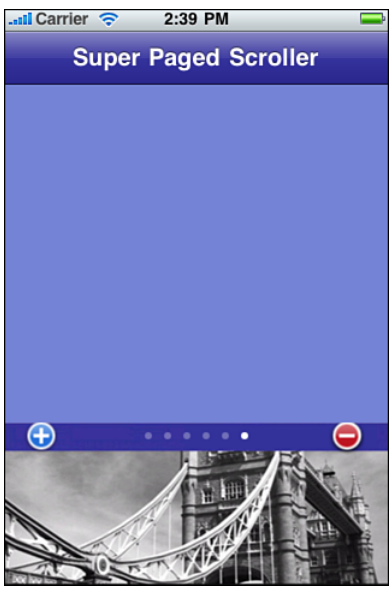


Figure 9-11 The + and – buttons let users add and remove paged views from the scroller. Deletion requires an extra step as a confirm button animates into place.

From the Library of Eric Berna

The buttons work like this: So long as there are fewer than eight pages, the user can tap Add to create a new view in the `UIScrollView`. When this button is tapped, the number of pages for the page control updates and the new view scrolls into place. There's also a check for the current page count; when that page count hits the maximum, the code disables the add button. The eight-page limit is arbitrary. You can adjust the code for a larger or smaller number.

Upon the Delete button being tapped, a confirm button animates into place and the invisible cancel button is enabled, covering the rest of the screen. If the user taps Confirm, the page deletes. A tap anywhere else causes the action to cancel, hiding the Confirm button without performing a page deletion.

This confirm/cancel approach mirrors Apple's delete-with-caution policy that's seen in table edits and in other user interfaces. It takes two taps to delete a page, and the user can cancel out without penalty. This prevents accidental page deletion and provides a safe exit route should the user decide not to continue.

Recipe 9-8 **Adding and Deleting Pages on the Fly**

```
@implementation TestBedViewController

// Page turn via the page control
- (void) pageTurn: (UIPageControl *) aPageControl
{
    int whichPage = aPageControl.currentPage;
    [UIView animateWithDuration:0.3f animations:^{
        scrollView.contentOffset =
            CGPointMake(dimension * whichPage, 0.0f);}];
}

// Page update via scrolling. Add math flexibility
- (void)scrollViewDidEndDecelerating:(UIScrollView *)aScrollView
{
    pageControl.currentPage = floor((scrollView.contentOffset.x /
        dimension) + 0.25);
}

// Return a new color
- (UIColor *)randomColor
{
    float red = (64 + (random() % 191)) / 256.0f;
    float green = (64 + (random() % 191)) / 256.0f;
    float blue = (64 + (random() % 191)) / 256.0f;
    return [UIColor colorWithRed:red green:green blue:blue alpha:1.0f];
}

// Layout pages on addition, deletion, and new orientation
- (void) layoutPages
{
```

From the Library of Eric Berna

```
    int whichPage = pageControl.currentPage;

    // Update the scroll view and its content size
    scrollView.frame = CGRectMake(0.0f, 0.0f, dimension, dimension);
    scrollView.contentSize =
        CGSizeMake(pageControl.numberOfPages * dimension, dimension);
    scrollView.center = CGPointMake(
        CGRectGetMidX(self.view.bounds),
        CGRectGetMidY(self.view.bounds));

    // Layout only pages (tagged with 999)
    float offset = 0.0f;
    for (UIView *eachView in scrollView.subviews)
    {
        if (eachView.tag == 999)
        {
            eachView.frame = CGRectMake(offset, 0.0f,
                dimension, dimension);
            offset += dimension;
        }
    }

    // Scroll to the new page location
    scrollView.contentOffset = CGPointMake(dimension * whichPage, 0.0f);
}

// Add a new page to the layout
- (void) addPage
{
    pageControl.numberOfPages = pageControl.numberOfPages + 1;
    pageControl.currentPage = pageControl.numberOfPages - 1;

    UIView *aView = [[UIView alloc] init];
    aView.backgroundColor = [self randomColor];
    aView.tag = 999;
    [scrollView addSubview:aView];

    [self layoutPages];
}

// User request for a new page
- (void) requestAdd: (UIButton *) button
{
    [self addPage];
    addButton.enabled = (pageControl.numberOfPages < 8) ? YES : NO;
    deleteButton.enabled = YES;
    [self pageTurn:pageControl];
}
```

From the Library of Eric Berna

```
// Remove the current page
- (void) deletePage
{
    int whichPage = pageControl.currentPage;
    pageControl.numberOfPages = pageControl.numberOfPages - 1;
    int i = 0;
    for (UIView *eachView in scrollView.subviews)
    {
        if ((i == whichPage) && (eachView.tag == 999))
        {
            [eachView removeFromSuperview];
            break;
        }

        if (eachView.tag == 999) i++;
    }

    [self layoutPages];
}

// Cancel out of delete
- (void) hideConfirmAndCancel
{
    cancelButton.enabled = NO;
    [UIView animateWithDuration:0.3f animations:^(void){
        confirmButton.center =
            CGPointMake(deleteButton.center.x - 300.0f,
                deleteButton.center.y);}];
}

// User confirms deletion of current page
- (void) confirmDelete: (UIButton *) button
{
    [self deletePage];
    addButton.enabled = YES;
    deleteButton.enabled = (pageControl.numberOfPages > 1) ? YES : NO;
    [self pageTurn:pageControl];
    [self hideConfirmAndCancel];
}

// User canceled delete
- (void) cancelDelete: (UIButton *) button
{
    [self hideConfirmAndCancel];
}

// User requests deletion of current page
- (void) requestDelete: (UIButton *) button
```

From the Library of Eric Berna

```
{
    // Bring forth the cancel and confirm buttons
    [cancelButton.superview bringSubviewToFront:cancelButton];
    [confirmButton.superview bringSubviewToFront:confirmButton];
    cancelButton.enabled = YES;

    // Animate the confirm button into place
    confirmButton.center =
        CGPointMake(deleteButton.center.x - 300.0f,
            deleteButton.center.y);

    [UIView animateWithDuration:0.3f animations:^(void){
        confirmButton.center = deleteButton.center;
    }];
}

// On load, setup the page control and scroll view
- (void) viewDidLoad
{
    // Update the page control
    pageControl.numberOfPages = 0;
    [pageControl addTarget:self action:@selector(pageTurn:)
        forControlEvents:UIControlEventValueChanged];

    // Create the scroll view and set its content size and delegate
    scrollView = [[UIScrollView alloc] init];
    scrollView.pagingEnabled = YES;
    scrollView.delegate = self;
    [self.view addSubview:scrollView];

    // Load in pages
    for (int i = 0; i < INITPAGES; i++)
        [self addPage];
    pageControl.currentPage = 0;

    // Increase the size of the add button for more touchability
    addButton.frame = CGRectInset(addButton.frame, -20.0f, -20.0f);
}

// Update the view layout
- (void) viewDidAppear:(BOOL)animated
{
    dimension = MIN(self.view.bounds.size.width,
        self.view.bounds.size.height) * 0.8f;
    [self layoutPages];
}
@end
```

From the Library of Eric Berna

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 9 and open the project for this recipe.

Building a Toolbar

You can build toolbars in Interface Builder inside Xcode, but when push comes to shove, it's often a lot easier coding items directly. That's because the IB user interface for adding and customizing a toolbar's bar button items is pretty dreadful. You need to keep switching between palettes and inspectors, and things quickly get messy.

After dragging a toolbar into an IB view, you must add and then customize each bar button item. Drag in one bar button item for each element you plan to add. Toolbar elements include view items such as buttons as well as the spacers that lie between those buttons, as shown in Figure 9-12 (left).

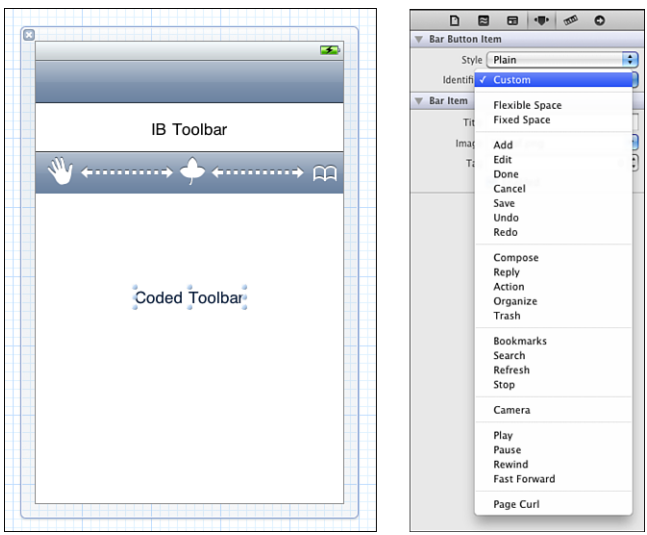


Figure 9-12 Adding bar button items in Interface Builder can be a tedious process.

Once added, the bar button item attributes inspector shown in Figure 9-12 (right) lets you choose which kind of item each bar button represents. Use the Custom style to create custom text- and image-based items. Otherwise, pick from the list of system-defined icons. These include icons for playing media, accessing the camera, editing a list, and more.

From the Library of Eric Berna

When using a system item, make sure your application uses that item in a manner that complies with Apple's Human Interface Guidelines. App Store reviewers take a dim view of "creative" icon interpretations.

On a similar note, avoid creating your own buttons that look like any Apple products or trademarks. Apps have been rejected for using icons that look like iOS and Apple's logo.

Building Toolbars in Code

It's easy to define and lay out toolbars in code provided that you've supplied yourself with a few handy macro definitions. The following macros return proper bar button items for the four available styles of items, and can easily be adapted if you need more control options in your code:

```
#define BARBUTTON(TITLE, SELECTOR) [[UIBarButtonItem alloc] \
    initWithTitle:TITLE style:UIBarButtonItemStylePlain\
    target:self action:SELECTOR]
#define IMGBARBUTTON(IMAGE, SELECTOR) [[UIBarButtonItem alloc] \
    initWithImage:IMAGE style:UIBarButtonItemStylePlain \
    target:self action:SELECTOR]
#define SYSBARBUTTON(ITEM, SELECTOR) [[UIBarButtonItem alloc] \
    initWithBarButtonSystemItem:ITEM \
    target:self action:SELECTOR]
#define CUSTOMBARBUTTON(VIEW) [[UIBarButtonItem alloc] \
    initWithCustomView:VIEW]
```

Those styles are text items, image items, system items, and custom view items. Each of these macros provides an autoreleased `UIBarButtonItem` that can be placed into a `UIToolbar`. Listing 9-2 demonstrates these macros in action, showing how to add each style, including spacers. You can even add a custom view to your toolbars, as Listing 9-2 does. It inserts a `UISwitch` instance as one of the bar button items, as shown in Figure 9-13.

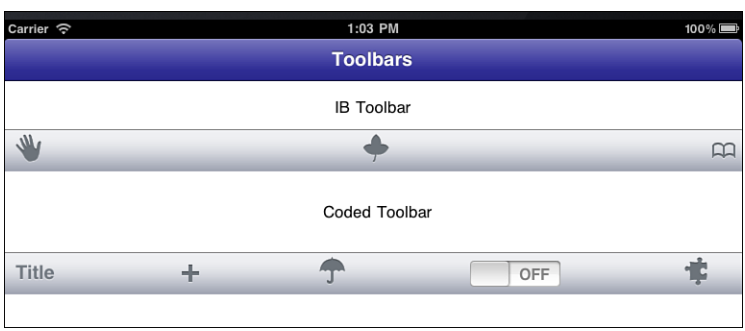


Figure 9-13 Custom toolbar items can include views such as this switch.

From the Library of Eric Berna

The fixed space bar button item represents the only instance where you need to move beyond these handy macros.You must set the item's `width` property to define how much space the item occupies.

Listing 9-2 **Creating Toolbars in Xcode**

```
@implementation TestBedViewController
- (void) action
{
    // no action actually happens
}

- (void) viewDidLoad
{
    UIToolbar *tb = [[UIToolbar alloc] initWithFrame:
        CGRectMake(0.0f, 0.0f, 320.0f, 44.0f)];
    tb.center = CGPointMake(160.0f, 200.0f);
    NSMutableArray *tbitems = [NSMutableArray array];

    // Set up the items for the toolbar
    [tbitems addObject:BARBUTTON(@"Title", @selector(action))];
    [tbitems addObject:SYSBARBUTTON(UIBarButtonSystemItemAdd,
        @selector(action))];
    [tbitems addObject:IMGBARBUTTON([UIImage
        imageNamed:@"TBUmbrella.png"], @selector(action))];
    [tbitems addObject:CUSTOMBARBUTTON([[[UISwitch alloc] init]
        autorelease])];
    [tbitems addObject:SYSBARBUTTON(UIBarButtonSystemItemFlexibleSpace,
        nil)];
    [tbitems addObject:IMGBARBUTTON([UIImage
        imageNamed:@"TBPuzzle.png"], @selector(action))];

    // Add fixed 20 pixel width
    UIBarButtonItem *bbi = [[[UIBarButtonItem alloc]
        initWithBarButtonSystemItem:UIBarButtonSystemItemFixedSpace
        target:nil action:nil] autorelease];
    bbi.width = 20.0f;
    [tbitems addObject:bbi];

    tb.items = tbitems;
    [self.view addSubview:tb];
    [tb release];
}
@end
```

From the Library of Eric Berna

iOS 5 Toolbar Tips

When you're working with toolbars, here are a few tricks of the trade that might come in handy:

- **Fixed spaces can have widths.** Of all `UIBarButtonItems`, only `UIBarButtonSystemItemFixedSpace` items can be assigned a width. So create the spacer item, set its width, and only then add it to your items array.
- **Use a single flexible space for left or right alignment.** Adding a single `UIBarButtonSystemItemFlexibleSpace` at the start of an items list right-aligns all the remaining items. Adding one at the end left-aligns. Use two, one at the start and one at the end, to create center alignments.
- **Take missing items into account.** When hiding a bar button item due to context, don't just use flexible spacing to get rid of the item. Instead, replace the item with a fixed-width space that matches the item's original size. That preserves the layout and leaves all the other icons in the same position both before and after the item disappears.

Summary

This chapter introduced many ways to interact with and get the most from the controls in your applications. Before you move on to the next chapter, here are a few thoughts for you to ponder:

- Just because an item belongs to the `UIControl` class doesn't mean you can't treat it like a `UIView`. Give it subviews, resize it, animate it, move it around the screen, or tag it for later.
- Core Graphics and Quartz 2D let you build visual elements as needed. Combine the comfort of the SDK classes with a little real-time wow to add punch to your presentation.
- If iOS SDK hasn't delivered the control you need, consider adapting an existing control or building a new control from scratch.
- Apple provides top-notch examples of user interface excellence. Consider mimicking their examples when creating new interaction styles such as the confirm button used in this chapter to safeguard a delete action.
- Interface Builder doesn't always provide the best solution for creating interfaces. With toolbars, you may save time in Xcode rather than customizing each element by hand in IB.

From the Library of Eric Berna

This page intentionally left blank

From the Library of Eric Berna

10

Working with Text

You might dismiss the utility of text on a family of touch-based devices. After all, there's so much information that a user can convey using simple gestures. But text plays an important role. Users have many reasons they need to enter and read characters onscreen. Text allows users to sign into accounts, view and reply to e-mail, specify URLs and read the web pages they refer to, and more. Apple's brilliant predictive keyboard transforms text entry into a simple and fairly reliable process; its classes and frameworks offer powerful ways to present and manipulate text from your applications. This chapter introduces text recipes that support a wide range of solutions. You'll read about controlling keyboards, making onscreen elements "text aware," scanning text, laying out text, and so forth. This chapter provides handy recipes for common problems that you'll encounter while working with text.

Recipe: Dismissing a `UITextField` Keyboard

The most commonly asked question about the `UITextField` control is, "How do I dismiss the keyboard after the user is done typing?" There's no built-in way to automatically detect that a user has finished typing and then respond. Yet, when users finish editing the contents of a `UITextField`, the keyboard really should go away. The iPad offers a keyboard-dismissal button but not the iPhone or iPod touch.

Fortunately, it takes little work to respond to the end of text field edits, regardless of platform. You do so by allowing users to tap Done and then resigning first responder status. Resigning first responder moves the keyboard out of sight, as Recipe 10-1 shows. Here are a few key points about implementing this approach:

- **Set the Return key type to `UIReturnKeyDone`, replacing the word "Return" with the word "Done."** You can do this in Interface Builder's attributes inspector or by assignment to the text field's `returnKeyType` property. Using a "Done"-style Return key tells the user *how* to finish editing, rather than just relying on the fact that users have used a similar approach on nonmobile systems. Figure 10-1 shows a keyboard using the Done key style.

From the Library of Eric Berna

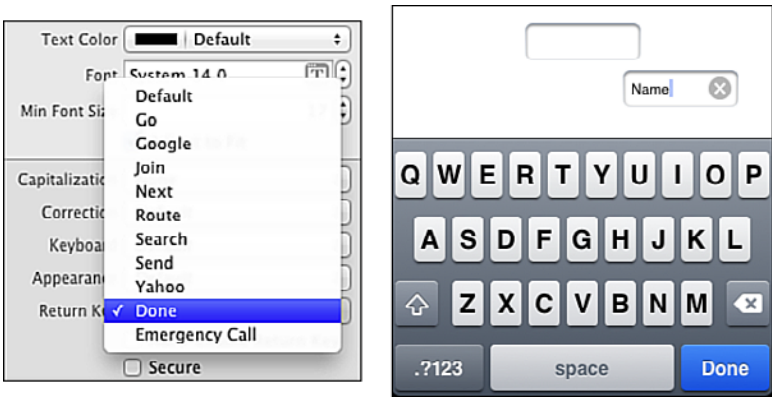


Figure 10-1 Setting the name of the Return key to "Done" (left) tells your user how to finish editing the field. Specify this directly in code or use Interface Builder's text field attributes inspector to customize the way the text field looks and acts.

- **Be the delegate.** You set the text field's `delegate` property to your view controller either in code or in Interface Builder by right-clicking the text field and making the assignment there. Make sure your view controller implements the `UITextFieldDelegate` protocol.
- **Implement `textFieldShouldReturn:`.** This method catches all Return key presses, no matter how they are named. Use the method to resign first responder. This hides the keyboard until the user touches another text field or text view.

> **Note**
>
> You can also use `textFieldShouldReturn:` to perform an action when the Return key is pressed as well as when dismissing the keyboard.

Your code needs to handle each of these points to create a smooth interaction process for your `UITextField` instances.

Text Trait Properties

Text fields implement the `UITextInputTraits` protocol. This protocol provides eight properties that you can set to define the way the field handles text input. Those traits are as follows:

- **`autocapitalizationType`**—Defines the text autocapitalization style. Available styles use sentence capitalization (`UITextAutocapitalizationTypeSentences`), word

From the Library of Eric Berna

capitalization (`UITextAutocapitalizationTypeWords`), all caps (`UITextAutocapitalizationTypeAllCharacters`), and no capitalization (`UITextAutocapitalizationTypeNone`). Avoid capitalizing when working with account name entry. Use word capitalization for proper names and street address entry.

- **`autocorrectionType`**—Specifies whether the text is subject to iOS's autocorrect feature. When this property is enabled (set to `UITextAutocorrectionTypeYes`), iOS suggests replacement words to the user.
- **`spellCheckingType`**—Determines whether to enable spell checking as the user types. Enable it with `UITextSpellCheckingTypeYes`, or disable it with `UITextSpellCheckingTypeNo`.
- **`enablesReturnKeyAutomatically`**—Helps control whether the Return key is disabled when there's no text in an entry field or view. If you set this property to `YES`, the Return key becomes enabled after the user types in at least one character.
- **`keyboardAppearance`**—Provides two keyboard presentation styles: the default style and a style meant to be used with an alert panel.
- **`keyboardType`**—Lets you choose the keyboard that first appears when a user interacts with a field or text view. The available keyboard types are `UIKeyboardTypeDefault`, `UIKeyboardTypeASCIICapable`, `UIKeyboardTypeNumbersAndPunctuation`, `UIKeyboardTypeURL`, `UIKeyboardTypeTwitter`, `UIKeyboardTypeNumberPad`, `UIKeyboardTypePhonePad`, `UIKeyboardTypeDecimalPad`, `UIKeyboardTypeNamePhonePad`, and `UIKeyboardTypeEmailAddress`. Each keyboard has its advantages and disadvantages in terms of the mix of characters it presents. The Email keyboard, for example, is meant to help enter addresses and includes the @ symbol, along with text. The newly added Twitter keyboard offers easy access to the hash tag (#) symbol as well as the user ID @ symbol.
- **`returnKeyType`**—Specifies the text shown on the keyboard's Return key. You can choose from the default ("Return"), Go, Google, Join, Next, Route, Search, Send, Yahoo, Done, and Emergency Call.
- **`secureTextEntry`**—Toggles a text-hiding feature meant to provide more secure text entry. When this property is enabled, you can see the last character typed, but all other characters are shown as a series of dots. Switch on this feature for password text fields, as is demonstrated in a number of recipes scattered throughout this cookbook outside of this chapter.

Other Text Field Properties

In addition to the standard text traits, text fields offer several other properties that control how the field is presented. The `placeholder` text is shown in light gray when the text field is empty, providing a user prompt. Use the placeholder to provide usage hints such as "User Name" or "E-mail address," as shown in Figure 10-2.

From the Library of Eric Berna



Figure 10-2 Placeholder text appears inside text fields in a light gray color when the field is empty. Any text added to the field obscures the placeholder. You can set the placeholder text using the attributes inspector in Interface Builder or by editing the placeholder property for the field.

Text fields allow you to control the type of `borderStyle` displayed around the text area.You can choose from a simple line, a bezel, and a rounded rectangle presentation. These are best seen in Interface Builder, where the attributes inspector lets you toggle between each style.

The text field clear button appears as an X at the right side of the entry area. Set the `clearButtonMode` to specify if and when this button appears: always, never, when editing, or unless editing is ongoing. I use "always" because I feel this is a feature that gives the greatest control to the user.

Recipe 10-1 **Using the Done Key to Dismiss a Text Field Keyboard**

```
@implementation TestBedViewController

- (BOOL)textFieldShouldReturn:(UITextField *)textField
{
    // Catch returns to resign first responder for the text field
    [textField resignFirstResponder];
    return YES;
}

- (void) loadView
{
    [super loadView];

    // Create a text field by hand
    tf = [[UITextField alloc] initWithFrame:
        CGRectMake(0.0f, 0.0f, 100.0f, 30.0f)];
    tf.center = CGPointMake(self.view.center.x, 30.0f);
    tf.placeholder = @"Name";
    [self.view addSubview:tf];

    // Update text fields, including the one defined in IB,
    // to set the delegate, return key type, and other traits
    // To be clear, this demonstrates setting text field properties
    // not how to do view searches - I knew a priori that I was only
    // dealing with a couple of subviews here.
```

From the Library of Eric Berna

```
    for (UIView *view in self.view.subviews)
    {
        if ([view isKindOfClass:[UITextField class]])
        {
            UITextField *aTextField = (UITextField *)view;
            aTextField.delegate = self;
            aTextField.returnKeyType = UIReturnKeyDone;
            aTextField.clearButtonMode =
                UITextFieldViewModeWhileEditing;
            aTextField.borderStyle = UITextBorderStyleRoundedRect;
            aTextField.autocorrectionType = UITextAutocorrectionTypeNo;
        }
    }

}

@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 10 and open the project for this recipe.

Recipe: Adjusting Views Around Keyboards

By necessity, iOS keyboards are rather large. They occupy a good portion of the screen whenever they are in use. Because of that you'll want to adjust your text fields and text views so the keyboard does not block them when it appears onscreen. Figure 10-3 demonstrates this problem. The leftmost image shows the source text view, before it becomes first responder. The middle image demonstrates what users *expect* to happen—namely that the entire view remains accessible by touch even when the keyboard is onscreen. The right image demonstrates what happens when you do *not* resize views. In this case, roughly one third of a screen of text view material becomes inaccessible. Users cannot see the final line of text, let alone edit it in any meaningful manner. The keyboard prevents any touches from getting through to the last paragraph or so of text.

Mitigate the keyboard's presence by allowing views to resize around it. When the keyboard appears, views that continue to require interaction should adjust themselves out of the way, so they don't overlap. To accomplish this, your application will need to subscribe to keyboard notifications.

From the Library of Eric Berna

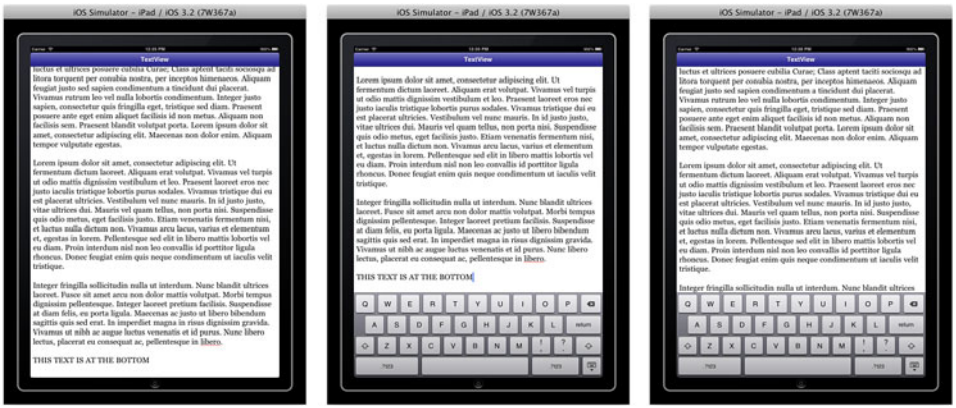


Figure 10-3 Keyboards occupy a large portion of the iOS device screen. If you do not force views to resize themselves when a keyboard appears, it will obscure onscreen material that should remain visible.

iOS offers several notifications that are transmitted using the standard `NSNotificationCenter`. They are `UIKeyboardWillShowNotification`, `UIKeyboardDidShowNotification`, `UIKeyboardWillChangeFrameNotification`, `UIKeyboardWillHideNotification`, and `UIKeyboardDidHideNotification`. Listen for these by adding your class as an observer. Don't forget to unsubscribe to these notifications somewhere in your implementation, especially as or before the object gets destroyed.

```
[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(keyboardWillHide:)
    name:UIKeyboardWillHideNotification object:nil];
```

The two notifications you'll usually want to listen for are "did show" and "will hide," which offer opportune times for you to react to the keyboard arriving onscreen or preparing to leave. Each notification provides a `userInfo` dictionary that supplies the bounds for the keyboard, using the `UIKeyboardBoundsUserInfoKey` key. Sadly, there are no `objects` passed with the notification. You are not granted direct access to the keyboard itself.

Retrieving the keyboard `bounds` lets you resize views to adapt them to the keyboard's presence. Recipe 10-2 shrinks its text view's height by the size of the keyboard upon its appearance. Although standard keyboards always appear at the bottom of the screen, generally when a text-capable view becomes first responder, custom keyboards are not limited to that style of presentation. For that reason, use `UIKeyboardFrameEndUserInfoKey` instead of the keyboard bounds when working with any nonstandard keyboard behavior.

See Recipe 10-4, later in this chapter, for a discussion of the ways you can handle a hardware Bluetooth keyboard, which requires some adaptation to the approach introduced in this recipe.

From the Library of Eric Berna

Recipe 10-2 **Resizing a Text View to Make Way for a Keyboard**

```
@implementation TestBedViewController
CGRect CGRectShrinkHeight(CGRect rect, CGFloat amount)
{
    Return CGRectMake(rect.origin.x, rect.origin.y,
        rect.size.width, rect.size.height - amount);
}

- (void) keyboardWillHide: (NSNotification *) notification
{
    // Return to previous text view size
    tv.frame = self.view.bounds;
}

- (void) keyboardDidShow: (NSNotification *) notification
{
    // Retrieve the keyboard bounds via the userInfo dictionary
    CGRect kbounds;
    NSDictionary *userInfo = [notification userInfo];
    [(NSValue *)[userInfo objectForKey:
        @"UIKeyboardBoundsUserInfoKey"] getValue:&kbounds];

    // Shrink the textview frame
    tv.frame = CGRectShrinkHeight(self.view.bounds, kbounds.size.height);
}

- (void) loadView
{
    [super loadView];

    tv = [[UITextView alloc] initWithFrame:self.view.bounds];
    [self.view addSubview:tv];

    // Subscribe to the two critical notifications
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(keyboardWillHide:)
        name:UIKeyboardWillHideNotification object:nil];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(keyboardDidShow:)
        name:UIKeyboardDidShowNotification object:nil];
}
@end
```

From the Library of Eric Berna

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 10 and open the project for this recipe.

Recipe: Dismissing Text Views with Custom Accessory Views

Custom accessory views allow you to present material whenever the keyboard is shown onscreen. Common uses include adding custom buttons and other controls such as font and color pickers that affect text as the user types. Recipe 10-3 adds two buttons: one that clears already-typed text and another that dismisses the keyboard. The keyboard with these add-ons is shown in Figure 10-4.

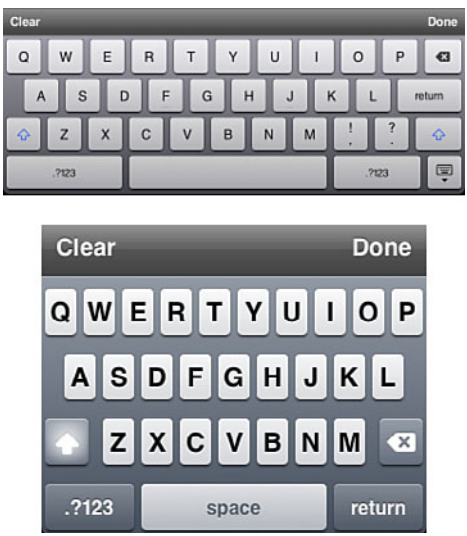


Figure 10-4 Accessory input views allow you to add custom view elements to standard iOS keyboard presentations. Here, a pair of buttons augment iPad and iPhone keyboards.

Each accessory view is associated with the keyboard for a given text-handling view, such as a text field or text view. Add accessories by setting the `inputAccessoryView` property for the view. Recipe 10-3 uses a simple toolbar as its accessory view, providing extra functionality with minimal coding.

Adding a Done button to the toolbar to dismiss the keyboard provides the same kind of user control for text views as Recipe 10-1 offered for text fields. The difference is that this approach allows text views to continue using the Return key to add carriage returns

From the Library of Eric Berna

to text for paragraph breaks. A Done button resigns first-responder status in its callback, invoked when the user finishes his or her edits and taps it. This button is not strictly required for iPad users whose keyboard automatically includes a dismiss button, but at the same time does no harm.

If you wish to filter out the Done button when a universal application is run on the iPad, check the current user interface idiom. The following macro provides a simple way to test for an iPad:

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

Always be aware that Apple may introduce new iOS device form factors, with more space or less space available to users, so try to code accordingly. There's really not much you can do on that account with the current two idioms (iPhone and iPad), but it's worth adding easy-to-find notes into code in places that could potentially see changes in the future.

Recipe 10-3 **Adding Custom Buttons to Keyboards**

```
@interface TestBedViewController : UIViewController
{
    UITextView *tv;
    UIToolbar *tb;
}
@end

@implementation TestBedViewController
- (UIToolbar *) accessoryView
{
    // Return an accessory toolbar with two buttons: Clear and Done
    tb = [[UIToolbar alloc] initWithFrame: CGRectMake(
        0.0f, 0.0f, self.view.frame.size.width, 44.0f)];
    tb.tintColor = [UIColor darkGrayColor];

    NSMutableArray *items = [NSMutableArray array];
    [items addObject:BARBUTTON(@"Clear", @selector(clearText))];
    [items addObject:SYSBARBUTTON(
        UIBarButtonSystemItemFlexibleSpace, nil)];
    [items addObject:BARBUTTON(@"Done", @selector(leaveKeyboardMode))];
    tb.items = items;

    return tb;
}

// Clear via accessory button
- (void) clearText
{
    [tv setText:@""];
}
```

From the Library of Eric Berna

```
// Dismiss keyboard via accessory button
- (void) leaveKeyboardMode
{
    [tv resignFirstResponder];
}

- (void) viewDidAppear: (BOOL) animated
{
    // Assign a custom accessory view to the text view
    tv = [[UITextView alloc] initWithFrame:self.view.bounds];
    tv.inputAccessoryView = [self accessoryView];
    [self.view addSubview:tv];
}
@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 10 and open the project for this recipe.

Recipe: Resizing Views with Hardware Keyboards

Input accessory views are always shown onscreen whenever their text-handling view is first responder. This happens regardless of whether the user is taking advantage of hardware keyboards or not. Figure 10-5 shows Recipe 10-3 when used with and without a hardware keyboard.

When it came to pre-iOS 5's support of hardware keyboards and notification updates, the rule of thumb was this: Sometimes it worked, and sometimes it did not. Handling basic keyboard use, such as that shown in Recipe 10-2, functioned properly regardless of whether you used the onscreen keyboard, a hardware keyboard, or a mix of both. That's because ejecting the hardware keyboard properly sent a keyboard-dismissal notice.

The trouble arose with custom accessory views, like in Recipe 10-3. In such a case, the keyboard resized, but it did not propagate a notification like it would if the keyboard was entirely dismissed or appeared onscreen. The accessory view remained onscreen and no notification occurred. This caused no end of trouble for those who implemented view resizing in their applications.

Fortunately, iOS 5 introduced a way to listen for the transfer between onscreen and hardware keyboards. The `UIKeyboardDidChangeFrameNotification` is fired whenever the onscreen keyboard changes its geometry and allows you to bypass any iOS 4 subclassing workarounds. This new notification is used in Recipe 10-4. When the keyboard frame changes, the recipe updates its text view bounds, matching them to the top of the accessory view. This approach is still not as clean as you might wish (the code checks the toolbar's superview), but it's a big improvement on the way things used to be.

From the Library of Eric Berna

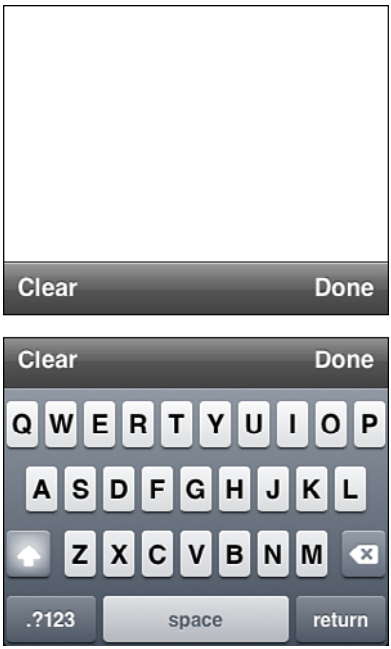


Figure 10-5 When a hardware keyboard is detected, the keyboard is dismissed but not the accessory view, which remains onscreen at all times that its owner view holds first responder.

In addition to frame resizing, Recipe 10-4 reloads the toolbar components in the `keyboardDidShow:` callback. That's because this code shows or hides the Done button on the iPad, depending on the keyboard view. A full iPad keyboard offers a dismiss button by default. When that dismiss button is available, the redundant Done button hides. In hardware mode, the visible button allows users to resign first responder with a tap, avoiding having to physically press keys.

Recipe 10-4 **Handling Hardware Keyboard Changes by Resizable Text Views**

```
@implementation TestBedViewController

// Decide whether to load the Done key into the accessory view
- (void) loadAccessoryView: (BOOL) addDoneKey
{
    NSMutableArray *items = [NSMutableArray array];
    [items addObject:BARBUTTON(@"Clear", @selector(clearText))];
    [items addObject:SYSBARBUTTON(
        UIBarButtonSystemItemFlexibleSpace, nil)];
    if (addDoneKey) [items addObject:BARBUTTON(@"Done",
```

From the Library of Eric Berna

```
            @selector(leaveKeyboardMode))];
    tb.items = items;
}

// Returns a plain accessory view
- (UIToolbar *) accessoryView
{
    tb = [[ObservableToolbar alloc] initWithFrame:
        CGRectMake(0.0f, 0.0f, self.view.frame.size.width, 44.0f)];
    tb.tintColor = [UIColor darkGrayColor];
    return tb;
}

// Provide functionality for the two accessory buttons
- (void) clearText {[tv setText:@""];}
- (void) leaveKeyboardMode {[tv resignFirstResponder];}

- (void) keyboardWillHide: (NSNotification *) notification
{
    // Return to previous text view size
    tv.frame = self.view.bounds;
    tb = nil;
}

- (void) keyboardDidShow: (NSNotification *) notification
{
    // Retrieve the keyboard bounds
    CGRect kbounds;
    NSDictionary *userInfo = [notification userInfo];
    [(NSValue *)[userInfo objectForKey:
        @"UIKeyboardBoundsUserInfoKey"] getValue:&kbounds];

    // Decide whether to show the Done button
    CGFloat startPoint = tb.superview.frame.origin.y;
    CGFloat endHeight = startPoint + kbounds.size.height;
    CGFloat viewHeight = self.view.window.frame.size.height;
    BOOL usingHardwareKeyboard = endHeight > viewHeight;
    [self loadAccessoryView:(!IS_IPAD || usingHardwareKeyboard)];
}

- (void) updateTextViewBounds: (NSNotification *) notification
{
    if (![tv isFirstResponder]) // no keyboard
    {
        tv.frame = self.view.bounds;
        return;
    }
```

From the Library of Eric Berna

```
    // Use the toolbar as the reference point
    CGRect newframe = self.view.bounds;
    newframe.size.height -= (self.view.frame.size.height -
        (tb.superview.frame.origin.y - 44.0f));
    tv.frame = newframe;
}

- (void) viewDidAppear: (BOOL) animated
{
    // Setup the text view
    tv = [[UITextView alloc] initWithFrame:self.view.bounds];
    tv.inputAccessoryView = [self accessoryView];
    [self.view addSubview:tv];

    // Subscribe to all three standard notifications
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(keyboardWillHide:)
        name:UIKeyboardWillHideNotification object:nil];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(keyboardDidShow:)
        name:UIKeyboardDidShowNotification object:nil];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(updateTextViewBounds:)
        name:UIKeyboardDidChangeFrameNotification object:nil];
}
@end
```

Get This Recipe's Code

To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from the book, go to the folder for Chapter 10 and open the project for this recipe.

Recipe: Creating a Custom Input View

Custom input views allow you to replace the keyboard with a view of your design whenever a text view or text field becomes first responder. You can add custom input views to non-text views as well as to text ones. This is covered later in this chapter in Recipe 10-7. For now, Recipe 10-5 focuses on the text scenario.

Whenever you set the `inputView` property, whatever view is assigned to that property is scrolled in and displayed in place of the system keyboard. The easiest way to demonstrate this feature is to create a colored view and assign it to the `inputView` property. Consider the following code snippet that creates two text fields. The second fields' `inputView` property is set to a basic `UIView` instance with a purple background. You can see the results of this code in Figure 10-6. When the first text field becomes first responder, the keyboard appears; when the second field is selected, the purple view appears instead.

From the Library of Eric Berna

```
// Create two standard text fields
UITextField *tf1 = [[UITextField alloc]
    initWithFrame:CGRectMake(0.0f, 0.0f, 200.0f, 30.0f)];
tf1.center = CGPointMake(self.view.frame.size.width / 2.0f, 30.0f);
tf1.borderStyle = UITextBorderStyleRoundedRect;
[self.view addSubview:tf1];

UITextField *tf2 = [[UITextField alloc]
    initWithFrame:CGRectMake(0.0f, 0.0f, 200.0f, 30.0f)];
tf2.center = CGPointMake(self.view.frame.size.width / 2.0f, 80.0f);
tf2.borderStyle = UITextBorderStyleRoundedRect;
[self.view addSubview:tf2];

// Create a purple view to be used as the input view
UIView *purpleView = [[UIView alloc] initWithFrame:CGRectMake(0.0f, 0.0f,
self.view.frame.size.width, 120.0f)];
purpleView.backgroundColor = COOKBOOK_PURPLE_COLOR;

// Assign the input view
tf2.inputView = purpleView;
```

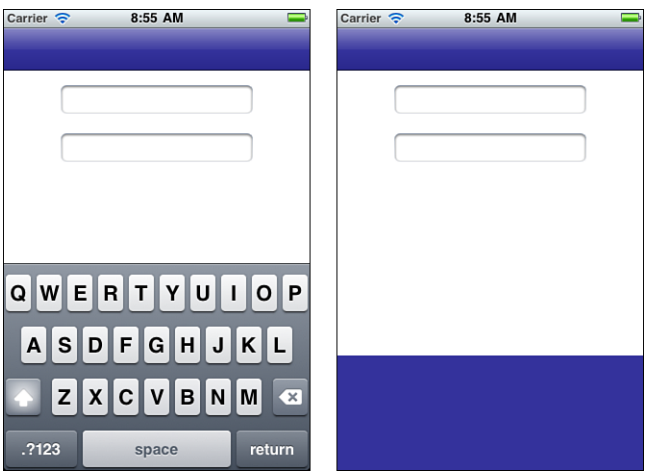


Figure 10-6 Otherwise identical, these two text fields produce different results when selected to become first responder. The top field (left image) presents a standard keyboard. The colored view assigned to the bottom field's `inputView` property (right image) causes the custom view to appear instead.

From the Library of Eric Berna

Because the purple view has no interactive elements, there's not much you can do with it other than to reselect the top text field to get rid of it. You cannot enter text; you cannot dismiss the "keyboard." The only thing you can do is marvel at the functionality of displaying a custom view.

For the most part, custom input views are *not* used for text input in real-life coding. Although input views play an important role in other design patterns, especially gaming, their utility for text is fairly limited. That's because the `inputAccessoryView` property lets you expand the keyboard without sacrificing the built-in keys. Further, the range of keyboard options now includes numeric and decimal entry (long since added in iOS 4.1), which was the prevailing reason for using custom keyboards for many developers.

So, where do custom input views make sense when working with text? For anyone willing to spend the time and effort developing their own keyboards, and taking into account the various platforms and orientations, not to mention Shift modifier keys, input views provide complete control over the user experience. You can create a fully customized skinnable input that replaces the system keyboard with a look and feel that is uniquely suited to your design. It requires a huge amount of work, at many levels.

Recipe 10-5 provides a bare-bones example of a custom text input view. Instead of character entry, it offers two buttons: One types "Hello," and the other types "World" (see Figure 10-7). When tapped, each button inserts the word into its attached text view.



Figure 10-7 The custom keyboard attached as this text view's input view allows users to enter "Hello" and "World," and that's pretty much all.

From the Library of Eric Berna

The challenge in creating a custom text input view like this one lies in how the text changes are propagated back to the first responder. iOS offers no direct link or property that tells a custom input view who its owner is, nor can you use simple superview properties. Because of this challenge, you may want to implement a simple class extension to `UIView` to recover the current first responder.

> **Note**
>
> In this code snippet, the class method, `currentResponder`, is named as it is to marginally avoid conflict with private APIs. `firstResponder` is an actual unpublished method. When you're adding category methods to Apple's classes in production code (rather than sample code, which this is), a good rule of thumb is to prefix all method names with your initials, your company's initials, or some other unique identifier, thus ensuring your method names do not overlap with Apple's or (importantly!) with any methods Apple might add in the future.

```
@interface UIView (FirstResponderUtility)
+ (UIView *) currentResponder;
@end

@implementation UIView (FirstResponderUtility)
- (UIView *) findFirstResponder
{
    if ([self isFirstResponder]) return self;

    for (UIView *view in self.subviews)
    {
        UIView *responder = [view findFirstResponder];
        if (responder) return responder;
    }

    return nil;
}

+ (UIView *) currentResponder
{
    return [[[UIApplication sharedApplication] keyWindow]
        findFirstResponder];
}
@end
```

Recipe 10-5 builds a custom `UIToolbar` intended for use as a custom input view. It displays its two options (Hello and World) as bar buttons. When tapped, the toolbar attempts to insert the strings into the first responder's text. It does so by retrieving the first responder, if it has not yet been set. Then it checks that the responder is a kind of `UITextView`. Only then does it append the new text to the view's existing text. This code could be easily modified to update `UITextField` instances as well as `UITextView` ones.

From the Library of Eric Berna

When an input view is shown, its owner always is first responder, and belongs to the key window. You can cautiously leverage these facts in code, although you'll probably want to expand the minimal error condition checking shown in Recipe 10-5, particularly with regard to the reuse of the `responderView` instance variable.

Recipe 10-5 **Creating a Custom Input View**

```
@interface InputToolbar : UIToolbar
{
    UIView *responderView;
}
@end

@implementation InputToolbar
- (void) appendString: (NSString *) string
{
    // Check for the responder view
    if (!responderView || ![responderView isFirstResponder])
    {
        responderView = [UIView requestFirstResponder];
        if (!responderView) return;
    }

    // Insert the text if the responder is a text view
    if ([responderView isKindOfClass:[UITextView class]])
    {
        UITextView *textView = (UITextView *) responderView;
        textView.text =
            [textView.text stringByAppendingString:string];
    else
        NSLog(@"Cannot append %@ to unknown class type (%@)",
            string, [responderView class]);
}

// Handle the two text entry items
- (void) hello: (id) sender {[self appendString:@"Hello "];}
- (void) world: (id) sender {[self appendString:@"World "];}

// Initialize the bar buttons on the toolbar
- (id) initWithFrame: (CGRect) aFrame
{
    if (!(self = [super initWithFrame: aFrame])) return self;

    NSMutableArray *theItems = [NSMutableArray array];
    [theItems addObject:SYSBARBUTTON(
        UIBarButtonSystemItemFlexibleSpace, nil)];
    [theItems addObject:BARBUTTON(@"Hello", @selector(hello:))];
    [theItems addObject:SYSBARBUTTON(
```

From the Library of Eric Berna

```
        UIBarButtonSystemItemFlexibleSpace, nil)];
    [theItems addObject:BARBUTTON(@"World", @selector(world:))];
    [theItems addObject:SYSBARBUTTON(
        UIBarButtonSystemItemFlexibleSpace, nil)];
    self.items = theItems;

    return self;
}
@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 10 and open the project for this recipe.

Recipe: Making Text-Input-Aware Views

Many months ago, I spent some time updating the open-source iOS BOCHS emulator build to allow it to work with keyboard entry. The default code allowed touch-based "mouse" interactions but didn't offer any keyboard support. After a little investigation, I discovered and then implemented the `UIKeyInput` protocol. This simple protocol, when added to a little first responder manipulation, allows you to update any view to offer text input.

Recipe 10-6 illustrates how to transform a standard `UIToolbar` into a view that accepts keyboard entry, letting users type text directly into the toolbar, as shown in Figure 10-8. As the user types, the toolbar text updates, even properly handling the delete key.



Figure 10-8 Adding the `UIKeyInput` protocol to a toolbar transforms the view into one that can accept and display keyboard input, including deletions.

From the Library of Eric Berna

This recipe requires several features. First, the toolbar must declare the `UIKeyInput` protocol. This protocol announces that the view implements simple text entry and can display the system keyboard (or a custom keyboard, if so desired) when it becomes first responder.

Second, the toolbar must retain state—namely, the current string it is storing. Saving the string as a retained mutable property allows the toolbar to know what text it is currently working with and to display that text to the user.

Next, the toolbar must be able to become first responder. It does so in two ways: by implementing `canBecomeFirstResponder` (returning `YES`) and by catching touches to detect when it should assume that role. Adding a touch handler allows Recipe 10-6 to become first responder when a user touches the view.

Finally, it must implement the three required `UIKeyInput` protocol methods, namely `hasText`, `insertText:`, and `deleteBackwards`. These methods do exactly what their names imply. The `hasText` method returns `YES` whenever the view has any text available. The other two methods insert text at the current insertion point (always at the end for this recipe) and delete a character at a time from the end of the displayed text.

By declaring the protocol, becoming first responder, and handling both the string state and the input callbacks, Recipe 10-6 provides a robust way to add basic text entry to standard `UIView` elements. You can extend these same text features to many other classes, including labels, navigation bars, buttons, and so forth, to use in your applications as needed.

Recipe 10-6 **Adding Keyboard Input to Non-Text Views**

```
@interface KeyInputToolbar: UIToolbar <UIKeyInput>
{
    NSMutableString *string;
}
@end

@implementation KeyInputToolbar
// Is there text available that can be deleted
- (BOOL) hasText
{
    return (string && string.length);
}

// Allow view to become first responder
- (BOOL)canBecomeFirstResponder
{
    return YES;
}

// Become first responder on touch
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;
```

From the Library of Eric Berna

```
{
    [self becomeFirstResponder];
}

// Become first responder on bar button tap
- (void) resume
{
    [self becomeFirstResponder];
}

// Reload the toolbar with the string
- (void) update
{
    NSMutableArray *theItems = [NSMutableArray array];
    [theItems addObject:SYSBARBUTTON(
        UIBarButtonSystemItemFlexibleSpace, nil)];
    [theItems addObject:BARBUTTON(string, @selector(resume))];
    [theItems addObject:SYSBARBUTTON(
        UIBarButtonSystemItemFlexibleSpace, nil)];

    self.items = theItems;
}

// Insert new text into the string
- (void)insertText:(NSString *)text
{
    if (!string) string = [NSMutableString string];
    [string appendString:text];
    [self update];
}

// Delete one character
- (void)deleteBackward
{
    // Super caution, even if hasText reports YES
    if ([self hasText])
    {
        self.string = [NSMutableString string];
        return;
    }

    // Remove a character
    [string deleteCharactersInRange:NSMakeRange(string.length - 1, 1)];
    [self update];
}
@end
```

From the Library of Eric Berna

> **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 10 and open the project for this recipe.

Recipe: Adding Custom Input Views to Non-Text Views

Although custom input views can be applied to text views and text fields, they are far more valuable in other use-cases. Input doesn't have to be about text. In fact, by taking the system keyboard out of the equation, custom input views can range into whatever kind of scenario you need. Think of input views as context-sensitive graphical menus that appear only when a particular view class becomes first responder. When you tap a warrior, perhaps a set of weapons scrolls onscreen, including a bow, a mace, and a sword. The user can select the kind of attack the warrior should apply. Or think of a graphics layout program. When a circle, square, or line is tapped, maybe an onscreen palette is revealed that lets users set the stroke width, the stroke color, and the fill. The only limit to the utility of custom input is your imagination.

Recipe 10-7 demonstrates how a custom input view can affect a non-text view. It combines the code from Recipes 10-5 and 10-6, creating both an input-aware view (`ColorView`), which can become first responder with a touch, and an input view (`InputToolbar`) that affects the display of that primary view. In this example, the base view's role is limited to displaying a color. The toolbar controls what color that is.

Because there's no other way to transfer first responder control, the input view also offers a Done button, allowing the user to dismiss the keyboard, thus resigning first responder from the big color view.

Adding Input Clicks

Use the `UIDevice` class to add input clicks to your custom input accessory views. The `playInputClick` method plays the standard system keyboard click and can be called when you respond to user input taps.

Adopt the `UIInputViewAudioFeedback` protocol in the accessory input class and add an `enableInputClicksWhenVisible` delegate method that always returns `YES`. This defers audio playback to the user's preferences, which are set in Settings > Sounds. In order to hear these clicks, the user must have enabled keyboard click feedback. If the user has not done so, your calls to `playInputClick` are simply ignored.

Recipe 10-7 **Creating a Custom Input Controller for a Non-Text View**

```
@interface ColorView : UIView <UIKeyInput>
@property (retain) UIView *inputView;
@end
```

From the Library of Eric Berna

```
#pragma mark Key Input Aware View
@implementation ColorView
@synthesize inputView;

// UITextInput protocol
- (BOOL) hasText {return NO;}
- (void)insertText:(NSString *)text {}
- (void)deleteBackward {}

// First responder support
- (BOOL)canBecomeFirstResponder {return YES;}
- (void)touchesBegan:(NSSet *)touches
    withEvent:(UIEvent *)event {[self becomeFirstResponder];}

// Initialize with user interaction allowed
- (id) initWithFrame:(CGRect)aFrame
{
    if (!(self = [super initWithFrame:aFrame])) return self;
    self.backgroundColor = COOKBOOK_PURPLE_COLOR;
    self.userInteractionEnabled = YES;
    return self;
}
@end

#pragma mark Color Input Toolbar
@interface InputToolbar : UIToolbar <UIInputViewAudioFeedback>
@end

@implementation InputToolbar
- (BOOL) enableInputClicksWhenVisible
{
    return YES;
}

- (void) updateColor: (UIColor *) aColor
{
    [UIView currentResponder].backgroundColor = aColor;
    [[UIDevice currentDevice] playInputClick];
}

// Color updates
- (void) light: (id) sender {
    [self updateColor:[COOKBOOK_PURPLE_COLOR
        colorWithAlphaComponent:0.33f]];}
- (void) medium: (id) sender {
    [self updateColor:[COOKBOOK_PURPLE_COLOR
        colorWithAlphaComponent:0.66f]];}
```

From the Library of Eric Berna

```
- (void) dark: (id) sender {
    [self updateColor:COOKBOOK_PURPLE_COLOR];}

// Resign first responder on pressing Done
- (void) done: (id) sender
{
    [[UIView currentResponder] resignFirstResponder];
}

// Create a toolbar with each option available
- (id) initWithFrame: (CGRect) aFrame
{
    if (!(self = [super initWithFrame: aFrame])) return self;

    NSMutableArray *theItems = [NSMutableArray array];
    [theItems addObject:SYSBARBUTTON(
        UIBarButtonSystemItemFlexibleSpace, nil)];
    [theItems addObject:BARBUTTON(@"Light", @selector(light:))];
    [theItems addObject:SYSBARBUTTON(
        UIBarButtonSystemItemFlexibleSpace, nil)];
    [theItems addObject:BARBUTTON(@"Medium", @selector(medium:))];
    [theItems addObject:SYSBARBUTTON(
        UIBarButtonSystemItemFlexibleSpace, nil)];
    [theItems addObject:BARBUTTON(@"Dark", @selector(dark:))];
    [theItems addObject:SYSBARBUTTON(
        UIBarButtonSystemItemFlexibleSpace, nil)];
    [theItems addObject:BARBUTTON(@"Done", @selector(done:))];
    self.items = theItems;

    return self;
}
@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 10 and open the project for this recipe.

Recipe: Building a Better Text Editor

Undo support and persistence allow you to create better text editors in your application. These features ensure that your users can reverse any mistakes and can pick up their work from where they left off. To accomplish this requires surprisingly little programming, as is demonstrated in Recipe 10-8.

Text views provide built-in support that works hand-in-hand with select, cut, copy, and paste. The undo manager understands these actions, so possible user messages might

From the Library of Eric Berna

include "Undo Paste," "Redo Cut," and so forth. All the view controller needs to do is instantiate an undo manager; it leaves the rest of the work to the built-in objects.

This recipe adds Undo and Redo buttons to the keyboard accessory view. These buttons must be updated each time the text view contents change. To accomplish this, the view controller becomes the text view's delegate and implements the `textViewDidChange:` delegate method. Buttons are enabled or disabled accordingly.

This recipe uses persistence to store the text contents between application launches. It archives its contents to file in the `performArchive` method. The application delegate calls this method right before the application is due to suspend and also each time the text view resigns first responder status. This better ensures that the data remains fresh and up to date between application sessions.

```
- (void) applicationWillResignActive:(UIApplication *)application
{
    [tbvc archiveData];
}
```

On launch, any data in that file is read in to initialize the text view instance during the view controller setup.

Recipe 10-8 **Adding Undo Support and Persistence to Text Views**

```
@implementation TestBedViewController
- (void) archiveData
{
    // Store current text out to file
    [tv.text writeToFile:DATAPATH atomically:YES
        encoding:NSUTF8StringEncoding error:nil];
}

// Decide how to load up the accessory view
- (void) loadAccessoryView
{
    NSMutableArray *items = [NSMutableArray array];
    UIBarButtonItem *spacer =
        SYSBARBUTTON(UIBarButtonSystemItemFixedSpace, nil);
    spacer.width = 40.0f;

    BOOL canUndo = [tv.undoManager canUndo];
    UIBarButtonItem *undoItem =  SYSBARBUTTON_TARGET(
        UIBarButtonSystemItemUndo, tv.undoManager, @selector(undo));
    undoItem.enabled = canUndo;
    [items addObject:undoItem];
    [items addObject:spacer];

    BOOL canRedo = [tv.undoManager canRedo];
    UIBarButtonItem *redoItem = SYSBARBUTTON_TARGET(
```

From the Library of Eric Berna

```
        UIBarButtonSystemItemRedo, tv.undoManager, @selector(redo));
    redoItem.enabled = canRedo;
    [items addObject:redoItem];
    [items addObject:spacer];

    [items addObject:SYSBARBUTTON(
        UIBarButtonSystemItemFlexibleSpace, nil)];
    [items addObject:BARBUTTON(@"Done",
        @selector(leaveKeyboardMode))];

    tb.items = items;
}

// Save whenever resigning keyboard mode
- (void) leaveKeyboardMode
{
    [tv resignFirstResponder];
    [self performArchive];
}

// Update the undo/redo buttons whenever there's a text change
- (void)textViewDidChange:(UITextView *)textView
{
    [self loadAccessoryView];
}

- (void) viewDidAppear: (BOOL) animated
{
    self.view.frame = [[UIScreen mainScreen] applicationFrame];

    // Set up the text view
    tv = [[UITextView alloc] initWithFrame:self.view.bounds];
    tv.inputAccessoryView = [self accessoryView];
    tv.delegate = self;

    // Load the document if it exists
    if ([[NSFileManager defaultManager] fileExistsAtPath:DATAPATH])
        tv.text = [NSString stringWithContentsOfFile:DATAPATH
            encoding:NSUTF8StringEncoding error:nil];

    [self.view addSubview:tv];
}
@end
```

From the Library of Eric Berna

> **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 10 and open the project for this recipe.

Recipe: Text Entry Filtering

At times you want to ensure that a user enters only a certain subset of characters. For example, you might want to create a numeric-only text field that does not handle letters. Although you can use predicates to test the final entry against a regular expression (the `NSPredicate` class's `MATCH` operator supports regex values, and is demonstrated in Recipe 10-10), for filtered data it's easier to check each new character as it's typed against a legal set.

A `UITextField` delegate can catch those characters as they are typed and decide whether to add the character to the active text field. The optional `textField:shouldChangeCharactersInRange:replacementString:` delegate method returns either `YES`, allowing the newly typed character(s), or `NO`, disallowing it (or them). In practice, this works on a character-by-character basis being called after each user keyboard tap. However, with iOS's pasteboard support, the replacement string could theoretically be longer when text is pasted to a text field.

Recipe 10-9 works by looking for any disallowed characters within the new string. When it finds them, it rejects the entry, leaving the text field unedited. So a paste of mixed allowed and disallowed text would be rejected entirely.

This recipe considers four scenarios: alphabetic text entry only, numeric, numeric with an allowed decimal point, and a mix of alphanumeric characters. You can adapt this example to any set of legal characters you want.

The third entry type, numbers with a decimal point, uses a little trick to ensure that only one decimal point gets typed. Once it finds a period character in the associated text field, it switches the characters it accepts from a set with the period to a set without it. Yes, you can sneak your way around this using paste, although it's unlikely that users will resort to doing so.

Recipe 10-9 **Filtering User Text Entry**

```
#define ALPHA @"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz "

@implementation TestBedViewController
- (BOOL)textField:(UITextField *)textField
    shouldChangeCharactersInRange: (NSRange)range
    replacementString:(NSString *)string
{
    NSMutableCharacterSet *cs =
        [NSMutableCharacterSet
            characterSetWithCharactersInString:@""];
```

From the Library of Eric Berna

```
    switch (seg.selectedSegmentIndex)
    {
        case 0:
            [cs addCharactersInString:ALPHA];
            break;
        case 1:
            [cs formUnionWithCharacterSet:
                [NSCharacterSet decimalDigitCharacterSet]];
            break;
        case 2:
            [cs formUnionWithCharacterSet:
                [NSCharacterSet decimalDigitCharacterSet]];

            // permit one decimal only
            if ([textField.text rangeOfString:@"."].location
                == NSNotFound)
                [cs addCharactersInString:@"."];
            break;
        case 3:
            [cs addCharactersInString:ALPHA];
            [cs formUnionWithCharacterSet:
                [NSCharacterSet decimalDigitCharacterSet]];
            break;
        default:
            break;
    }

    NSString *filtered =
        [[string componentsSeparatedByCharactersInSet:[cs invertedSet]]
            componentsJoinedByString:@""];
    BOOL basicTest = [string isEqualToString:filtered];
    return basicTest;
}


- (void) segmentChanged: (UISegmentedControl *) seg
{
    // Reset text on segment change
    tf.text = @"";
}

- (void) viewDidAppear: (BOOL) animated
{
    // Create a testbed text field to work with
    tf = [[UITextField alloc] initWithFrame:
        CGRectMake(0.0f, 0.0f, 200.0f, 30.0f)];
    tf.delegate = self;
```

From the Library of Eric Berna

```
    [self.view addSubview:tf];

    // Add segmented control with entry options
    seg = [[UISegmentedControl alloc] initWithItems:
            [@"ABC 123 2.3 A2C" componentsSeparatedByString:@" "]];
    [seg addTarget:self action:@selector(segmentChanged:)
        forControlEvents:UIControlEventValueChanged];
    self.navigationItem.titleView = seg;
}
@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 10 and open the project for this recipe.

Recipe: Detecting Text Patterns

Recipe 10-9 introduced ways to limit users to entering legal characters. From there, it's just a short hop to matching user input against a variety of legal patterns. Consider a floating-point number. It might be described as an optional sign followed by a whole component followed by an optional decimal and then a fractional component. Or maybe the whole component should be optional but the sign mandatory. Unfortunately, there are many standard ways of describing things, and those ways increase exponentially when you expand from simple numbers to phone numbers, e-mail address, and URLs. Apple has taken care of many of these for you, with its built-in data detector classes, but it often helps to know how to roll your own.

Rolling Your Own Expressions

Some standards organizations have published descriptions of exactly what makes up a legal value, and enterprising developers have transformed many of those descriptions into fairly portable regular expressions. Consider the following regular expression definition of a floating-point number:

```
^[+-]?[0-9]+[\.]?[0-9]*$
```

It's not a perfect definition, but for many purposes it's a pretty good one, and a flexible one to boot. It accepts a pretty good range of floating-point numbers with optional signs. Admittedly, as presented it won't accept -.75, but it will also not accept –. which I think makes it a fair compromise, because -0.75 isn't too hard to guess on the part of the user. Alternatively, you could use a set of regular expression checks, and accept any positive result that occurs out of that set—for example, adding in floating points that do not require a whole portion but do require a decimal point to start them followed by one or more digits:

```
^[+-]?\.[0-9]+$
```

From the Library of Eric Berna

You can use an `NSPredicate` to compare the text from an `NSString` to a regular expression, detecting whether a user has entered a valid floating-point number. Here's an example:

```
NSPredicate *fpPredicate = [NSPredicate predicateWithFormat:
    @"SELF MATCHES '^[+-]?[0-9]+[\\.]?[0-9]*$'"];
BOOL match = [fpPredicate evaluateWithObject:string];
```

It is, as already stated, a bit harder to detect phone numbers, e-mail, and other more sophisticated entry types. Here's my inexpert go at the phone number problem, in terms of a regular expression:

```
^[\(]?([2-9][0-9]{2})[\)]?[-.\. ]?([2-9][0-9]{2})[-.\. ]?([0-9]{4})$
```

This regular expression offers optional parentheses, although there is no way to check that they balance—you could accomplish that with some simple additional Objective-C coding. It ensures that both the area code and the phone number prefix don't start with 0 or 1, and allows the user to enter optional spacers between the numbers (a space, a dash, or a period). In other words, for one line of description, it's a pretty okay but not spectacular definition of phone numbers.

Recipe 10-10 uses this regular expression to determine when it has detected a phone number entered by the user. Upon receiving a positive match, it updates the navigation bar's title to acknowledge success. It demonstrates how you can perform real-time filtering and pattern matching to detect some goal pattern and provide a way to act on positive results.

Enumerating Regular Expressions

The `NSRegularExpression` class offers a simple block-based enumeration approach that allows you to find all matches within a string, applying updates to the given ranges. If you're working with attributed text (see the Core Text discussion later in this chapter), you can apply color or font hints to just those elements that match the regex.

Create a regular expression, and then enumerate it over a string (typically one found in a text view of some sort), and use each range to create some kind of visual update.

```
// Check for matches
NSRegularExpression *regex = [NSRegularExpression
    regularExpressionWithPattern:@"REGEXHERE "
    options:NSRegularExpressionCaseInsensitive error:nil];

// Enumerate over a string
[regex enumerateMatchesInString:text options:0 range:fullRange
    usingBlock:^(NSTextCheckingResult *match,
        NSMatchingFlags flags, BOOL *stop){
        NSRange range = match.range;
       // Perform some action on the range
}];
```

From the Library of Eric Berna

Data Detectors

The `NSDataDetector` class is a subclass of `NSRegularExpression`. Data Detectors allow you to search for well-defined data types, including dates, addresses, URL links, phone numbers, and transit information using Apple's fully tested algorithms instead of trying to create your own regular expressions. Take the same approach used previously for enumerating regular expressions. This code snippet searches for links (URLs) and phone numbers.

```
NSError *error = NULL;
NSDataDetector *detector = [NSDataDetector
    dataDetectorWithTypes:NSTextCheckingTypeLink|NSTextCheckingTypePhoneNumber
    error:&error];

// Enumerate over a string
[detector enumerateMatchesInString:text options:0 range:fullRange
    usingBlock:^(NSTextCheckingResult *match,
        NSMatchingFlags flags, BOOL *stop){
NSRange range = match.range;
       // Perform some action on the range
}];
```

The checks are built around the `NSTextCheckingResult` class. This class describes items that match the data detector's content discovery. The kinds of data detectors supported by iOS are going to grow over time. For now, they are limited to `NSTextCheckingTypeDate`, `NSTextCheckingTypeAddress`, `NSTextCheckingTypeLink`, `NSTextCheckingTypePhoneNumber`, and `NSTextCheckingTypeTransitInformation`.

Adding Built-in Type Detectors

`UITextViews` and `UIWebViews` offer built-in data type detectors, including phone numbers, HTTP links, and so forth. Set the `dataDetectorTypes` property to allow the view to automatically convert pattern matches into clickable URLs that are embedded into the view's text. Legal types include addresses, calendar events, links, and phone numbers. Use `UIDataDetectorTypeAll` to match all supported types, or use `UIDataDetectorTypeNone` to disable pattern matching.

Recipe 10-10 **Detecting Text Patterns Using Predicates and Regular Expressions**

```
@implementation TestBedViewController
- (void) updateStatus: (NSString *) string
{
    NSPredicate *telePredicate = [NSPredicate predicateWithFormat:
        @"SELF MATCHES \
        '^[\\(]?([2-9][0-9]{2})[\\)]?[-.\\. ]?([2-9][0-9]{2})\
        [-.\\. ]?([0-9]{4})$'"];
    BOOL match = [telePredicate evaluateWithObject:string];
    self.title = match ? @"Phone Number" : nil;
}
```

From the Library of Eric Berna

```
- (BOOL)textField:(UITextField *)textField
    shouldChangeCharactersInRange:(NSRange)range
    replacementString:(NSString *)string
{
    NSString *newString = [textField.text
        stringByReplacingCharactersInRange:range withString:string];

    if (!string.length)
    {
        [self updateStatus:newString];
        return YES;
    }

    NSMutableCharacterSet *cs = [NSMutableCharacterSet
        characterSetWithCharactersInString:@""];
    [cs formUnionWithCharacterSet:
        [NSCharacterSet decimalDigitCharacterSet]];
    [cs addCharactersInString:@"()-. "];

    // Legal characters check
    NSString *filtered = [[string componentsSeparatedByCharactersInSet:
        [cs invertedSet]] componentsJoinedByString:@""];
    BOOL basicTest = [string isEqualToString:filtered];

    // Test for phone number
    [self updateStatus:basicTest ? newString : textField.text];

    return basicTest;
}

- (void) loadView
{
    [super loadView];

    tf = [[UITextField alloc] initWithFrame:
        CGRectMake(0.0f, 0.0f, 200.0f, 30.0f)];
    tf.center = CGPointMake(self.view.frame.size.width / 2.0f, 40.0f);
    tf.borderStyle = UITextBorderStyleRoundedRect;
    tf.autocorrectionType = UITextAutocorrectionTypeNo;
    tf.clearButtonMode = UITextFieldViewModeAlways;
    tf.delegate = self;
    [self.view addSubview:tf];
}
@end
```

From the Library of Eric Berna

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 10 and open the project for this recipe.

Recipe: Detecting Misspelling in a `UITextView`

The `UITextChecker` class provides a way to automatically scan text for misspellings. To use this class, you must first set the target language—for example, en for English, en_US for U.S. English, fr_CA for Canadian French, and so on. The language codes use a combination of ISO 639-1 and optional ISO 3166-1 regions, so while you can choose to use a general English dictionary (en), you can also differentiate between usage in the U.S. (en_US), Australia (en_AU), and the UK (en_GB). Query `UITextChecker` for an array of `availableLanguages` from which to pick.

The class also allows you to learn new words (`learnWord:`) and forget words (`unlearnWord:`) to customize the onboard dictionary to the user's need. Learned words are used globally across languages, so when you add a person's name, that name is available universally. Checker objects can also set words to ignore using instance methods.

Recipe 10-11 demonstrates how to incorporate a text checker into your application by iteratively selecting each misspelled word. To do this, you need to control range selection for the text view. To select text in a `UITextView`, it must already be first responder. Check the responder status and update the view if needed:

```
if (![tv isFirstResponder])
    [tv becomeFirstResponder];
```

Then calculate a range you wish to select, making sure you take the contents' length into account, and set the `selectedRange` property for the text view:

```
tv.selectedRange = NSMakeRange(offset, length);
```

Because a text view must be editable, as well as the first responder, the keyboard will appear onscreen while you perform any range selection. The user will be able to edit any material you have onscreen, so code for cases in which user edits may disrupt your application.

Recipe 10-11 **Searching for Misspellings**

```
@implementation TestBedViewController
- (void) nextMisspelling: (id) sender
{
    // Scan for a new word from the current offset
    NSRange range = [checker rangeOfMisspelledWordInString: tv.text
        range:NSMakeRange(0, tv.text.length) startingAt:offset
        wrap:YES language:@"en"];

    // Skip forward each time a new misspelling is found
    if (range.location != NSNotFound)
```

From the Library of Eric Berna

```
        offset = range.location + range.length;
    else
        offset = 0;

    // Select the word
    if (![tv isFirstResponder])[tv becomeFirstResponder];
    if (range.location != NSNotFound)
        tv.selectedRange = range;
}

- (void) viewDidAppear: (BOOL) animated
{
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Next Misspelling", @selector(nextMisspelling:));

    tv = [[UITextView alloc] initWithFrame:self.view.bounds];
    [self.view addSubview:tv];
    tv.editable = YES;

    checker = [[UITextChecker alloc] init];
}
@end
```

Searching for Text Strings

It takes little work to adapt Recipe 10-11 to search for text. To implement search, add a text field to your navigation bar and change the bar button to "Find." Use `NSString`'s `rangeOfString:options:range:` method to locate the desired string. Careful, the string you search for must not be `nil`. Once you've found the range of your target text (and assuming the location is not `NSNotFound`), you can then scroll the text view to the right position by calling `scrollRangeToVisible:`. Pass the range returned by the string method.

Note

`NSNotFound` is a constant used to indicate that a range was not successfully located. Always check the `location` field after a search to ensure that a valid value was set.

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 10 and open the project for this recipe.

From the Library of Eric Berna

Recipe: Dumping Fonts

These days, iOS ships with a fairly rugged collection of fonts, which you can easily explore to your heart's content. Recipe 10-12 iterates through `UIFont`'s collection of font families and their members to create an HTML-styled message that is automatically placed into a mail message. You can address the e-mail and send off a copy of that font list to your home computer, allowing the iOS device to report its font contents to you.

Make sure you add the MessageUI framework to your project to allow the `MFMailComposeViewController` references to compile. And don't forget to check whether the device can send e-mail before attempting to build the message.

Recipe 10-12 **Sending a Report of Available Fonts by E-mail**

```
- (void) send: (id) sender
{
    NSMutableString *results = [NSMutableString string];

    for (NSString *family in [UIFont familyNames])
    {
        [results appendFormat:@"<h3>%@ Family</h3><ul>", family];

        for (NSString *font in [UIFont fontNamesForFamilyName:family])
            [results appendFormat:@"<li>%

        [results appendString:@"</ul>"];
    }

    MFMailComposeViewController *mcvc =
        [[MFMailComposeViewController alloc] init];
    [mcvc setSubject:@"Available iOS Fonts"];
    [mcvc setMessageBody:results isHTML:YES];

    mcvc.mailComposeDelegate = self;
    mcvc.modalTransitionStyle = UIModalTransitionStyleFlipHorizontal;
    [self presentModalViewController:mcvc animated:YES];
}
```

Get This Recipe's Code

To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from the book, go to the folder for Chapter 10 and open the project for this recipe.

From the Library of Eric Berna

Recipe: Adding Custom Fonts to Your App

You are never limited to the device's repertoire of onboard fonts now that iOS supports custom font integration into applications. This recipe demonstrates how you can add your own fonts into an application for your use. Figure 10-9 shows a screenshot using the Pirulen font along with some basic Lorem Ipsum text.



Figure 10-9 Adding custom fonts to your applications allows you to stylize your applications with a unique textual finish.

Start by adding the TrueType font to your application's Resources folder. Then edit your application's Info.plist file to declare `UIAppFonts` (a.k.a. "Fonts provided by application"). This is an array of filenames that you add to. Recipe 10-13 uses a single font entry, namely pirulen.ttf. Save your changes to the property list before continuing.

As Recipe 10-13 demonstrates by its brevity, there's little more to do other than to set the font property for a text view or text field to the desired face and size.

Recipe 10-13 **Using `UIFont` to Access a Custom Font**

```
- (void) viewDidAppear: (BOOL) animated
{
    UITextView *tv = [[UITextView alloc] initWithFrame:self.view.bounds];
    [self.view addSubview:tv];

    // This call to UIFont uses a custom font name
```

From the Library of Eric Berna

```
    tv.font = [UIFont fontWithName:@"pirulen"
        size:IS_IPAD ? 28.0f : 12.0f];
    tv.text = lorem;
    tv.editable = NO;
}
```

> **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 10 and open the project for this recipe.

Recipe: Basic Core Text and Attributed Strings

Core Text is a Mac and iOS technology that developers use to programmatically support typesetting tasks such as text description and layout. Its APIs enable you to define fonts, colors, alignment, word wrapping, and other features that move strings beyond being a simple collection of characters.

Core Text accomplishes this challenge by building complex attributed strings. Attributed strings, as the name implies, add characteristics such as font assignments and colors to select substrings. To get a sense of how attributes can combine, consider the following string:

`This **is a *sample*** *string* that demonstrates how attributes can combine.`

In this non-iOS example, constrained by the realities of book publishing, the preceding string uses a bold attribute from its 6th through 16th characters, and italic from the 11th through 23rd characters.

On iOS, the kinds of attributes applied to a string differ from those used when writing a book. With Core Text, you do not add emphasis or bolding. Those are set by changing the font face—for example, from Courier to Courier-Bold or to Courier-BoldOblique. You specify the range where that font face should apply, just as you can specify ranges to set a given stroke color or a text alignment.

Define traits by working with members of the `NSAttributedString` class or its mutable cousin `NSMutableAttributedString`. The mutable version offers far more flexibility, allowing you to layer attributes individually rather than having to add everything all at once.

Even with mutable instances, working with Core Text feels a lot like you've jumped back to Core Foundation rather than moving you forward to Cocoa Touch. Its functions, constants, and general approach are not especially friendly for Objective-C users. To add paragraph traits, for example, you'll have to specify the trait name, a pointer to a variable that stores the value for the trait, and the size of that value, not to mention the number of traits in use.

To address this awkward API, Recipe 10-14 introduces a string helper class that iteratively builds up attributes and contents. This class allows you to approach attributed strings using an Objective-C wrapper. To use this helper class, create a new string helper and then start applying traits. Traits are remembered until they are reset, so if you set the text color

From the Library of Eric Berna

to red, that trait will endure until you reset the `foregroundColor` property or change it to another value.

The following snippet uses Recipe 10-14 to build a large centered red-colored headline followed by a justified paragraph using a much smaller font. After these commands are executed, the `stringHelper` instance stores all the attributes and text into a single `NSMutableAttributedString`.

```
// Initialize a string helper
stringHelper = [StringHelper buildHelper];

stringHelper.fontName = @"Futura";
stringHelper.fontSize = 36.0f;
stringHelper.foregroundColor = [UIColor redColor];
stringHelper.alignment = @"Center";
[stringHelper appendFormat:@"Declaration of Independence\n\n"];

NSString *sourceText = @"When in the Course of human events it becomes necessary
for one people to dissolve the political bands which have connected them with
another and to assume among the powers of the earth, the separate and equal
station to which the Laws of Nature and of Nature's God entitle them, a decent
respect to the opinions of mankind requires that they should declare the causes
which impel them to the separation.";

stringHelper.fontSize = 18.0f;
stringHelper.alignment = @"Justified";
stringHelper.foregroundColor = [UIColor blackColor];
[stringHelper appendFormat:@"%@", sourceText];
```

The next challenge lies in drawing the attributed text to a context such as a view or an image. To support this, Recipe 10-14 implements a custom view class and overrides the `drawRect:` method. The following method demonstrates a typical approach for moving from an attributed string to an onscreen display, such as the one shown in Figure 10-10:

```
- (void) drawRect:(CGRect)rect
{
    [super drawRect: rect];
    CGContextRef context = UIGraphicsGetCurrentContext();

    CGContextSetTextMatrix(context, CGAffineTransformIdentity);
    CGContextTranslateCTM(context, 0, self.bounds.size.height);
    CGContextScaleCTM(context, 1.0, -1.0); // flip the context

    // Slightly inset from the edges of the view
    CGMutablePathRef path = CGPathCreateMutable();
    CGRect insetRect = CGRectInset(self.frame, 100.0f, 80.0f);
    CGPathAddRect(path, NULL, insetRect);
```

From the Library of Eric Berna

```
    // Draw the text
    CTFramesetterRef framesetter =
        CTFramesetterCreateWithAttributedString(
            (__bridge CFAttributedStringRef)self.string);
    CTFrameRef theFrame =
        CTFramesetterCreateFrame(framesetter,
            CFRangeMake(0, self.string.length), path, NULL);
    CTFrameDraw(theFrame, context);

    CFRelease(framesetter);
    CFRelease(path);
    CFRelease(theFrame);
}
```

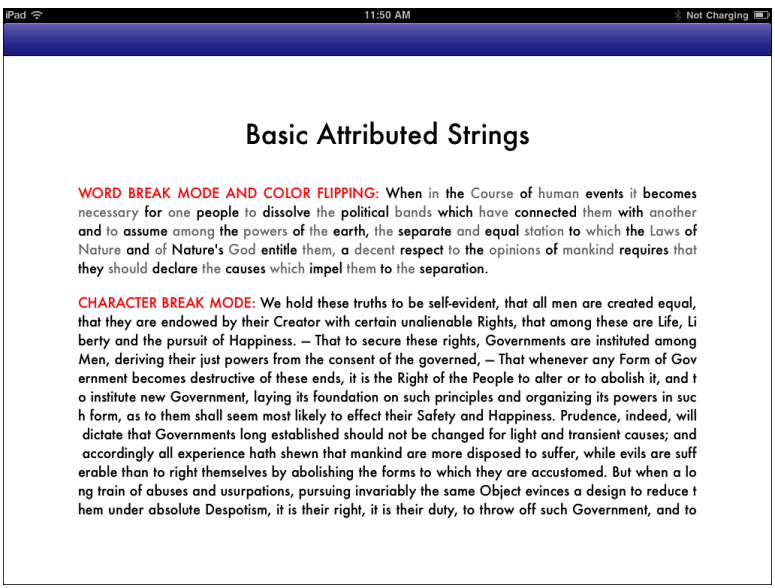


Figure 10-10 This text was typeset entirely using Core Text and attributed strings created by the `StringHelper` wrapper class from Recipe 10-14. It demonstrates both character and paragraph traits, including font size updates, color switches, line break modes, and alignment.

This method starts by flipping its context, similar to several methods in Chapter 7, "Working with Images," allowing the text to start drawing from the top-left corner. It then creates an inset rectangle using the view bounds as a starting point and moving in from there. Finally, it establishes a framesetter object, which is responsible for managing

From the Library of Eric Berna

the elements that break down into individual characters. Those elements are called "lines," each of which contains a number of glyph "runs." Each run contains a series of text glyphs, or characters. Each glyph in a run shares common attributes. The framesetter uses all this information to build a "frame," which can then draw the text.

By changing the path, you can achieve eye-catching visual effects quite cheaply. You'll see some of this in the recipes that follow this one. For this example, when you adjust the size of the inset rectangle, you'll find that the text adapts and re-adjusts its wrapping to match its new boundaries. Other ways to change the shape include adding ellipses and arcs to the path. Under recent iOS releases, Core Text ably handles nonrectangular paths.

Recipe 10-14 **Building Attributed Strings with an Objective-C Wrapper**

```
#define MATCHSTART(STRING1, STRING2) \
    ([[STRING1 uppercaseString] hasPrefix:[STRING2 uppercaseString]])

@implementation StringHelper
@synthesize string;
@synthesize fontName, fontSize;
@synthesize foregroundColor;
@synthesize alignment, breakMode;

// Class convenience method
+ (id) buildHelper
{
    return [[self alloc] init];
}

// Set up defaults
- (id) init
{
    if (!(self = [super init])) return self;

    string = [[NSMutableAttributedString alloc] init];
    fontName = @"Helvetica";
    fontSize = 12.0f;

    return self;
}

// Return a Core Text alignment
- (uint8_t) ctAlignment
{
    if (!alignment) return kCTNaturalTextAlignment;
    if (MATCHSTART(alignment, @"n")) return kCTNaturalTextAlignment;
    if (MATCHSTART(alignment, @"l")) return kCTLeftTextAlignment;
    if (MATCHSTART(alignment, @"c")) return kCTCenterTextAlignment;
```

From the Library of Eric Berna

```
    if (MATCHSTART(alignment, @"r")) return kCTRightTextAlignment;
    if (MATCHSTART(alignment, @"j")) return kCTJustifiedTextAlignment;
    return kCTNaturalTextAlignment;
}

// Return a Core Text break mode
- (uint8_t) ctBreakMode
{
    if (!breakMode) return kCTLineBreakByWordWrapping;
    if (MATCHSTART(breakMode, @"word"))
        return kCTLineBreakByWordWrapping;
    if (MATCHSTART(breakMode, @"char"))
        return kCTLineBreakByCharWrapping;
    if (MATCHSTART(breakMode, @"clip"))
        return kCTLineBreakByClipping;
    if (MATCHSTART(breakMode, @"head"))
        return kCTLineBreakByTruncatingHead;
    if (MATCHSTART(breakMode, @"tail"))
        return kCTLineBreakByTruncatingTail;
    if (MATCHSTART(breakMode, @"mid")) return
        kCTLineBreakByTruncatingMiddle;
    return kCTLineBreakByWordWrapping;
}

// Return a paragraph style that represents both alignment and
// word break settings. The style returned has a +1 retain count
- (CTParagraphStyleRef) newParagraphStyle
{
    int addedTraits = 0;
    if (alignment) addedTraits++;
    if (breakMode) addedTraits++;
    if (!addedTraits) return nil;

    uint8_t theAlignment = [self ctAlignment];
    CTParagraphStyleSetting alignSetting = {
        kCTParagraphStyleSpecifierAlignment,
        sizeof(uint8_t),
        &theAlignment};

    uint8_t theLineBreak = [self ctBreakMode];
    CTParagraphStyleSetting wordBreakSetting = {
        kCTParagraphStyleSpecifierLineBreakMode,
        sizeof(uint8_t),
        &theLineBreak};

    CTParagraphStyleSetting settings[2] = {
        alignSetting, wordBreakSetting};
```

From the Library of Eric Berna

```
    CTParagraphStyleRef paraStyle =
        CTParagraphStyleCreate(settings, 2);

    return paraStyle;
}

// Append text to the attributed string using the current settings
- (void) appendFormat: (NSString *) formatstring, ...
{
    if (!formatstring) return;

    va_list arglist;
    va_start(arglist, formatstring);
    NSString *outstring = [[NSString alloc]
        initWithFormat:formatstring arguments:arglist];
    va_end(arglist);

    CTFontRef basicFontRef = CTFontCreateWithName(
        (__bridge CFStringRef)fontName, fontSize, NULL);
    NSMutableDictionary *basicFontAttr =
        [NSMutableDictionary dictionaryWithObjectsAndKeys:
            (__bridge id) basicFontRef,
                (__bridge NSString *) kCTFontAttributeName,
            nil];
    CFRelease(basicFontRef);

    if (foregroundColor)
        [basicFontAttr
            setObject:(__bridge id) foregroundColor.CGColor
            forKey:(__bridge NSString *)
                kCTForegroundColorAttributeName];

    CTParagraphStyleRef style = [self newParagraphStyle];
    if (style)
    {
        [basicFontAttr
            setObject:(__bridge id)style
            forKey:(__bridge NSString *)
                kCTParagraphStyleAttributeName];
        CFRelease(style);
    }

    NSAttributedString *newString = [[NSAttributedString alloc]
        initWithString:outstring attributes:basicFontAttr];
    [self.string appendAttributedString:newString];
}
@end
```

From the Library of Eric Berna

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 10 and open the project for this recipe.

Using Pseudo-HTML to Create Attributed Text

Individually adding attributes to a string, even with a helper class, soon becomes tedious. That's why using HTML, a familiar standard built for text markup, can transform and simplify your layout tasks. It's easier to produce basic HTML content than to iteratively implement the Core Text calls that create a similar presentation. Consider the following string:

```
<h1>Using Core Text with Markup</h1><p>This is an example
    <i>of some text that uses <b>HTML styling</b>.</p>
```

ISO-compliant HTML layout for this simple string would require quite a bit of Core Text coding to create the final effect specified by the markup. But if you're willing to be flexible, you can automate an HTML-like solution that creates the Core Text coding for you.

The `UIWebView` class intrinsically supports typesetting through HTML, and does so in a rigorously standards-supported way. Core Text views, however, can be produced with much lighter weight code, without the memory overhead and extra features introduced by web views. For example, you might not want to add a dozen web views to your `UITableView` cells, but using Core Text–backed alternatives would work much better, with little memory overhead.

The challenge lies in transforming an HTML annotated string into a Core Text attributed string. Recipe 10-15 tackles this problem, offering an enhanced HTML subset that parses contents into attributed strings.

Note

Of course, roughly 20 minutes after I finished implementing my little recipe, I was introduced to iPhone developer Oliver Drobnik's attributed strings extensions library hosted at GitHub, which I highly recommend that you explore. You can find his repository at https://github.com/Cocoanetics/NSAttributedString-Additions-for-HTML.

Recipe 10-15 brings more to the equation than standard HTML, however. Because of it's relatively simple scanning and matching approach, it's just a matter of minutes to add custom tags of your own. I've placed in a couple of convenience tags that don't rely on standard HTML—namely the `color` and `size` tags found toward the end of the `scanString:` method.

Utilities like these ultimately exist to serve your development needs, not necessarily to support a standard such as HTML. If you can add new tags with little work for your own use, then a custom solution like this recipe may save you a lot of effort in your application

From the Library of Eric Berna

development. Don't feel you must be tied to the standard. Once you've added (and rigorously debugged, of course) a scanner and parser like Recipe 10-15 to your code, it can save you a lot of time in future projects where you reuse your Core Text routines without having to bother with a lot of the Core Text overhead.

Recipe 10-15 **Automatically Parsing Markup Text into Core Text Attributed Strings**

```
// Scan the pseudo-"HTML" and create an attributed string
+ (NSAttributedString *) stringFromMarkup: (NSString *) aString
{
    // Prepare to scan
    NSScanner *scanner = [NSScanner scannerWithString:aString];
    [scanner setCharactersToBeSkipped:[NSCharacterSet
        newlineCharacterSet]];
    NSCharacterSet *startSet = [NSCharacterSet
        characterSetWithCharactersInString:@"<"];
    NSCharacterSet *endSet = [NSCharacterSet
        characterSetWithCharactersInString:@">"];

    // Initialize a string helper
    StringHelper *stringHelper = [StringHelper buildHelper];
    CGFloat fontSize = BASE_TEXT_SIZE;

    // Initialize headers, bolding, italics.
    int hlevel = 0;
    BOOL bold = NO, emph = NO;

    // Scan through the source string
    NSUInteger loc = 0;
    while (loc < aString.length)
    {
        NSString *contentText = nil; // scan to tag
        [scanner scanUpToCharactersFromSet:startSet
            intoString:&contentText];

        // Handle content (non-tag) text here
        if (contentText)
        {
            // Move the next location forward
            scanner.scanLocation = (loc += contentText.length + 1);

            // Set the font for the content material
            NSString *fontName = @"Georgia";
            if (hlevel == 0)
            {
                if (bold && emph) fontName = @"Georgia-BoldItalic";
                else if (bold) fontName = @"Georgia-Bold";
```

From the Library of Eric Berna

```
            else if (emph) fontName = @"Georgia-Italic";
        }

        // Apply current attributes
        stringHelper.fontName = fontName;
        stringHelper.fontSize = fontSize;
        [stringHelper appendFormat:contentText];
    }

    // Scan for a tag
    NSString *baseTag = nil;
    [scanner scanUpToCharactersFromSet:endSet intoString:&baseTag];
    if (!baseTag)
    {
        NSLog(@"Unexpected error encountered while scanning");
        return stringHelper.string;
    }

    // Move the next location forward
    scanner.scanLocation = (loc += baseTag.length + 1);

    // Restore standard tag form
    NSString *tagText = [baseTag stringByAppendingString:@">"];
    if (![tagText hasPrefix:@"<"])
        tagText = [@"<" stringByAppendingString:tagText];

    // -- PROCESS TAGS --

    // Header Tags
    if (STRMATCH(tagText, @"</h")) // finish any headline
    {
        hlevel = 0;
        [stringHelper appendFormat:@"\n"];
        fontSize = BASE_TEXT_SIZE;
    }
    else if (STRMATCH(tagText, @"<h1>")) hlevel = 1;
    else if (STRMATCH(tagText, @"<h2>")) hlevel = 2;
    else if (STRMATCH(tagText, @"<h3>")) hlevel = 3;
    else hlevel = 0;
    if (hlevel)
        fontSize = BASE_TEXT_SIZE + (8.0f - hlevel) * 2.0f;

    // Bold and Italic Tags
    if (STRMATCH(tagText, @"</i>"))      emph = NO;
    else if (STRMATCH(tagText, @"<i>"))  emph = YES;
    else if (STRMATCH(tagText, @"</b>")) bold = NO;
    else if (STRMATCH(tagText, @"<b>"))  bold = YES;
```

From the Library of Eric Berna

```
        // Center Tag
        if (STRMATCH(tagText, @"</center>"))
            stringHelper.alignment = @"natural";
        else if (STRMATCH(tagText, @"<center>"))
            stringHelper.alignment = @"center";

        // Custom (non-HTML) tag examples: Color and Size

        if (STRMATCH(tagText, @"<color red>"))
            stringHelper.foregroundColor = [UIColor redColor];
        if (STRMATCH(tagText, @"<color green>"))
            stringHelper.foregroundColor = [UIColor greenColor];
        if (STRMATCH(tagText, @"<color blue>"))
            stringHelper.foregroundColor = [UIColor blueColor];
        else if (STRMATCH(tagText, @"</color")) // match partial
            stringHelper.foregroundColor = [UIColor blackColor];

        if (STRMATCH(tagText, @"<size")) // match partial
        {
            // Scan the value for the new font size
            NSScanner *newScanner = [NSScanner scannerWithString:tagText];
            NSCharacterSet *cs = [[NSCharacterSet
                decimalDigitCharacterSet] invertedSet];
            [newScanner setCharactersToBeSkipped:cs];
            [newScanner scanFloat:&fontSize];
        }
        else if (STRMATCH(tagText, @"</size>"))
            fontSize = BASE_TEXT_SIZE;


        // Paragraph and line break tags
        if (STRMATCH(tagText, @"<br")) // match all variants
            [stringHelper appendFormat:@"\n"];
        else if (STRMATCH(tagText, @"</p>"))
            [stringHelper appendFormat:@"\n\n"];
        else if (STRMATCH(tagText, @"<p>")) // default alignment
            stringHelper.alignment = @"justified";
    }
    return stringHelper.string;
}
```

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 10 and open the project for this recipe.

From the Library of Eric Berna

Recipe: Splitting Core Text into Pages

As you saw in Recipe 10-15, producing Core Text attributed strings from markup helps you separate presentation from implementation. It's a really flexible approach that lets you edit your source material without affecting your code base. The problem comes when you are dealing with text that ranges beyond a single page. For those cases, you'll want to split the text into sections on a page-by-page basis.

Recipe 10-16 shows how you might do that. It uses a Core Text framesetter to return an array of recommended page breaks based on a given point-based page size. You can use this hand-in-hand with Recipe 10-15's attributed strings and Chapter 5's page view controller to automatically build a book from your marked-up text, as shown in Figure 10-11. The application shown in the figure represents Recipe 10-16's full sample code app. It reads in text, converts it from marked-up source into an attributed string, and then breaks it into pages for display in the paged controller.

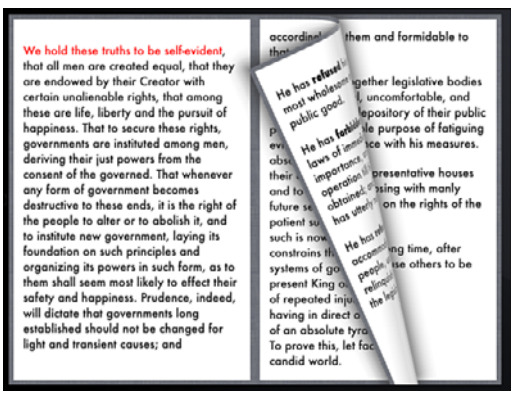


Figure 10-11 Core Text framesetters allow you to break attributed strings into sections that fit a given page size.

It's based on the method you see in Recipe 10-16. This method iterates through an attributed string, collecting range information on a page-by-page basis. The Core Text `CTFramesetterSuggestFrameSizeWithConstraints` method takes a starting range and a destination size, producing a destination range that fits its attributed string. Although this recipe doesn't use any control attributes, you may want to explore these for your own implementations. Apple's technical Q&A QA1698 offers some pointers.

From the Library of Eric Berna

Recipe 10-16 **Multipage Core Text**

```
- (NSArray *) findPageSplitsForString: (NSAttributedString *)theString
    withPageSize: (CGSize) pageSize
{
    NSInteger stringLength = theString.length;
    NSMutableArray *pages = [NSMutableArray array];

    CTFramesetterRef frameSetter =
        CTFramesetterCreateWithAttributedString(
            (__bridge CFAttributedStringRef) theString);

    CFRange baseRange = {0,0};
    CFRange targetRange = {0,0};
    do {
        CTFramesetterSuggestFrameSizeWithConstraints(
            frameSetter, baseRange, NULL, pageSize, &targetRange);
        NSRange destRange = {baseRange.location, targetRange.length};
        [pages addObject:[NSValue valueWithRange:destRange]];
        baseRange.location += targetRange.length;
    } while(baseRange.location < stringLength);

    CFRelease(frameSetter);
    return pages;
}
```

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 10 and open the project for this recipe.

Recipe: Drawing Core Text into PDF

The Core Text recipes you have seen so far all draw their text into views, rendering them in `drawRect` updates. Recipe 10-17 transfers that drawing from views to PDFs. It builds a new PDF file, adding content page-by-page, using the same page layout method introduced in Recipe 10-16.

Drawing to a PDF context proves similar to drawing to an image context, the difference lying in the call to begin a new PDF page. Recipe 10-17 builds its multipage PDF document and stores it to the path passed in the method call.

Recipe 10-17 **Drawing to PDF**

```
- (void) dumpToPDFFile: (NSString *) pdfPath
{
    // Read in data, convert it to an attributed string
    NSString *path = [[NSBundle mainBundle]
```

From the Library of Eric Berna

```
        pathForResource:@"data" ofType:@"txt"];
    NSString *markup = [NSString stringWithContentsOfFile:path
        encoding:NSUTF8StringEncoding error:nil];
    NSAttributedString *attributed =
        [MarkupHelper stringFromMarkup:markup];

    CGRect theBounds = CGRectMake(0.0f, 0.0f, 640.0f, 480.0f);
    CGRect insetRect = CGRectInset(theBounds, 20.0f, 20.0f);

    NSArray *pageSplits =
        [self findPageSplitsForString:attributed
            withPageSize:insetRect.size];
    int offset = 0;

    UIGraphicsBeginPDFContextToFile(pdfPath, theBounds, nil);

    for (NSNumber *pageStart in pageSplits)
    {
        UIGraphicsBeginPDFPage();
        CGContextRef context = UIGraphicsGetCurrentContext();

        CGContextSetTextMatrix(context, CGAffineTransformIdentity);
        CGContextTranslateCTM(context, 0, theBounds.size.height);
        CGContextScaleCTM(context, 1.0, -1.0);

        CGMutablePathRef path = CGPathCreateMutable();
        CGPathAddRect(path, NULL, insetRect);

        NSRange offsetRange = {offset, [pageStart integerValue]};
        NSAttributedString *subString =
            [attributed attributedSubstringFromRange:offsetRange];
        offset += offsetRange.length;

        CTFramesetterRef framesetter =
            CTFramesetterCreateWithAttributedString(
                (__bridge CFAttributedStringRef)subString);
        CTFrameRef theFrame = CTFramesetterCreateFrame(
            framesetter, CFRangeMake(0, subString.length),
            path, NULL);
        CTFrameDraw(theFrame, context);

        CFRelease(framesetter);
        CFRelease(path);
        CFRelease(theFrame);
    }

    UIGraphicsEndPDFContext();
}
```

From the Library of Eric Berna

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 10 and open the project for this recipe.

Recipe: Drawing into Nonrectangular Paths

Recipes 10-16 and 10-17 used rectangles to bound the displayed text, but there's no reason on modern iOS devices not to use more challenging path shapes. Recipe 10-18 introduces code that draws the shaped text shown in Figure 10-12. To achieve this effect requires little more than changing the path from a rectangle to an ellipse. In addition, the recipe adds a little extra Quartz drawing to provide the backdrop and the outer stroke.

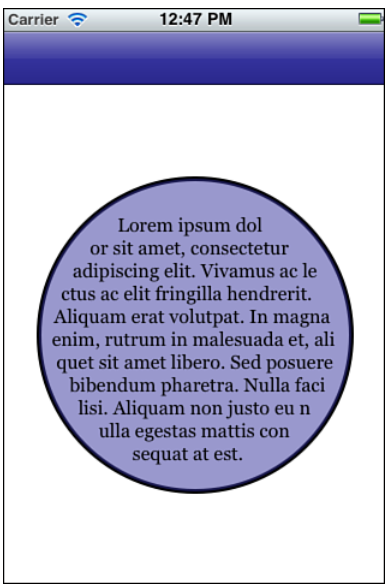


Figure 10-12 Using circles and ellipses (or any other shaped path) can produce eye-catching visual elements in your applications.

It's easy to extend this approach to quite silly extremes, as demonstrated in Figure 10-13. These were a couple of sample images I created for fun while working on Recipe 10-18. They were built by creating complex path elements using mutable paths in Quartz.

From the Library of Eric Berna

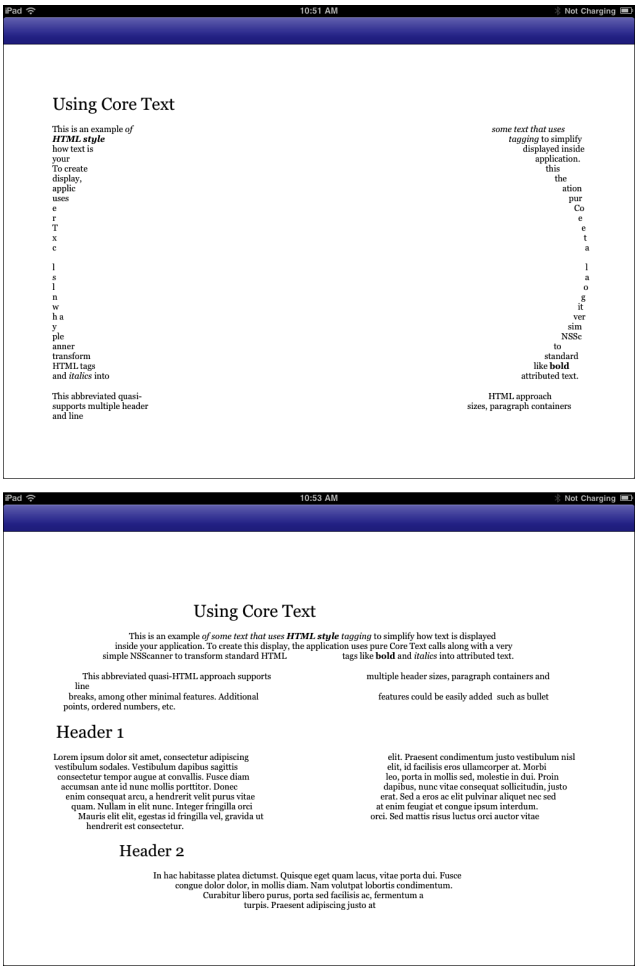


Figure 10-13 Use Core Text path drawing responsibly. Let these examples serve as an inspiration of why designers should be involved early and often in application development.

Recipe 10-18 **Drawing Core Text into Circles**

```
// Draw text into a circle using Core Text and Quartz
- (void) drawRect:(CGRect)rect
{
    [super drawRect: rect];
    CGContextRef context = UIGraphicsGetCurrentContext();
```

From the Library of Eric Berna

```
    // Flip the context
    CGContextSetTextMatrix(context, CGAffineTransformIdentity);
    CGContextTranslateCTM(context, 0, self.bounds.size.height);
    CGContextScaleCTM(context, 1.0, -1.0);

    // Calculate a centered square
    CGFloat minSide = MIN(self.frame.size.width, self.frame.size.height);
    CGRect squareRect = CGRectMake(
        CGRectGetMidX(self.frame) - minSide / 2.0f,
        CGRectGetMidY(self.frame) - minSide / 2.0f, minSide, minSide);
    CGRect insetRect = CGRectInset(squareRect, 30.0f, 30.0f);

    // Create an ellipse path
    CGMutablePathRef backPath = CGPathCreateMutable();
    CGPathAddEllipseInRect(backPath, NULL, insetRect); // circle path

    // Stroke that path
    CGContextAddPath(context, backPath);
    CGContextSetLineWidth(context, 4.0f);
    [[UIColor blackColor] setStroke];
    CGContextStrokePath(context);

    // Fill that path
    CGContextAddPath(context, backPath);
    [[COOKBOOK_PURPLE_COLOR colorWithAlphaComponent:0.5f] setFill];
    CGContextFillPath(context);

    CFRelease(backPath);

    // Inset even further, so the text won't touch the edges
    insetRect = CGRectInset(insetRect, 10.0f, 10.0f);
    CGMutablePathRef insetPath = CGPathCreateMutable();
    CGPathAddEllipseInRect(insetPath, NULL, insetRect);

    CTFramesetterRef framesetter =
        CTFramesetterCreateWithAttributedString(
            (CFAttributedStringRef)string);
    CTFrameRef theFrame = CTFramesetterCreateFrame(framesetter,
        CFRangeMake(0, string.length), insetPath, NULL);
    CTFrameDraw(theFrame, context);

    CFRelease(framesetter);
    CFRelease(theFrame);
    CFRelease(insetPath);
}
```

From the Library of Eric Berna

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 10 and open the project for this recipe.

Recipe: Drawing Text onto Paths

Sadly, Apple has not provided any API support whatsoever to allow you to draw text onto arbitrary paths such as `UIBezierPath` instances. That's a shame, because it's one of the most requested features for Core Text. Fortunately, Recipe 10-19 introduces a way to accomplish this. The method is passed an array of points representing the elements of a path. Recipe 10-19 draws text onto that path, making sure to preserve typographic integrity.

Figure 10-14 presents a gallery of drawing solutions that can be created with Recipe 10-19. Figure 10-14 (top left) shows drawing to a circle. Notice how the letters are laid out properly. Because wider characters take up more space, a sequence such as "wiw" would appear with typographic proportions, avoiding extra spaces on each side of the relatively thin "i" character.

Figure 10-14 (top right) demonstrates a spiral layout. Here, a path was created that moved further and further away from a center point. The following method shows how you might calculate those points:

```
- (UIBezierPath *) spiralPath
{
    float cX = CGRectGetMidX(self.bounds);
    float cY = CGRectGetMidY(self.bounds);

    float radius = IS_IPAD ? 60.0f : 30.0f;
    float dRadius = 1.0f;

    float dTheta = 2 * M_PI / 60.0f;

    UIBezierPath *path = [UIBezierPath bezierPath];
    [path moveToPoint:CGPointMake(cX + radius, cY)];

    for (float theta = dTheta; theta < 8 * M_PI; theta += dTheta)
    {
        radius += dRadius;
        float dx = radius * cos(theta);
        float dy = radius * sin(theta);
        [path addLineToPoint:CGPointMake(cX + dx, cY + dy)];
    }

    return path;
}
```

From the Library of Eric Berna

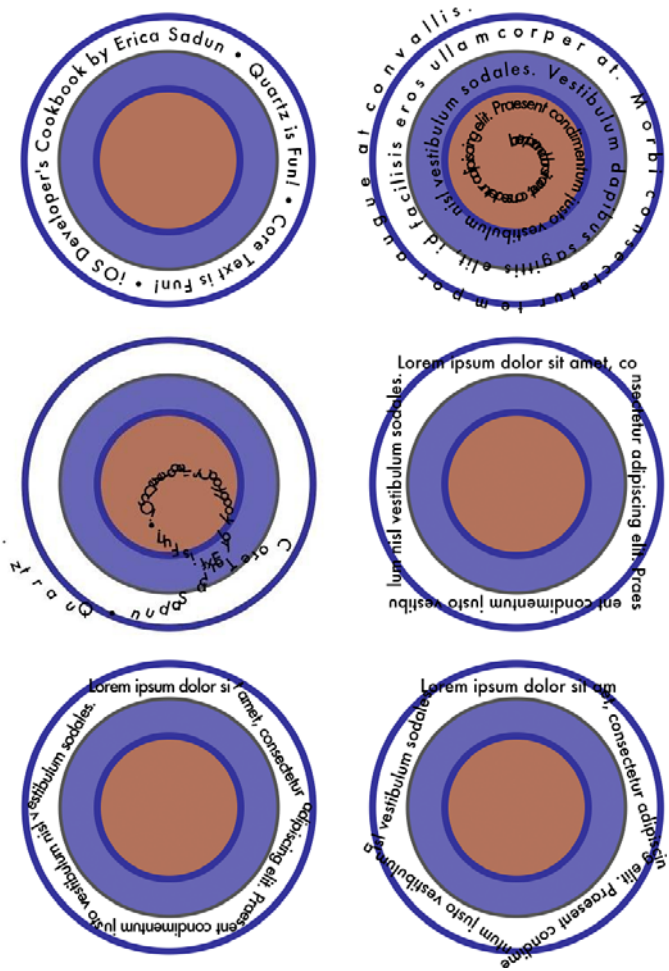


Figure 10-14 This recipe allows you to place custom text onto arbitrary paths.

The remaining images in Figure 10-14 show other easy-to-calculate paths, some of whose utility exceeds that of others. The key to this recipe is not the individual path shapes. It's the generalized method that adapts to any point layout.

Drawing Text onto Bezier Paths

Bezier paths offer a lot of utility for working with paths but do not directly integrate with the array-of-points approach used in this recipe. To use them with this method, you must first retrieve the points that make up the path. The `CGPathApply` function allows you to

From the Library of Eric Berna

iterate through each element in a graphics path, such as a Bezier path, with a custom function.

```
UIBezierPath *bpath = [UIBezierPath bezierPathWithRoundedRect:
    insetRect cornerRadius:64.0f];
NSMutableArray *points = [NSMutableArray array];
CGPathApply(bpath.CGPath, points, getPointsFromBezierCurve);
```

The function applied to the path in this case adds each path point to the points array. This mutable array is passed to the function via its info pointer. This argument is a user data element that you can set to any object from the `CGPathApply()` function call.

```
void getPointsFromBezierCurve (void *info, const CGPathElement *element)
{
    NSMutableArray *bezierPoints = (NSMutableArray *)info;

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

    // Add the points if they're available (per type)
    if (type != kCGPathElementCloseSubpath)
    {
        [bezierPoints addObject:[NSValue valueWithCGPoint:points[0]]];
        if ((type != kCGPathElementAddLineToPoint) &&
            (type != kCGPathElementMoveToPoint))
            [bezierPoints addObject:[NSValue valueWithCGPoint:points[1]]];
    }
    if (type == kCGPathElementAddCurveToPoint)
        [bezierPoints addObject:[NSValue valueWithCGPoint:points[2]]];
}
```

Drawing Proportionately

Once you've retrieved a point array, you'll want to place your text proportionately along those points. Recipe 10-19 calculates the total path length by iterating through the points and determining the distance between each pair.

```
float distance (CGPoint p1, CGPoint p2)
{
    float dx = p2.x - p1.x;
    float dy = p2.y - p1.y;

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

// Calculate the length of the point path
    float totalPointLength = 0.0f;
```

From the Library of Eric Berna

```
    for (int i = 1; i < pointCount; i++)
        totalPointLength += distance(
            [[points objectAtIndex:i] CGPointValue],
            [[points objectAtIndex:i-1] CGPointValue]);
```

Once it has calculated the total point length, it determines the proportional distance for each point and stores that to an array. You need these measures to create a correspondence between the text's glyph-by-glyph progress and the path's point-by-point progress.

```
NSMutableArray *pointPercentArray = [NSMutableArray array];
[pointPercentArray addObject:[NSNumber numberWithFloat:0.0f]];
float distanceTravelled = 0.0f;
for (int i = 1; i < pointCount; i++)
{
        distanceTravelled += distance(
            [[points objectAtIndex:i] CGPointValue],
            [[points objectAtIndex:i-1] CGPointValue]);
        [pointPercentArray addObject:[NSNumber numberWithFloat:
            (distanceTravelled / totalPointLength)]];
}
```

Drawing the Glyph

For each glyph, determine how far it has traveled along its total line and then find the corresponding pair of points in the Bezier path.

```
// Calculate the percent travel
float glyphWidth = CTRunGetTypographicBounds(run, CFRangeMake(runGlyphIndex, 1),
NULL, NULL, NULL);
float percentConsumed = glyphDistance / lineLength;
glyphDistance += glyphWidth;

// Find a corresponding pair of points in the path
CFIndex index = 1;
while ((index < pointPercentArray.count) &&
    percentConsumed >
        [[pointPercentArray objectAtIndex:index] floatValue]))
    index++;
```

Calculate the intermediate distance between those two points. This results in a target point where you'll want to draw the glyph.

```
// Calculate the intermediate distance between the two points
CGPoint point1 = [[points objectAtIndex:index - 1] CGPointValue];
CGPoint point2 = [[points objectAtIndex:index] CGPointValue];

float percent1 = [[pointPercentArray objectAtIndex:index - 1] floatValue];
float percent2 = [[pointPercentArray objectAtIndex:index] floatValue];
float percentOffset = (percentConsumed - percent1) / (percent2 - percent1);
```

From the Library of Eric Berna

```
float dx = point2.x - point1.x;
float dy = point2.y - point1.y;

CGPoint targetPoint = CGPointMake(point1.x + (percentOffset * dx),
    (point1.y + percentOffset * dy));
targetPoint.y = currentContextBounds.size.height - targetPoint.y;
```

You'll want to place each glyph *onto* the path rather than just add it to where the path passes through. An angle of rotation that represents the tangent to the path at that point allows you to do this. This snippet calculates the angle (using the arc tangent) and also flips the text when the text is going left to right. I simply think this looks better, so the text is always on the "outside" of the path. You can omit the extra step if you like.

```
float angle = -atan(dy / dx);
if (dx < 0) angle += M_PI; // going left, update the angle
```

The only step remaining is to draw the glyph in the right place at the right angle. To do this, use transforms that translate and rotate the context into place. Perform the drawing, and then use reverse transformations to restore the context back to its prior state.

```
CGContextTranslateCTM(context, targetPoint.x, targetPoint.y);
CGContextRotateCTM(context, angle);

// ...perform the drawing here...

CGContextRotateCTM(context, -angle);
CGContextTranslateCTM(context, -targetPoint.x, -targetPoint.y);
```

These steps involve a lot of math and overhead but the results can be well worth the effort.

Recipe 10-19 **Drawing Text onto a Path**

```
- (void) drawOnPoints: (NSArray *) points
{
    int pointCount = points.count;
    if (pointCount < 2) return;

    // PRELIMINARY CALCULATIONS

    // Calculate the length of the point path
    float totalPointLength = 0.0f;
    for (int i = 1; i < pointCount; i++)
        totalPointLength += distance(
            [[points objectAtIndex:i] CGPointValue],
            [[points objectAtIndex:i-1] CGPointValue]);

    // Create the typographic line, retrieve its length
    CTLineRef line = CTLineCreateWithAttributedString(
```

From the Library of Eric Berna

```
    (__bridge CFAttributedStringRef)string);
if (!line) return;
double lineLength = CTLineGetTypographicBounds(
    line, NULL, NULL, NULL);

// Retrieve the runs
CFArrayRef runArray = CTLineGetGlyphRuns(line);

// Count the items
int glyphCount = 0; //  Number of glyphs encountered
float runningWidth; //  running width tally
int glyphNum = 0;   //  Current glyph

// Iterate through the run array and calculate a running width
for (CFIndex runIndex = 0;
    runIndex < CFArrayGetCount(runArray); runIndex++)
{
    CTRunRef run = (CTRunRef)CFArrayGetValueAtIndex(
        runArray, runIndex);
    for (CFIndex runGlyphIndex = 0;
        runGlyphIndex < CTRunGetGlyphCount(run); runGlyphIndex++)
    {
        runningWidth += CTRunGetTypographicBounds(
            run, CFRangeMake(runGlyphIndex, 1),
            NULL, NULL, NULL);
        if (!glyphNum && (runningWidth > totalPointLength))
            glyphNum = glyphCount;
        glyphCount++;
    }
}

// Use total length to calculate the % path consumed at each point
NSMutableArray *pointPercentArray = [NSMutableArray array];
[pointPercentArray addObject:[NSNumber numberWithFloat:0.0f]];
float distanceTravelled = 0.0f;
for (int i = 1; i < pointCount; i++)
{
    distanceTravelled += distance(
        [[points objectAtIndex:i] CGPointValue],
        [[points objectAtIndex:i-1] CGPointValue]);
    [pointPercentArray addObject:
        [NSNumber numberWithFloat:
            (distanceTravelled / totalPointLength)]];
}

// Add a final item just to stop with
[pointPercentArray addObject:[NSNumber numberWithFloat:2.0f]];
```

From the Library of Eric Berna

```
// PREPARE FOR DRAWING

NSRange subrange = {0, glyphNum};
NSAttributedString *newString =
    [string attributedSubstringFromRange:subrange];

// Re-establish line and run array for the new string
if (glyphNum)
{
    CFRelease(line);

    line = CTLineCreateWithAttributedString(
        (__bridge CFAttributedStringRef)newString);
    if (!line) {NSLog(@"Error re-creating line"); return;}

    lineLength = CTLineGetTypographicBounds(
        line, NULL, NULL, NULL);
    runArray = CTLineGetGlyphRuns(line);
}

// Keep a running tab of how far the glyphs have travelled to
// be able to calculate the percent along the point path
float glyphDistance = 0.0f;


// Fetch the context and begin to draw
CGContextRef context = UIGraphicsGetCurrentContext();
CGContextSaveGState(context);

// Set the initial positions
CGPoint textPosition = CGPointMake(0.0f, 0.0f);
CGContextSetTextPosition(context, textPosition.x, textPosition.y);

for (CFIndex runIndex = 0;
    runIndex < CFArrayGetCount(runArray); runIndex++)
{
    // Retrieve the run
    CTRunRef run = (CTRunRef)CFArrayGetValueAtIndex(
        runArray, runIndex);

    // Retrieve font and color
    CFDictionaryRef attributes = CTRunGetAttributes(run);
    CTFontRef runFont = CFDictionaryGetValue(
        attributes, kCTFontAttributeName);
    CGColorRef fontColor =
        (CGColorRef) CFDictionaryGetValue(attributes,
```

From the Library of Eric Berna

```
        kCTForegroundColorAttributeName);
CFShow(attributes);
if (fontColor) [[UIColor colorWithCGColor:fontColor] set];

// Iterate through each glyph in the run
for (CFIndex runGlyphIndex = 0;
    runGlyphIndex < CTRunGetGlyphCount(run); runGlyphIndex++)
{
    // Calculate the percent travel
    float glyphWidth = CTRunGetTypographicBounds(run,
        CFRangeMake(runGlyphIndex, 1), NULL, NULL, NULL);
    float percentConsumed = glyphDistance / lineLength;

    // Find a corresponding pair of points in the path
    CFIndex index = 1;
    while ((index < pointPercentArray.count) &&
       (percentConsumed > [[pointPercentArray
           objectAtIndex:index] floatValue]))
        index++;

    // Don't try to draw if we're out of data
    if (index > (points.count - 1)) continue;

    // Calculate the intermediate distance between the two points
    CGPoint point1 =
        [[points objectAtIndex:index - 1] CGPointValue];
    CGPoint point2 =
        [[points objectAtIndex:index] CGPointValue];

    float percent1 =
        [[pointPercentArray objectAtIndex:index - 1] floatValue];
    float percent2 =
        [[pointPercentArray objectAtIndex:index] floatValue];
    float percentOffset =
        (percentConsumed - percent1) / (percent2 - percent1);

    float dx = point2.x - point1.x;
    float dy = point2.y - point1.y;

    CGPoint targetPoint = CGPointMake(
        point1.x + percentOffset * dx,
        point1.y + percentOffset * dy);
    targetPoint.y = view.bounds.size.height - targetPoint.y;
```

From the Library of Eric Berna

```
            // Set the x and y offset in the context
            CGContextTranslateCTM(context,
               targetPoint.x, targetPoint.y);
            CGPoint positionForThisGlyph =
                CGPointMake(textPosition.x, textPosition.y);

            // Rotate the context
            float angle = -atan(dy / dx);
            if (dx < 0) angle += M_PI; // if going left, flip
            CGContextRotateCTM(context, angle);

            // Apply text matrix transform
            textPosition.x -= glyphWidth;
            CGAffineTransform textMatrix = CTRunGetTextMatrix(run);
            textMatrix.tx = positionForThisGlyph.x;
            textMatrix.ty = positionForThisGlyph.y;
            CGContextSetTextMatrix(context, textMatrix);

            // Draw the glyph
            CGGlyph glyph;
            CGPoint position;
            CGFontRef cgFont = CTFontCopyGraphicsFont(runFont, NULL);
            CFRange glyphRange = CFRangeMake(runGlyphIndex, 1);
            CTRunGetGlyphs(run, glyphRange, &glyph);
            CTRunGetPositions(run, glyphRange, &position);
            CGContextSetFont(context, cgFont);
            CGContextSetFontSize(context, CTFontGetSize(runFont));
            CGContextShowGlyphsAtPositions(context, &glyph, &position, 1);

            CFRelease(cgFont);

            // Reset context transforms
            CGContextRotateCTM(context, -angle);
            CGContextTranslateCTM(context,
                -targetPoint.x, -targetPoint.y);

            glyphDistance += glyphWidth;
        }
    }

    CFRelease(line);
    CGContextRestoreGState(context);
}
```

From the Library of Eric Berna

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 10 and open the project for this recipe.

One More Thing: Big Phone Text

On the Macintosh, Address Book's big text display is one of my favorite desktop features. It enlarges text to provide an easy-to-read display of short information, information that you might be able to read even from across the room. So why not use the same make-it-big-and-readable philosophy on your iPhone, iPad, or iPod touch, as shown in Figure 10-15? Big text that is unaffected by device rotation with high contrast display could be a real asset to many applications.

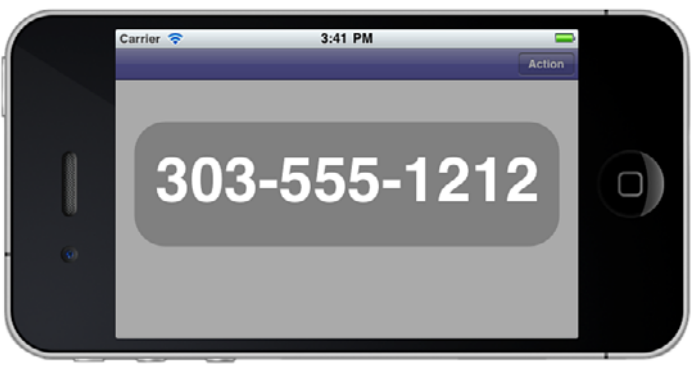


Figure 10-15 High-contrast large text displays can help users show off short snippets of information in a highly readable manner.

That's because big text isn't just about phone numbers. You might display names, level unlock codes, or e-mail addresses, as well as those phone numbers, making it easier to visually transfer information to someone using another iOS device, a physical notepad, a game console, a computer, or even a third-party smart phone. The motivating features here are that the text is short in length, big in size, and easy to read.

Having an algorithmic tool to display otherwise hard-to-read text is a really great software feature, and one whose inspiration I proudly stole from Address Book in order to create this final recipe's reusable Objective-C class. Recipe 10-20's `BigTextView` class is meant to overlay the key device window and will not respond to device orientation, ensuring that text doesn't start moving as the user reorients the device, especially when showing the information to someone else.

The current implementation dismisses the view with a tap, but it can easily be adapted to require a double-tap or a long press to avoid any premature dismissal during typical handling.

From the Library of Eric Berna

Recipe 10-20 **Big Text. Really Big Text.**

```
@implementation BigTextView
- (void) drawRect:(CGRect)rect
{
    [super drawRect:rect];
    CGContextRef context = UIGraphicsGetCurrentContext();

    // Flip coordinates and inset enough so the status bar isn't an issue
    CGRect flipRect = CGRectMake(0.0f, 0.0f,
        self.frame.size.height, self.frame.size.width);
    flipRect = CGRectInset(flipRect, 24.0f, 24.0f);

    // Iterate until finding a set of font traits that fits this rectangle
    // Thanks for the inspiration from the lovely QuickSilver people
    for(fontSize = 24; fontSize < 300; fontSize++ )
    {
        textFont = [UIFont boldSystemFontOfSize:fontSize+1];
        textSize = [string sizeWithFont:textFont];
        if (textSize.width > (flipRect.size.width +
            ([textFont descender] * 2)))
            break;
    }

    // Initialize the string helper to match the traits
    StringHelper *shelper = [StringHelper buildHelper];
    shelper.fontSize = fontSize;
    shelper.foregroundColor = [UIColor whiteColor];
    shelper.alignment = @"Center";
    shelper.fontName = @"GeezaPro-Bold";
    [shelper appendFormat:@"%@", string];

    // Establish a frame that encloses the text at the maximum size
    CGRect textFrame = CGRectZero;
    textFrame.size = [string sizeWithFont:
        [UIFont fontWithName:shelper.fontName size:shelper.fontSize]];

    // Center the destination rect within the flipped rectangle
    CGRect centerRect = rectCenteredInRect(textFrame, flipRect);

    // Flip coordinates so the text reads the right way
    CGContextSetTextMatrix(context, CGAffineTransformIdentity);
    CGContextTranslateCTM(context, 0, self.bounds.size.height);
    CGContextScaleCTM(context, 1.0, -1.0);

    // Rotate 90 deg to write text horizontally along
    // window's vertical axis
    CGContextRotateCTM(context, -M_PI_2);
    CGContextTranslateCTM(context, -self.frame.size.height, 0.0f);
```

From the Library of Eric Berna

```
    // Draw a nice gray backsplash
    [[UIColor grayColor] set];
    CGRect insetRect = CGRectInset(centerRect, -20.0f, -20.0f);
    [[UIBezierPath bezierPathWithRoundedRect:insetRect
        cornerRadius:32.0f] fill];

    // Create a path for the text to draw into
    CGMutablePathRef path = CGPathCreateMutable();
    CGPathAddRect(path, NULL, centerRect);

    // Draw the text
    CTFramesetterRef framesetter =
        CTFramesetterCreateWithAttributedString(
            (__bridge CFAttributedStringRef)shelper.string);
    CTFrameRef theFrame = CTFramesetterCreateFrame(
        framesetter, CFRangeMake(0, shelper.string.length), path, NULL);
    CTFrameDraw(theFrame, context);

    // Clean up
    CFRelease(framesetter);
    CFRelease(path);
    CFRelease(theFrame);
}

+ (void) bigTextWithString:(NSString *)theString
{
    // Add overlay directly to the key window, not to a regular view
    // The drawRect: geometry must compensate for this
    UIWindow *keyWindow = [[UIApplication sharedApplication] keyWindow];
    BigTextView *btv = [[[BigTextView alloc] initWithFrame:
        keyWindow.bounds] autorelease];
    btv.backgroundColor =
        [[UIColor darkGrayColor] colorWithAlphaComponent:0.5f];
    btv.string = theString;
    [keyWindow addSubview:btv];

    return;
}

- (void) touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    // Disappear with a touch -- adjust if you like to use
    // double-tap or whatever
    [self removeFromSuperview];
}
@end
```

From the Library of Eric Berna

> **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 10 and open the project for this recipe.

Summary

This chapter introduced many ways to creatively use text in your iOS applications. In this chapter, you've read about controlling the keyboard and resizing views to accommodate text entry. You've discovered how to create custom input views, to filter text and test it for valid entry, and to work with Core Text. Before you leave this chapter, here are a few final thoughts to take away:

- Don't assume your users will not be using Bluetooth keyboards. Make sure you test your applications with hardware as well as software text entry.
- Although accessory views provide a wonderful way to add extra functionality to your text-input chores, don't overdo the accessories. Keyboards on the iPhone and iPod touch already cover an overwhelming portion of the screen. Adding accessory views diminishes user space even more. So, where possible go spartan and minimal in your accessory design for text entry.
- Never assume your user will *ever* use shake-to-undo. Instead, try to provide undo/redo support directly in your application's GUI, where the user can immediately recognize what to do rather than have to recall that Apple added that obscure feature and that it's available for use. Shake-to-undo should always supplement other undo/redo support, not replace it. Undo/redo buttons are a best-use scenario for accessory views.
- Even though you might not be able to construct a perfect regular expression to test user input, don't discount regular expressions that are good enough to cover most cases. And don't forget that you can always use more than one regular expression in sequence to test different approaches to the same problem.
- With the proper utility classes, Core Text is amazingly simple to use. Don't let its Core Foundation–style C functions scare you off from what is otherwise an easy way to create aesthetically pleasing presentations with very little coding. Take advantage of built-in features such as `NSScanner` and `UIBezierPath` that help support this class.

From the Library of Eric Berna

11

Creating and Managing Table Views

Tables provide a scrolling list-based interaction class that works particularly well for small GUI elements. Many apps that ship natively with the iPhone and iPod touch center on tables, including Contacts, Settings, iPod, and YouTube. On these smaller iOS devices, limited screen size makes tables, with their scrolling and individual item selection, an ideal way to deliver information and content in a simple, easy-to-manipulate form. On the larger iPad, tables integrate with larger detail presentations, providing an important role in split view controllers. In this chapter, you discover how iOS tables work, what kinds of tables are available to you as a developer, and how you can use table features in your own programs.

Introducing `UITableView` and `UITableViewController`

The standard iOS table consists of a simple scrolling list of individual cells. Users may scroll or flick their way up and down until they find an item they want to interact with. Then, they can work with that item independently of other rows. On iOS, tables are ubiquitous. Nearly every standard software package uses them, and they form the core of many third-party applications too. In this section, you discover how tables function and what elements you need to bring together to create your own.

The iOS SDK supports several kinds of tables, many of which are implemented as flavors of the `UITableView` class. In addition to the standard scrolling list of cells, which provides the most generic table implementation, you can create specialized tables. These include the kind of tables you see in the Preferences application, with their blue-gray background and rounded cell edges; tables with sections and an index, such as the ones used in the Contacts application; and related classes of wheeled tables, such as those used to set appointment dates and alarms. No matter what type of table you use, they all work in the same general way. They contain cells provided from a data source and respond to user interactions by calling well-defined delegate methods.

From the Library of Eric Berna

The UITableViewController class derives from the UIViewController class. Like its parent class, it helps you build onscreen presentations with minimal programming and maximum convenience. The UITableViewController class greatly simplifies the process of creating a UITableView, reducing or eliminating the repetitive steps required for working directly with table instances. UITableViewController handles the fussy details for the table view layout and provides table-specific convenience by adding a local tableView instance variable and automatic table protocol support for delegates and data sources.

Creating the Table

To implement tables, you define three key elements: how the table is laid out, the kinds of things used to fill the table, and how the table reacts to user interaction. Specify these elements by adding descriptions and methods to your application. You create the visual layout when building your views, you define a data source that feeds table cells on demand, and you implement delegate methods that respond to user interactions such as row-selection changes.

Laying out the View

UITableView instances are, as the name suggests, views. They present interactive tables on the iPhone screen. The UITableView class inherits from the UIScrollView class. This inheritance provides the up and down scrolling capabilities used by the table. Like other views, UITableView instances define their boundaries through frames, and they can be children or parents of other views. To create a table view, you allocate it, initialize it with a frame just like any other view, and then add all the bookkeeping details by assigning data source and delegate objects.

UITableViewControllers take care of the layout work for you. The UITableViewController class creates a standard UIViewController and populates it with a single UITableView, setting its frame to allow for any navigation bars or toolbars. You may access that table view via the tableView instance variable.

Assigning a Data Source

UITableView instances rely on an external source to feed either new or existing table cells on demand. Cells are small views that populate the table, adding row-based content. This external source is called a "data source" and refers to the object whose responsibility it is to return a cell on request to a table.

The table's dataSource property sets an object to act as a table's source of cells and other layout information. That object must implement the UITableViewDataSource protocol. In addition to returning cells, a table's data source specifies the number of sections in the table, the number of cells per section, any titles associated with the sections, cell heights, an optional table of contents, and more. The data source defines how the table looks and the content that populates it.

From the Library of Eric Berna

Typically, the `UITableViewController` that owns the table view acts as the data source for that view. When working with `UITableViewController` subclasses, you need not declare the protocol because the parent class implicitly supports that protocol and automatically assigns the controller as the data source.

Serving Cells

The table's data source populates the table with cells by implementing the `tableView:cellForRowAtIndexPath:` method. Any time the table's `reloadData` method is invoked, the table starts querying its data source to load the actual onscreen cells into your table. Your code can call `reloadData` at any time to force the table to reload its contents.

Data sources provide table cells based on an index path, which is passed as a parameter to the cell request method. Index paths, objects of the `NSIndexPath` class, describe the path through a data tree to a particular node—namely their section and their row. You can create an index path by supplying a section and row:

```
myIndexPath = [NSIndexPath indexPathForRow:5 inSection:0];
```

Tables use sections to split data into logical groups and rows to index members within each group. It's the data source's job to associate an index path with a concrete `UITableViewCell` instance and return that cell on demand.

Your data source can respond to cell requests by building cells from code or it can load its cells from Interface Builder sources. Typically, you use a built-in mechanism for reusing table cells before creating any new cells. When cells scroll off the table and out of view, the table can cache them into a reuse queue. You tag these cells for reuse and then pop them off that queue as needed. This saves memory and provides a fast, efficient way to feed cells when users scroll quickly through long lists onscreen.

You're not limited to single cell types. Mix and match cells within a table however you need. The following snippet chooses which of two kinds of cells to request from the reusable cell queue. If a cell cannot be dequeued, one is created. Default-styled cells provide a single label; subtitle cells add a second. The identifier used here is arbitrary, as defined by the developer. It helps differentiate between kinds of cells in use.

```
// Choose an identifier based on model object properties
if (item.notes)
    identifier = @"notescell";
else
    identifier = @"basecell";

// Attempt to dequeue an existing cell with that identifier
cell = [aTableView dequeueReusableCellWithIdentifier:identifier];

// Create a new cell if needed
if (!cell)
```

From the Library of Eric Berna

```
{
    style = item.notes ? UITableViewCellStyleSubtitle :
        UITableViewCellStyleDefault;
    cell = [[UITableViewCell alloc] initWithStyle:style
        reuseIdentifier:identifier];
}
```

Assigning a Delegate

Like many other Cocoa Touch interaction objects, `UITableView` instances use delegates to respond to user interactions and implement a meaningful response. Your table's delegate can respond to events such as the table scrolling or row selection changes. Delegation tells the table to hand off responsibility for reacting to these interactions to the object you specify, typically the `UITableViewController` object that owns the table view.

If you're working directly with a `UITableView`, assign the `delegate` property to a responsible object. The delegate must implement the `UITableViewDelegate` protocol. As with data sources, you may skip setting the delegate when working with `UITableViewController` or its custom subclass. That class already handles this assignment.

Recipe: Implementing a Basic Table

The `UITableViewController` class embeds a `UITableView` into a `UIViewController` object that manages its table view. This view is accessed via the `tableView` property. These controllers automatically set the data source and delegate methods for the table view to itself. So it's really a plug-and-play situation. For a really basic table, all you need to bring to the table is some data and a few data source functions that feed cells and report the number of rows and sections.

Populating a Table

Pretty much any array of strings can be used to set up and populate a table. Recipe 11-1 builds an array of strings from a collection of letters and uses this to create its flat (nonsectioned) table presentation.

```
items = [@"A*B*C*D*E*F*G*H*I*J*K*L"
    componentsSeparatedByString:@"*"];
```

Figure 11-1 shows the simple table created by this recipe. As Recipe 11-1 shows, the coding to support this data model and to produce that table is absolutely minimal.

From the Library of Eric Berna

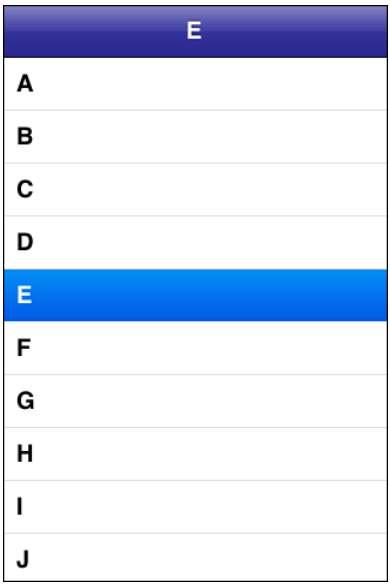


Figure 11-1 It's easy to fill a `UITableView` with cells based on any array of strings. This table contains letters, which it presents as a flat list without sections.

Data Source Methods

To display a table, every table data source must implement three core methods. These methods define how the table is structured and provide contents for the table:

- **`numberOfSectionsInTableView`**—Tables can display their data in sections or as a single list. For simple tables, return 1. This indicates that the entire table should be presented as one single list. For sectioned lists, return a value of 2 or higher.
- **`tableView:numberOfRowsInSection:`**—This method returns the number of rows for each section. When working with simple lists, return the number of rows for the entire table here. For more complex lists, you'll want to provide a way to report back per section. As with all counting in iOS, section ordering starts with 0 as the first section.
- **`tableView:cellForRowAtIndexPath:`**—This method returns a cell to the calling table. Use the index path's `row` and `section` properties to determine which cell to provide and make sure to take advantage of reusable cells where possible to minimize memory overhead.

From the Library of Eric Berna

Reusing Cells

One of the ways the iPhone conserves memory is by reusing cells. You can assign an identifier string to each cell. This specifies what kind of cell it is, and when that cell scrolls offscreen allows it to be recovered for reuse. Use different IDs for different kinds of cells for tables that use multiple cell styles. For simple tables, a single identifier does the job. In the case of Recipe 11-1, it is `@"BaseCell"`. The strings are arbitrary. Define them the way you want, but when using multiple cell types keep the names meaningful.

Before allocating a new cell, always check whether a reusable cell is available. If your table returns `nil` from a request to `dequeueReusableCellWithIdentifier:`, you need to allocate a new cell. On a standard iPhone device (iPhone and iPod touch), you'll never use more than a dozen or so cells at a time.

If the method returns a cell, update that cell with the information that's meaningful for the current row and section indices. You do not need to add cells to the reuse queue. Cocoa Touch handles all those details for you, as cells scroll offscreen.

Responding to User Touches

Recipe 11-1 creates a table and fills that table with a list of letters. When the user taps a cell, the view controller sets the letter as its title, displaying it in the navigation bar at the top of the display. This behavior is defined in the `tableView:didSelectRowAtIndexPath:` delegate method, which is called when a user taps a row.

Recipe 11-1 **Building a Basic Table**

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)aTableView
{
    // This simple table has only one section
    return 1;
}

- (NSInteger)tableView:(UITableView *)aTableView
    numberOfRowsInSection:(NSInteger)section
{
    // Return the number of items for the single section
    return items.count;
}

- (UITableViewCell *)tableView:(UITableView *)aTableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Dequeue or create a cell
    UITableViewCellStyle style = UITableViewCellStyleDefault;
    UITableViewCell *cell =
        [aTableView dequeueReusableCellWithIdentifier:@"BaseCell"];
```

From the Library of Eric Berna

```
    if (!cell)
        cell = [[UITableViewCell alloc] initWithStyle:style
            reuseIdentifier:@"BaseCell"];

    cell.textLabel.text = [items objectAtIndex:indexPath.row];
    return cell;
}

- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Respond to user interaction
    self.title = [items objectAtIndex:indexPath.row];
}

- (void) loadView
{
    [super loadView];
    items = [@"A*B*C*D*E*F*G*H*I*J*K*L"
        componentsSeparatedByString:@"*"];
}
```

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 11 and open the project for this recipe.

Selection Color

Tables enable you to set the color for the selected cell by choosing between a blue or gray overlay. Set the `selectionStyle` property to either `UITableViewCellSelectionStyleBlue` or `UITableViewCellSelectionStyleGray`. If you'd rather not show a selection, use `UITableViewCellSelectionStyleNone`. The cell can still be selected, but the overlay color will not display.

Changing a Table's Background Color

To use a color for your table's background other than white, assign the table view's `backgroundColor` property. Individual cells inherit this color, producing a table whose components all show that color. Make sure you choose a cell text color that compliments any table background color.

You can add a backdrop to a table by setting its background color to clear (`[UIColor clearColor]`) and placing an image behind it. The image will "bleed" through the table, allowing you to provide a custom look to your table.

From the Library of Eric Berna

The table scrolls over the image, which remains static behind it. Keep the imagery relevant and geometrically sensible so as not to interfere with the table's text presentation. Use a text color that contrasts well with any background image.

Cell Types

The iPhone offers four kinds of base table view cells. These types, which are shown in Figure 11-2, provide utilitarian cell implementations.



Figure 11-2 Cocoa Touch provides four standard cell types, some of which support optional images.

Cells provide both a `textLabel` and a `detailTextLabel` property, which offer access to the labels themselves. With direct label access, you can set each label's text traits as desired. Here is a roundup of the four new styles:

- **`UITableViewCellStyleDefault`**—This cell offers a single left-aligned text label and an optional image. When images are used, the label is pushed to the right, decreasing the amount of space available for text. You can access and modify the `detailTextLabel`, but it is not shown onscreen.
- **`UITableViewCellStyleSubtitle`**—This cell, which is used in the iPod application, pushes the standard text label up a bit to make way for the smaller detail label beneath it. The detail label displays in gray. Like the default cell, the subtitle cell offers an optional image.

From the Library of Eric Berna

- **`UITableViewCellStyleValue1`**—This cell style, seen in the Settings application, offers a large, black primary label on the left side of the cell and a slightly smaller, blue subtitle detail label to its right. This cell does not support images.
- **`UITableViewCellStyleValue2`**—The Phone/Contacts application uses this kind of cell, which consists of a small blue primary label on the left and a small black subtitle detail label to its right. The small width of the primary label means that most text will be cut off by an ellipsis. This cell does not support images.

Recipe: Building Custom Cells in Interface Builder

Interface Builder helps you create custom `UITableViewCell` instances without subclassing. You can build your cells directly in IB and load them in your code, and even take advantage of cell reuse, which is what Recipe 11-2 does.

In Xcode, create a new .xib file with File > New > New File > User Interface > Empty. Choose an iPhone template. Name it CustomCell.xib and then save it. Open the empty .xib file in the Interface Builder editor pane and drag a `UITableViewCell` into the editor. Set the cell to resize horizontally in the size inspector, setting both horizontal struts as well.

Customize the cell contents by adding art and other interface items. Most developers will want to add a custom backsplash to their cells, which is a tiny bit complicated. That's because by default, the cell contents will obscure any custom art, as shown in Figure 11-3 (left). You need to set the table's background color to clear in order to expose the art, as in Figure 11-3 (right). When doing this, ensure that your art contains a solid background color to form the visual backsplash for your table.

Be aware that text-editing-based classes such as `UITextField` and `UITextView` do not work well in table view cells unless you take special care. When adding custom items, try to clear enough space (about 40 points) on the right side of the cell to allow the cell to shift right when entering edit mode. Otherwise, those items will be cut off.

You can set the cell's image and selected image using the inspector, but in real life, these are usually generated based on live data. You'll probably want to handle any image setting (via the `image` and `selectedImage` properties) in code. Make sure that the images you send are properly sized or that you use the attributes inspector to set the image view mode to fill or fit your art. See the recipes about creating thumbnail versions of images in Chapter 7, "Working with Images."

Set the reuse identifier (for example, "CustomCell") in the cell's attributes inspector. The identifier field lies near the top of the inspector. This identifier allows you to register the cell for reuse, as shown in Recipe 11-2's `loadView` method. Once it is registered, you can simplify the cell production method to use the NIB loader to retrieve and create cells as needed.

You cannot pick a cell style in Interface Builder as of the time this book was being written, and you cannot change a cell style once you've loaded the NIB. If you need to use a cell style other than the default, build your cell in code. Use any cell height you need and then set the table's `rowHeight` property to match.

From the Library of Eric Berna

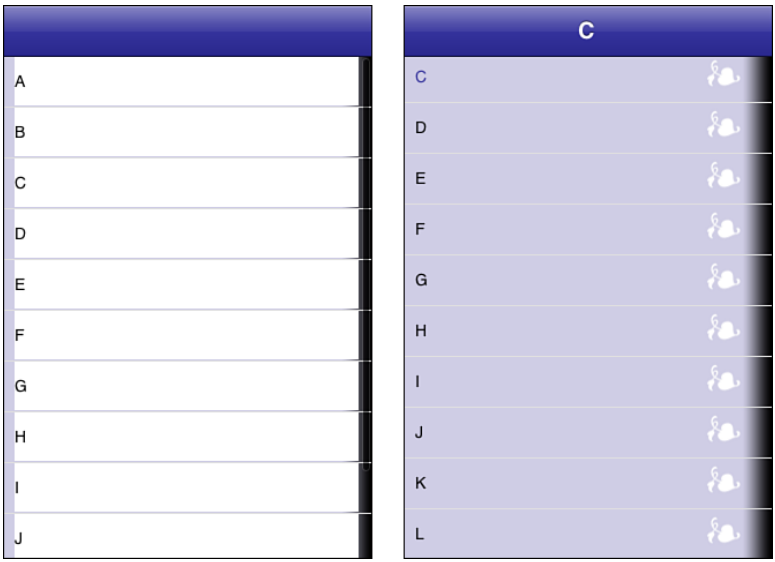


Figure 11-3 Table cell labels will obscure background art unless the table's background color is set to clear.

Recipe 11-2 **Using Custom Cells Built-in Interface Builder**

```
- (UITableViewCell *)tableView:(UITableView *)aTableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Use the built-in nib loader
    UITableViewCell *cell =
        [aTableView dequeueReusableCellWithIdentifier:@"CustomCell"];

    NSString *title = [items objectAtIndex:indexPath.row];
    cell.textLabel.text = title;
    return cell;
}

- (void) loadView
{
    [super loadView];
    items = [@"A*B*C*D*E*F*G*H*I*J*K*L"
        componentsSeparatedByString:@"*"];

    // Register the nib for reuse
    [self.tableView registerNib: [UINib nibWithNibName:@"CustomCell"
        bundle:[NSBundle mainBundle]]
        forCellReuseIdentifier:@"CustomCell"];
```

From the Library of Eric Berna

```
    // Allow the custom background to be seen
    self.tableView.backgroundColor = [UIColor clearColor];
}
```

> **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 11 and open the project for this recipe.

Adding in Custom Selection Traits

When users select cells, Cocoa Touch helps you emphasize the cell's selection. Customize a cell's selection behavior by updating its traits to stand out from its fellows. There are two ways to do this.

The `selectedBackgroundView` property allows you to add controls and other views to just the currently selected cell. This works in a similar manner to the accessory views that appear when a keyboard is shown. You might use the selected background view to add a preview button or a purchase option to the selected cell.

The cell label's `highlightedTextColor` property can be set in code or in IB. It lets you choose an alternative text color when the cell is selected.

Alternating Cell Colors

Although blue and white cell alternation is a common and highly requested table feature, Apple has not included that option in its iOS SDK. Recipe 11-2 showed how to import a cell designed in Interface Builder. You can easily expand that approach to alternate between two cells. A simple even/odd check (`row % 2`) could specify whether to load a blue or white cell. Because this table uses just one section, it simplifies the math considerably. Blue/white alternating cells work best for nongrouped, nonsectioned tables both visually and programmatically.

Be aware that although this cell style works with edits (both deletion and reordering), you'll want to reload the table after each user change to keep the blue/white/blue/white ordering. As the user drags an item into place, it will retain its original coloring, possibly causing a visual discontinuity until the edit finishes. For reordering, issue that reload command using a delayed selector of at least a quarter to half a second.

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Choose the cell kind
    NSString *identifier = (indexPath.row % 2) ?
        @"WhiteCell" : @"BlueCell";

    // Dequeue/load the properly colored cell
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier: identifier];
```

From the Library of Eric Berna

```
    // Create a new cell if needed
    if (!cell)
        cell = [[[UINib nibWithNibName:identifier bundle:[NSBundle mainBundle]]
            instantiateWithOwner:self options:nil] lastObject];
    cell.textLabel.backgroundColor = [UIColor clearColor];

    // Update the text, return the cell
    cell.textLabel.text = [titleArray objectAtIndex:indexPath.row];
    return cell;
}
```

Removing Selection Highlights from Cells

At times, when working with tables, you need to avoid retaining a cell state. This happens when you want users to be able to interact with the table and touch cells, but you don't want to maintain that selected state after the user has finished the interaction. Cocoa Touch offers two approaches for tables that need to deny persistent cell selection.

For the first approach, you can set a cell's `selectionStyle` property to `UITableViewCellSelectionStyleNone`. This disables the blue or gray overlays that display on the selected cell. The cell is still selected but will not highlight on selection in any way. If selecting your cell produces some kind of side effect other than presenting information, this is not the best way to approach things.

Another approach allows the cell to highlight but removes that highlight after the interaction completes. You do that by telling the table to deselect the cell in question. In the following snippet, each user selection triggers a delayed deselection (the custom `deselect:` method defined in the recipe) after a half a second. This method calls the table view's `deselectRowAtIndexPath:animated:` method, which fades away the current selection. Using this approach offers both the highlight that confirms a user action and the state-free display that hides any current selection from the user.

```
// Perform the deselection
- (void) deselect: (id) sender
{
    [self.tableView deselectRowAtIndexPath:[self.tableView
        indexPathForSelectedRow] animated:YES];
}

// Respond to user selection
- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)newIndexPath
{
    printf("User selected row %d\n", [newIndexPath row] + 1);
    [self performSelector:@selector(deselect:) withObject:nil
        afterDelay:0.5f];
}
```

From the Library of Eric Berna

Creating Grouped Tables

On the iPhone, tables come in two formats: grouped tables and plain table lists. The iOS Settings application offers grouped lists in action. These lists display on a blue-gray background, and each subsection appears within a slightly rounded rectangle.

To change styles requires nothing more than initializing the table view controller with a different style. You can do this explicitly when creating a new instance. Here's an example:

```
myTableViewController = [[UITableViewController alloc]
    initWithStyle:UITableViewStyleGrouped];
```

Or you can override the `init` method in your `UITableViewController` subclass to ensure that new instances of this subclass produce a grouped-style table. You cannot change this attribute on the fly after the table is built, so this method localizes that setting to the table's creation.

```
- (TableListViewController *) init
{
    // Set the grouped style
    self = [super initWithStyle:UITableViewStyleGrouped];
    return self;
}
```

Recipe: Remembering Control State for Custom Cells

Cells have no "memory" to speak of. They do not know how an application last used them. They are views and nothing more. That means if you reuse cells without tying those cells to some sort of data model, you can end up with unexpected and unintentional results. This is a natural consequence of the Model-View-Controller design paradigm.

Consider the following scenario. Say you created a series of cells, each of which owned a toggle switch. Users can interact with that switch and change its value. A cell that scrolls offscreen, landing on the reuse queue, could therefore show an already-toggled state for a table element that user hasn't yet touched.

Figure 11-4 demonstrates this problem. The cell used for Item A was reused for Item L, presenting an OFF setting, even though the user has never interacted with Item L. It's the cell that retains the setting, not the logical item. Don't depend on cells to retain state that way.

To fix this problem, check your cell state against a stored model. This keeps the view consistent with your application semantics. Recipe 11-3 uses a custom dictionary to associate cell state with the cell item. There are other ways to approach this problem, but this simple example provides a taste of the model/view balance needed by a data source whose views present state information.

From the Library of Eric Berna

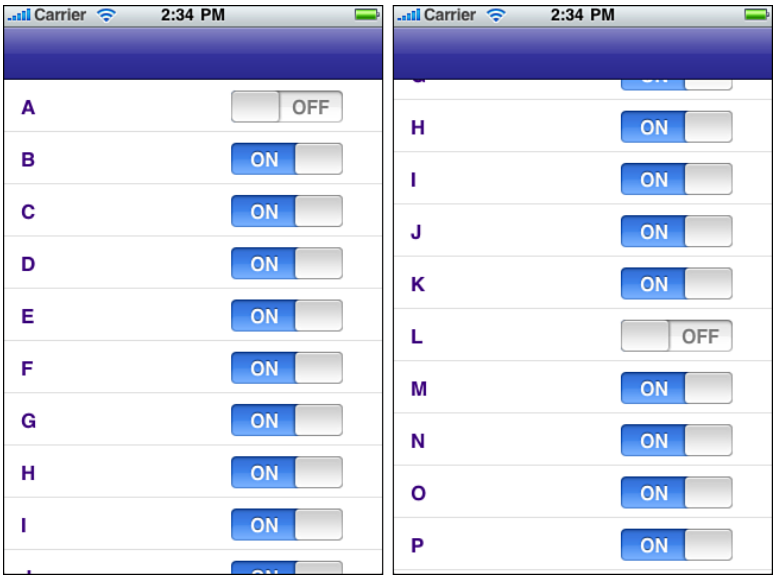


Figure 11-4 The cell used to present Item A (left) is reused to present Item L (right) while retaining its previous switch setting.

Because the switch state is Boolean, this recipe implements a simple `NSMutableDictionary` extension. This category simplifies storing and retrieving Boolean state. It's inspired by the way that `NSUserDefaults` allows direct Boolean access to its values.

```
@interface NSMutableDictionary (Boolean)
- (BOOL) boolForKey: (NSString *) aKey;
- (void) setBool: (BOOL) boolValue ForKey: (NSString *) aKey;
@end

@implementation NSMutableDictionary (Boolean)
- (BOOL) boolForKey: (NSString *) aKey
{
    if (![self objectForKey:aKey]) return NO;

    id obj = [self objectForKey:aKey];

    if ([obj respondsToSelector:@selector(boolValue)])
        return [(NSNumber *)obj boolValue];

    return NO;
}
```

From the Library of Eric Berna

```
- (void) setBool: (BOOL) boolValue ForKey: (NSString *) aKey
{
    [self setObject:[NSNumber numberWithBool:boolValue] forKey:aKey];
}
@end
```

Each cell needs to be able to "call home," so to speak, when its switch value changes, so it can update the stored state. Although a custom cell could store some kind of link between the cell and its model, Recipe 11-3 utilizes a far simpler approach. Because the data is completely linear and is indexed by row, this code tags the cell's `contentView`. That number translates to a key, indexing into the stored state dictionary.

Recipe 11-3 **Using Stored State to Refresh a Reused Table Cell**

```
- (void) toggleSwitch: (UISwitch *) aSwitch
{
    // Store the state for the active switch
    [switchStates setBool:aSwitch.isOn
        ForKey:NUMSTR(aSwitch.superview.tag)];
}

- (UITableViewCell *)tableView:(UITableView *)aTableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Use the built-in nib loader
    UITableViewCell *cell =
        [aTableView dequeueReusableCellWithIdentifier:@"CustomCell"];

    // Retrieve the switch and add a target if needed
    UISwitch *switchView = (UISwitch *)[cell viewWithTag:99];
    if (![switchView allTargets].count)
        [switchView addTarget:self action:@selector(toggleSwitch:)
            forControlEvents:UIControlEventValueChanged];

    // Set the cell label
    cell.textLabel.text = [items objectAtIndex:indexPath.row];

    // Comment this out to see "wrong" behavior
    switchView.on = [switchStates boolForKey:NUMSTR(indexPath.row)];

    // Tag the cell's content view
    cell.contentView.tag = indexPath.row;

    return cell;
}

- (void) loadView
{
```

From the Library of Eric Berna

```
    [super loadView];
    items = [@"A*B*C*D*E*F*G*H*I*J*K*L*M*N*O*P*Q*R*S*T*U*V*W*X*Y*Z"
        componentsSeparatedByString:@"*"];

    [self.tableView registerNib:[UINib nibWithNibName:@"CustomCell"
        bundle:[NSBundle mainBundle]]
        forCellReuseIdentifier:@"CustomCell"];

    self.tableView.backgroundColor = [UIColor clearColor];

    switchStates = [NSMutableDictionary dictionary];
}
```

> **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 11 and open the project for this recipe.

Visualizing Cell Reuse

Recipe 11-3 helps fix problems with cell/model discrepancies. The following code snippet visualizes exactly how your cells are getting reused. This implementation tags each new cell on creation, letting you track how each cell is used and reused in the lifetime of a very large table. In this case, the table is about a million items long. I encourage you to test this snippet (a full version is included in the sample code for this book) and energetically scroll through the list in both directions. You'll see that with a jerky enough interaction style, you can really mix up your cell ordering. You'll also discover that even for a million item table, you'll max out at about 11 table cells total on the iPhone and 23 table cells total on the iPad, assuming portrait orientation and a navigation bar. Landscape orientation uses even fewer cells (7 and 17, respectively) for a navigation-bar-headed table view.

```
@implementation TableListViewController
- (NSInteger)numberOfSectionsInTableView:(UITableView *)aTableView
{
    return 1;
}

- (NSInteger)tableView:(UITableView *)aTableView
    numberOfRowsInSection:(NSInteger)section
{
    return 999999; // lots
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
```

From the Library of Eric Berna

```
{
    UITableViewCellStyle style =  UITableViewCellStyleDefault;
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:@"BaseCell"];

    // Create a new cell with a unique number whenever
    // a cell cannot be dequeued
    if (!cell)
    {
        cell = [[UITableViewCell alloc] initWithStyle:style
            reuseIdentifier:@"BaseCell"];
        cell.textLabel.text = [NSString stringWithFormat:
            @"Cell %d", ++count];
    }
    return cell;
}
@end
```

Each cell implements the `prepareForReuse` method, which is invoked before a cell can be returned from the table view's dequeue request. You can subclass `UITableViewCell` and override this method to reset content before reusing a cell.

Creating Checked Table Cells

Accessory views expand normal `UITableViewCell` functionality. The most common accessories are the Delete buttons and drag bars for reordering, but you can also add check marks to create interactive one-of-*n* or *n*-of-*n* selections. With these kinds of selections, you can ask your users to pick what they want to have for dinner or choose which items they want to update. This kind of radio button/check box behavior provides a richness of table interaction.

Figure 11-5 shows checks in an interface—a standard `UITableView` with accessorized cells. Check marks appear next to selected items. When tapped, the checks toggle on or off. You'll want to use Recipe 11-3's state dictionary approach to track which logical items are checked, avoiding inconsistency issues that arise from cell reuse.

Checked items use the `UITableViewCellAccessoryCheckmark` accessory type. Unchecked items use the `UITableViewCellAccessoryNone` variation. You set these by assigning the cell's `accessoryType` property.

```
// Set cell checkmark
NSNumber *checked = [stateDictionary objectForKey:key];
if (!checked) [stateDictionary
    setObject: (checked = [NSNumber numberWithBool:NO])
    forKey:key];
cell.accessoryType = checked.boolValue ?
    UITableViewCellAccessoryCheckmark :
    UITableViewCellAccessoryNone;
```

From the Library of Eric Berna

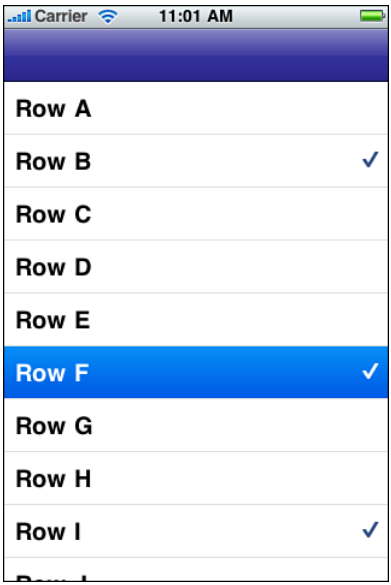


Figure 11-5 Check mark accessories offer a convenient way of making one-of-*n* or *n*-of-*n* selections from a list.

Note that it's the cell that's being checked here, not the logical item associated with the cell (although that logical item's value is updated in the shared `stateDictionary`). Reused cells remain checked or unchecked at next use, so you must always set the accessory to match the state dictionary when dequeuing a cell.

Working with Disclosure Accessories

Disclosures refer to those small, blue or gray, right-facing chevrons found on the right of table cells. Disclosures help you to link from a cell to a view that supports that cell. In the Contacts list and Calendar applications on the iPhone and iPod touch, these chevrons connect to screens that help you to customize contact information and set appointments. Figure 11-6 shows a table view example where each cell displays a disclosure control, showing the two available types.

On the iPad, you should consider using a split view controller (see Chapter 5, "Working with View Controllers") rather than disclosure accessories. The greater space on the iPad display allows you to present both an organizing list and its detail view at the same time, a feature that the disclosure chevrons attempt to mimic on the smaller iPhone units.

The blue and gray chevrons play two roles. The blue `UITableViewCellAccessoryDetailDisclosureButton` versions are actual buttons. They respond to touches and are supposed to indicate that the button leads to a full interactive detail view. The gray `UITableViewCellAccessoryDisclosureIndicator` does not track

From the Library of Eric Berna

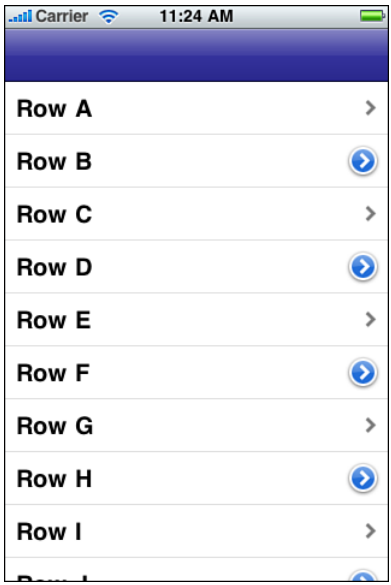


Figure 11-6 The right-pointing chevrons indicate disclosure controls, allowing you to link individual table items to another view.

touches and should lead your users to a further options view—specifically, options about that choice.

You see these two accessories in play in the Settings application. In the Wi-Fi Networks screen, the detail disclosures lead to specific details about each Wi-Fi network: its IP address, subnet mask, router, DNS, and so forth. The disclosure indicator for "Other" enables you to add a new network by scrolling up a screen for entering network information. A new network then appears with its own detail disclosure.

You also find disclosure indicators whenever one screen leads to a related submenu. When working with submenus, stick to the simple gray chevron. The rule of thumb is this: Submenus use gray chevrons, and object customization uses blue ones. Respond to cell selection for gray chevrons and to accessory button taps for blue chevrons.

The following snippet demonstrates how to use disclosure buttons (the blue accessories) in your applications. This code sets the `accessoryType` for each cell to `UITableViewCellAccessory-DetailDisclosureButton`. Importantly, it also sets `editingAccessoryType` to `UITableViewCellAccessoryNone`. When delete or reorder controls appear, your disclosure chevron will hide, enabling your users full control over their edits without accidentally popping over to a new view.

To handle user taps on the disclosure, the `tableView:accessoryButtonTappedForRowWithIndexPath:` method enables you to determine the row that was tapped and implement some appropriate response. In real life,

From the Library of Eric Berna

you'd move to a view that explains more about the selected item and enables you to choose from additional options.

Gray disclosures use a different approach. Because these accessories are not buttons, they respond to cell selection rather than the accessory button tap. Add your logic to `tableView:didSelectRowAtIndexPath:` to push the disclosure view onto your navigation stack or by presenting a modal view controller.

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Use the built-in nib loader
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:@"CustomCell"];

    // Set cell label
    cell.textLabel.text = [items objectAtIndex:indexPath.row];

    // Specify the accessory types
    cell.accessoryType =
        UITableViewCellAccessoryDetailDisclosureButton;
    cell.editingAccessoryType = UITableViewCellAccessoryNone;

    return cell;
}

// Respond to accessory button taps
-(void) tableView:(UITableView *)tableView
    accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath
{
    // Do something here
}
```

Recipe: Table Edits

Bring your tables to life by adding editing features. Table edits transform static information display into an interactive scrolling control that invites your user to add and remove data. Although the bookkeeping for working with table edits is moderately complex, the same techniques easily transfer from one app to another. Once you master the basic elements of entering and leaving edit mode and supporting undo, these items can be reused over and over.

Recipe 11-4 introduces a table that responds meaningfully to table edits. In this example, users create new cells by tapping an Add button and may remove cells either by swiping or entering edit mode (tapping Edit) and using the red remove controls (see Figure 11-7). Users leave edit mode by tapping Done. In day-to-day use, every iOS user quickly

From the Library of Eric Berna

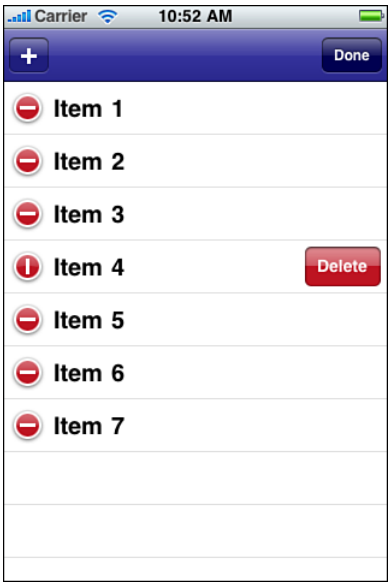


Figure 11-7 Red remove controls allow your users to interactively delete items from a table.

becomes familiar with the small, red circles that let him or her delete cells from tables. Many users also pick up on basic swipe-to-delete functionality.

Displaying Remove Controls

The iOS SDK displays table-based remove controls with a single call, `[self.tableView setEditing:YES animated:YES]`. This updates the table's `editing` property and displays the remove controls shown in Figure 11-7. The animated parameter is optional but recommended. As a rule, use animations in your iOS interfaces to lead your users from one state to the next, so they're prepared for the mode changes that happen onscreen.

Recipe 11-4 defines an `enterEditMode` that moves the table into an editing state. When a user taps the Edit button, this method removes the current item selection, calls the `setBarButtonItems` method that swaps out the title from "Edit" to "Done," and enables the table's editing property.

Dismissing Remove Controls

When users complete their edits and want to return to normal table display, they tap Done. The `leaveEditMode` method works in reverse. It dismiss the controls (`[self.tableView setEditing:NO animated:YES]`) and updates the navigation bar button.

From the Library of Eric Berna

Recipe 11-4 checks whether any items remain, hiding the Edit button if none do, and showing it otherwise.

Handling Delete Requests

On row deletion, the table communicates with your application by issuing a `tableView:commitEditingStyle:forRowAtIndexPath:` callback. A table delete removes an item from the visual table but does not alter the underlying data. Unless you manage the item removal from your data source, the "deleted" item will reappear on the next table refresh. This method offers the place for you to update your data source and respond to the row deletion that the user just performed.

Here is where you actually delete the item from the data structure that supplies the data source methods (in this recipe, through an `NSMutableArray` of item titles) and handle any real-world action such as deleting files, removing contacts, and so on, that occur as a consequence of the user-led edit.

Supporting Undo

Notice that both adding and deleting items are handled by the same method, `updateItemAtIndexPath:withString:`. The method works like this: Any non-`nil` string is inserted at the index path. When the passed string is `nil`, the item at that index path is deleted instead.

This may seem like an odd way to handle requests, as it involves an extra method and extra steps, but there's an underlying motive. This approach provides a unified foundation for undo support, allowing simple integration with undo managers.

The method, therefore, has two jobs to do. First, it prepares an undo invocation. That is, it tells the undo manager how to reverse the edits it is about to apply. Second, it applies the actual edits, making its changes to the `items` array and updating the table and bar buttons.

Swiping Cells

Swiping provides a clean method for removing items from your `UITableView` instances. To enable swipes, simply provide the commit-editing-style method. The table takes care of the rest.

To swipe, users drag swiftly from the left to the right side of the cell. The rectangular delete confirmation appears to the right of the cell, but the cells do *not* display the round remove controls on the left.

After users swipe and confirm, the `tableView:commitEditingStyle:forRowAtIndexPath:` method applies data updates just as if the deletion had occurred in edit mode.

Adding Cells

Recipe 11-4 introduces an Add button to create new content for the table. This uses a system bar button item, which displays as a plus sign. (See the top-left corner of

From the Library of Eric Berna

Figure 11-7.) The `addItem:` method in Recipe 11-4 appends a new cell title at the end of the `items` array and then tells the table to update the data source using `reloadData`. This lets the normal table mechanism check the data and re-create the table view using the updated data source.

Recipe 11-4 **Editing Tables**

```
- (void) setBarButtonItems
{
    // Always display the Add button
    self.navigationItem.leftBarButtonItem =
        SYSBARBUTTON(UIBarButtonSystemItemAdd,
            @selector(addItem:));

    // Display Edit/Done button as needed
    if (self.tableView.isEditing)
        self.navigationItem.rightBarButtonItem =
            SYSBARBUTTON(UIBarButtonSystemItemDone,
                @selector(leaveEditMode));
    else
        self.navigationItem.rightBarButtonItem = items.count ?
            SYSBARBUTTON(UIBarButtonSystemItemEdit,
                @selector(enterEditMode)) : nil;
}

-(void)enterEditMode
{
    // Deselect the current row
    [self.tableView deselectRowAtIndexPath:
        [self.tableView indexPathForSelectedRow] animated:YES];

    // Start editing. Update the bar buttons
    [self.tableView setEditing:YES animated:YES];
    [self setBarButtonItems];
}

-(void)leaveEditMode
{
    // Stop editing. Update the bar buttons.
    [self.tableView setEditing:NO animated:YES];
    [self setBarButtonItems];
}

// This method creates a single entry point for adding and
// deleting items, which better supports adding undo control.
// Passing nil as the string is a delete action, otherwise
// it is an insert/add action.
```

From the Library of Eric Berna

```
- (void) updateItemAtIndexPath: (NSIndexPath *) indexPath
    withString: (NSString *) string
{
    // Create an undo invocation before applying any action
    NSString *undoString =
        string? nil : [items objectAtIndex:indexPath.row];
    [[self.undoManager prepareWithInvocationTarget:self]
        updateItemAtIndexPath:indexPath withString:undoString];

    // Passing nil is a delete request
    if (!string)
        [items removeObjectAtIndex:indexPath.row];
    else
        [items insertObject:string atIndex:indexPath.row];

    // Refresh the table and reload the bar buttons
    [self.tableView reloadData];
    [self setBarButtonItems];
}

// Add a new string
- (void) addItem: (id) sender
{
    // Add a new item
    NSIndexPath *newPath =
        [NSIndexPath indexPathForRow:items.count inSection:0];
    NSString *newTitle =
        [NSString stringWithFormat:@"Item %d", ++count];
    [self updateItemAtIndexPath:newPath withString:newTitle];
}

// Delete a string
- (void)tableView:(UITableView *)aTableView
    commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
    forRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Delete item
    [self updateItemAtIndexPath:indexPath withString:nil];
}
```

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 11 and open the project for this recipe.

From the Library of Eric Berna

Reordering Cells

You empower your users when you allow them to directly reorder the cells of a table. Figure 11-8 shows a table displaying the reorder control's stacked gray lines. Users can apply this interaction to sort to-do items by priority or choose which songs should go first in a playlist. iOS ships with built-in table reordering support that's easy to add to your applications.

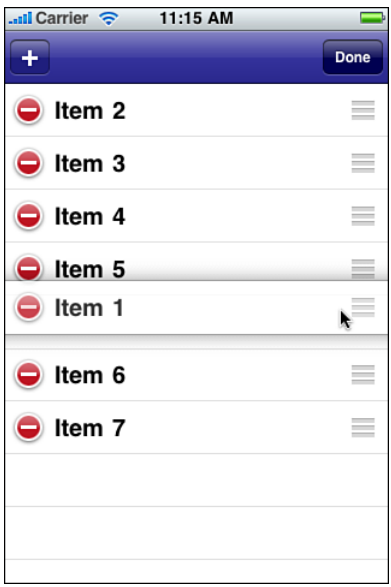


Figure 11-8 Reorder controls appear at the right of each cell during edit mode. They appear as three stacked gray lines. This screenshot shows Item 1 being dragged into place below Item 5.

Like swipe-to-delete, cell reordering support is contingent on the presence or absence of a single method. The `tableView:moveRowAtIndexPath:toIndexPath` method synchronizes your data source with the onscreen changes, similar to committing edits for cell deletion. Adding this method to Recipe 11-4 instantly enables reordering support.

The reordering method allows your code to update its data source. The following snippet moves the object corresponding to the cell's title to an updated location in the `items` mutable array:

```
-(void) tableView: (UITableView *) tableView
    moveRowAtIndexPath: (NSIndexPath *) oldPath
    toIndexPath:(NSIndexPath *) newPath
```

From the Library of Eric Berna

```
{
    if (oldPath.row == newPath.row) return;

    [[self.undoManager prepareWithInvocationTarget:self]
        tableView:self.tableView moveRowAtIndexPath:newPath
        toIndexPath:oldPath];

    NSString *item = [items objectAtIndex:oldPath.row];
    [items removeObjectAtIndex:oldPath.row];
    [items insertObject:item atIndex:newPath.row];

    [self setBarButtonItems];
    [self.tableView performSelector:@selector(reloadData)
        withObject:nil afterDelay:0.25f];
}
```

Sorting Tables Algorithmically

A table is its data source in every meaningful sense. When you sort the information that powers a table and then reload its data, you end up with a sorted table. The following snippet demonstrates how the data model could update itself with various sort types. To provide these sorts, this example uses the new blocks-supported `sortedArrayUsingComparator:` array method. As the user taps a segmented control, the `items` array replaces itself with a version using the selected sort. See Chapter 12, "A Taste of Core Data," for Core Data approaches that use sorting while fetching results from a persistent data store.

```
- (void) updateSort: (UISegmentedControl *) seg
{
    if (seg.selectedSegmentIndex == 0)
        items = [items sortedArrayUsingComparator:
            ^(id obj1, id obj2) {
                return [obj1 caseInsensitiveCompare:obj2];
        }];
    else if (seg.selectedSegmentIndex == 1)
        items = [items sortedArrayUsingComparator:
            ^(id obj1, id obj2) {
                return -1 * [obj1 caseInsensitiveCompare:obj2];
        }];
    else if (seg.selectedSegmentIndex == 2)
        items = [items sortedArrayUsingComparator:
            ^(id obj1, id obj2) {
                int l1 = [(NSString *)obj1 length];
                int l2 = [(NSString *)obj2 length];
                if (l1 == l2) return NSOrderedSame;
                return (l1 > l2) ?
```

From the Library of Eric Berna

```
                    NSOrderedDescending : NSOrderedAscending;
        }];

    [self.tableView reloadData];
}
```

Recipe: Working with Sections

Many iOS applications use sections as well as rows. Sections provide another level of structure to lists, grouping items together into logical units. The most commonly used section scheme is the alphabet, although you are certainly not limited to organizing your data this way. You can use any section scheme that makes sense for your application.

Figure 11-9 shows a table that uses sections to display grouped names. Each section presents a separate header (that is, "Crayon names starting with..."), and an index on the right offers quick access to each of the sections. Notice that there are no sections listed for K, Q, X, and Z in that index. You generally want to omit empty sections from the index.

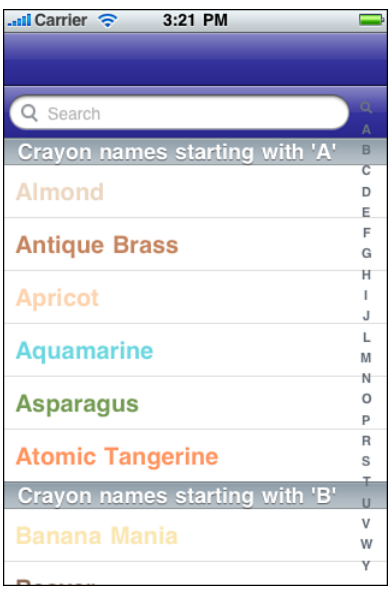


Figure 11-9 Sectioned tables let you present both headers and an index to better find information as quickly as possible.

From the Library of Eric Berna

Building Sections

When working with groups and sections, think two dimensionally. Section arrays let you store and access the members of data in a section-by-section structure. Implement this approach by creating an array of arrays. A section array can store one array for each section, which in turn contains the titles for each cell.

Predicates help you build sections from a list of strings. The following method alphabetically retrieves items from a flat array. The `beginswith` predicate matches each string that starts with the given letter.

```
- (NSArray *) itemsInSection: (NSInteger) section
{
    NSPredicate *predicate = [NSPredicate predicateWithFormat:
        @"SELF beginswith[cd] %@", [self firstLetter:section]];
    return [crayonColors.allKeys
        filteredArrayUsingPredicate:predicate];
}
```

Add these results iteratively to a mutable array to create a two-dimensional sectioned array from an initial flat list.

```
sectionArray = [NSMutableArray array];
for (int i = 0; i < 26; i++)
[sectionArray addObject:[self itemsInSection:i]];
```

To work, this particular implementation relies on two things: first, that the words are already sorted (each subsection adds the words in the order they're found in the array) and, second, that the sections match the words. Entries that start with punctuation or numbers won't work with this loop. You can trivially add an "other" section to take care of these cases, which this (simple) sample omits.

Although, as mentioned, alphabetic sections are useful and probably the most common grouping, you can use any kind of grouping structure you like. For example, you might group people by departments, gems by grades, or appointments by date. No matter what kind of grouping you choose, an array of arrays provides the table view data source that best matches sectioned tables.

From this initial startup, it's up to you to add or remove items using this two-dimensional structure. As you can easily see, creation is simple but maintenance gets tricky. Here's where Core Data really helps out. Instead of working with multileveled arrays, you can query your data store on any object field, sorting it as desired. Chapter 12 introduces using Core Data with tables. And as you will read in that chapter, it greatly simplifies matters. For now, this example will continue to use a simple array of arrays to introduce sections and their use.

From the Library of Eric Berna

Counting Sections and Rows

Sectioned tables require customizing two key data source methods:

- **`numberOfSectionsInTableView`**— This method specifies how many sections appear in your table, establishing the number of groups to display. When using a section array, as recommended here, return the number of items in the section array—that is, `sectionArray.count`. If the number of items is known in advance (26 in this case), you can hard-code that amount, but it's better to code more generally where possible.
- **`tableView:numberOfRowsInSection`**— This method is called with a section number. Specify how many rows appear in that section. With the recommended data structure, just return the count of items at the *n*th subarray:

```
[[sectionArray objectAtIndex:sectionNumber] count]
```

Returning Cells

Sectioned tables use both row and section information to find cell data. Earlier recipes in this chapter used a flat array with a row number index. Tables with sections must use the entire index path to locate both the section and row index for the data populating a cell. This method first retrieves the current items for the section and then pulls out the specific item by row:

```
- (UITableViewCell *)tableView:(UITableView *)aTableView
     cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Dequeue or create a cell
    UITableViewCellStyle style =  UITableViewCellStyleDefault;
    UITableViewCell *cell =
        [aTableView dequeueReusableCellWithIdentifier:@"BaseCell"];
    if (!cell)
        cell = [[UITableViewCell alloc]
            initWithStyle:style reuseIdentifier:@"BaseCell"] ;

    // Retrieve the crayon and its color
    NSArray *currentItems =
        [sectionArray objectAtIndex:indexPath.section];
    NSString *crayon = [currentItems objectAtIndex:indexPath.row];

    // Update the cell
    cell.textLabel.text = crayon;
    if (![crayon hasPrefix:@"White"])
        cell.textLabel.textColor = [crayonColors objectForKey:crayon];
    else
        cell.textLabel.textColor = [UIColor blackColor];
    return cell;
}
```

From the Library of Eric Berna

Creating Header Titles

It takes little work to add section headers to your grouped table. The optional `tableView:titleForHeaderInSection:` method supplies the titles for each section. It's passed an integer. In return, you supply a title. If your table does not contain any items in a given section or when you're only working with one section, return `nil`.

```
// Return the header title for a section
- (NSString *)tableView:(UITableView *)aTableView
    titleForHeaderInSection:(NSInteger)section
{
    if ([[sectionArray objectAtIndex:section] count] == 0) return nil;

    return [NSString stringWithFormat:
        @"Crayon names starting with '%@'", [self firstLetter:section]];
}
```

Creating a Section Index

Tables that implement `sectionIndexTitlesForTableView:` present the kind of index view that appears on the right of Figure 11-9. This method is called when the table view is created, and the array that is returned determines what items are displayed onscreen. Return `nil` to skip an index, as is done here for the search table. Apple recommends only adding section indices to plain table views—that is, table views created using the default plain style of `UITableViewStylePlain`.

```
// Return an array of section titles for index
- (NSArray *)sectionIndexTitlesForTableView:(UITableView *)aTableView
{
    NSMutableArray *indices = [NSMutableArray array];

    for (int i = 0; i < sectionArray.count; i++)
        if ([[sectionArray objectAtIndex:i] count])
            [indices addObject:[self firstLetter:i]];

    return indices;
}
```

Although this example uses single-letter titles, you are certainly not limited to those items. You can use words or, if you're willing to work out the Unicode equivalents, pictures, including emoji items (available to iPhone users in Japan) that are part of the iOS character library. Here's how you could add a small yellow smile:

```
[indices addObject:@"\ue057"];
```

From the Library of Eric Berna

Handling Section Mismatches

Indices move users along the table based on the user touch offset. As mentioned earlier in this section, this particular table does not display sections for K, Q, X, and Z. These missing letters can cause a mismatch between a user selection and the results displayed by the table.

To remedy this, implement the optional `tableView:sectionForSectionIndexTitle:` method. This method's role is to connect a section index title (that is, the one returned by the `sectionIndexTitlesForTableView:` method) with a section number. This overrides any order mismatches and provides an exact one-to-one match between a user index selection and the section displayed.

```
- (NSInteger)tableView:(UITableView *)tableView
    sectionForSectionIndexTitle:(NSString *)title
    atIndex:(NSInteger)index
{
    return [ALPHA rangeOfString:title].location;
}
```

Delegation with Sections

As with data source methods, the trick to implementing delegate methods in a data source table involves using the index path `section` and `row` properties. These properties provide the double access needed to find the correct section array and then the item within that array for this example. Recipe 11-5 shows how to update the navigation bar tint by recovering the color associated with a user tap on a section-based table.

Recipe 11-5 **Responding to User Touches in a Section-based Table**

```
- (void)tableView:(UITableView *)aTableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSArray *currentItems =
        [sectionArray objectAtIndex:indexPath.section];
    NSString *crayon = [currentItems objectAtIndex:indexPath.row];

    UIColor *crayonColor = [crayonColors objectForKey:crayon];
    self.navigationController.navigationBar.tintColor = crayonColor;
}
```

> **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 11 and open the project for this recipe.

From the Library of Eric Berna

Recipe: Searching Through a Table

Search display controllers allow users to filter a table's contents in real time, providing instant responsiveness to a user-driven query. It's a great feature that lets users interactively find what they're looking for, with the results updating as each new character is entered into the query field.

Searches are best built around predicates, allowing you to filter arrays to retrieve matching items with a simple method call. Here is how you might search through a flat array of strings to retrieve items that match the text from a search bar. The `[cd]` after `contains` refers to case- and diacritic-insensitive matching. Diacritics are small marks that accompany a letter, such as the dots of an umlaut (¨) or the tilde (~) above a Spanish *n*.

```
NSPredicate *predicate =
    [NSPredicate predicateWithFormat:@"SELF contains[cd] %@",
        searchBar.text];
filteredArray = [crayonColors.allKeys
    filteredArrayUsingPredicate:predicate];
```

The search bar in question should appear at the top of the table as its header view, as in Figure 11-10 (left). The same search bar is assigned to the search display controller, as shown in the following code snippet. Once users tap in the search box, the view shifts and the search bar moves up to the navigation bar area, as shown in Figure 11-10 (right). It remains there until the user taps Cancel, returning the user to the unfiltered table display.

```
self.tableView.tableHeaderView = searchBar;
searchController = [[UISearchDisplayController alloc]
    initWithSearchBar:searchBar contentsController:self];
```

Creating a Search Display Controller

Search display controllers help manage the display of data owned by another controller, in this case a standard `UITableViewController`. The search display controller presents a subset of that data, usually by filtering that data source through a predicate. You initialize a search display controller by providing it with a search bar and a contents controller.

Set up the search bar's text trait features as you would normally do but do not set a delegate. The search bar works with the search display controller without explicit delegation on your part.

When setting up the search display controller, make sure you set both its search results data source and delegate, as shown here. These point back to the primary table view controller subclass, which is where you'll adjust your normal data source and delegate methods to comply with the searchable table.

```
// Create a search bar
searchBar = [[UISearchBar alloc]
    initWithFrame:CGRectMake(0.0f, 0.0f, width, 44.0f)];
searchBar.tintColor = COOKBOOK_PURPLE_COLOR;
searchBar.autocorrectionType = UITextAutocorrectionTypeNo;
```

From the Library of Eric Berna

```
searchBar.autocapitalizationType = UITextAutocapitalizationTypeNone;
searchBar.keyboardType = UIKeyboardTypeAlphabet;
self.tableView.tableHeaderView = searchBar;

// Create the search display controller
searchController = [[UISearchDisplayController alloc]
    initWithSearchBar:searchBar contentsController:self];
searchController.searchResultsDataSource = self;
searchController.searchResultsDelegate = self;
```

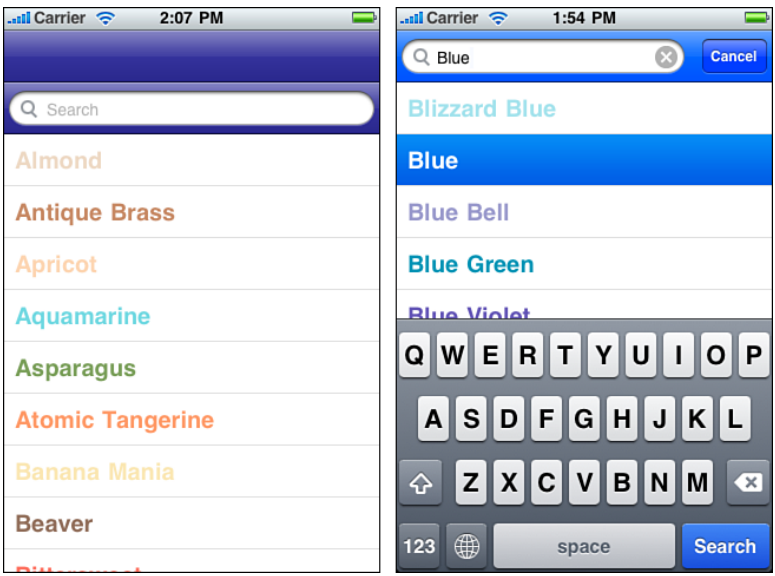


Figure 11-10 The user must scroll to the top of the table to initiate a search. The search bar appears as the first item in the table in its header view (left). Once the user taps within the search bar and makes it active, the search bar jumps into the navigation bar and presents a filtered list of items based on the search criteria (right).

Building the Searchable Data Source Methods

The number of items displayed in the table changes as users search. You report the correct number of rows for each. To detect whether the table view controller or the search display controller is currently in charge, compare the table view parameter against the table view controller's built-in `tableView` property. If it is the same, you're dealing with the normal table view. If it differs, that means the search display controller is in charge and is using its own table view. Adjust the row count accordingly.

From the Library of Eric Berna

```
// Return the number of rows per section
- (NSInteger)tableView:(UITableView *)aTableView
    numberOfRowsInSection:(NSInteger)section
{
    // Normal table
    if (aTableView == self.tableView)
        return [[sectionArray objectAtIndex:section] count];

    // This is always called before *using* the filtered array
    NSPredicate *predicate = [NSPredicate predicateWithFormat:
        @"SELF contains[cd] %@", searchBar.text];
    filteredArray = [crayonColors.allKeys
        filteredArrayUsingPredicate:predicate];
    return filteredArray.count;
}
```

Use a predicate to report the count of items that match the text in the search box. Predicates provide an extremely simple way to filter an array and return only those items that match a search string. The predicate used here performs a case-insensitive `contains` match. Each string that contains the text in the search field returns a positive match, allowing that string to remain part of the filtered array. Alternatively, you might want to use `beginswith` to avoid matching items that do not start with that text.

Use the table view check to determine what cells to present. The following method return cells retrieved from either the standard or the filtered set:

```
// Produce a cell for the given index path
- (UITableViewCell *)tableView:(UITableView *)aTableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Dequeue or create a cell
    UITableViewCellStyle style = UITableViewCellStyleDefault;
    UITableViewCell *cell =
        [aTableView dequeueReusableCellWithIdentifier:@"BaseCell"];
    if (!cell)
        cell = [[UITableViewCell alloc] initWithStyle:style
            reuseIdentifier:@"BaseCell"] ;

    // Retrieve the crayon and its color for the current table
    NSArray *currentItems =
        [sectionArray objectAtIndex:indexPath.section];
    NSArray *keyCollection = (aTableView == self.tableView) ?
        currentItems : FILTEREDKEYS;
    NSString *crayon = [keyCollection objectAtIndex:indexPath.row];

    cell.textLabel.text = crayon;
    if (![crayon hasPrefix:@"White"])
        cell.textLabel.textColor = [crayonColors objectForKey:crayon];
```

From the Library of Eric Berna

```
    else
        cell.textLabel.textColor = [UIColor blackColor];
    return cell;
}
```

Delegate Methods

Search awareness is not limited to data sources. Determining the context of a user tap is critical for providing the correct response in delegate methods. As with the previous data source methods, this delegate method compares the table view parameter sent with the callback to the built-in parameter. Based on this result, it chooses how to act, which in this case involves coloring both the search bar and the navigation bar with the currently selected color.

```
// Respond to user selections by updating tint colors
- (void)tableView:(UITableView *)aTableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    NSArray *currentItems =
        [sectionArray objectAtIndex:indexPath.section];
    NSArray *keyCollection = (aTableView == self.tableView) ?
        currentItems : FILTEREDKEYS;
    NSString *crayon = [keyCollection objectAtIndex:indexPath.row];

    UIColor *crayonColor = [crayonColors objectForKey:crayon];
    self.navigationController.navigationBar.tintColor = crayonColor;
    searchBar.tintColor = crayonColor;
}
```

Using a Search-Aware Index

Recipe 11-6 shows some of the other ways you'll want to adapt your sectioned table to accommodate search-ready tables. When you support search, the first item added to a table's section index should be the `UITableViewIndexSearch` constant. Intended for use only in table indices, and only as the first item in the index, this option adds the small magnifying glass icon that indicates that the table supports searches.

Use it to provide a quick jump to the beginning of the list. Update the `tableView:sectionForSectionIndexTitle:atIndex:` to catch user requests. The `scrollRectToVisible:animated:` call used in this recipe manually moves the search bar into place when a user taps on the magnifying glass. Otherwise, users would have to scroll back from section 0, which is the section associated with the letter A.

Add a call in `viewWillAppear:` to scroll the search bar offscreen when the view first loads. This allows your table to start with the bar hidden from sight, ready to be scrolled up to or jumped to as the user desires.

Finally, respond to cancelled searches by proactively clearing the search text from the bar.

From the Library of Eric Berna

Recipe 11-6 **Using Search Features**

```
// Add Search to the index
- (NSArray *)sectionIndexTitlesForTableView:(UITableView *)aTableView
{
    if (aTableView == self.tableView)  // regular table
    {
        NSMutableArray *indices =
           [NSMutableArray arrayWithObject:UITableViewIndexSearch];
        for (int i = 0; i < sectionArray.count; i++)
            if ([[sectionArray objectAtIndex:i] count])
                [indices addObject:[self firstLetter:i]];

        return indices;
    }
    else return nil; // search table
}

// Handle both the search index item and normal sections
- (NSInteger)tableView:(UITableView *)tableView
    sectionForSectionIndexTitle:(NSString *)title
    atIndex:(NSInteger)index
{
    if (title == UITableViewIndexSearch)
    {
        [self.tableView scrollRectToVisible:searchBar.frame animated:NO];
        return -1;
    }
    return [ALPHA rangeOfString:title].location;
}

// Handle the cancel button by resetting the search text
- (void)searchBarCancelButtonClicked:(UISearchBar *)aSearchBar
{
    [searchBar setText:@""];
}

// Upon appearing, scroll away the search bar
- (void) viewDidAppear:(BOOL)animated
{
    NSIndexPath *path = [NSIndexPath indexPathForRow:0 inSection:0];
    [self.tableView scrollToRowAtIndexPath:path
        atScrollPosition:UITableViewScrollPositionTop animated:NO];
}
```

From the Library of Eric Berna

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 11 and open the project for this recipe.

Customizing Headers and Footers

Sectioned table views are extremely customizable. You just read about using the `tableHeaderView` property to accommodate a `UISearchBar` search field. This, and the related `tableFooterView` property, can be assigned to any type of view, each with its own subviews. So you might add in labels, text fields, buttons, and other controls to extend the table's features.

Headers and footers do not stop with the full table. Each section offers a customizable header and footer view as well. You can alter heights or swap elements out for custom views. In Figure 11-11, the left image uses a larger-than-normal height and is created by implementing the optional `tableView:heightForHeaderInSection:` method. The right-hand image represents the use of custom views. The solid-colored header view, with its label and button subviews, is loaded from an .xib file and returned via the optional `tableView:viewForHeaderInSection:` method. Corresponding methods exist for footers as well as headers.

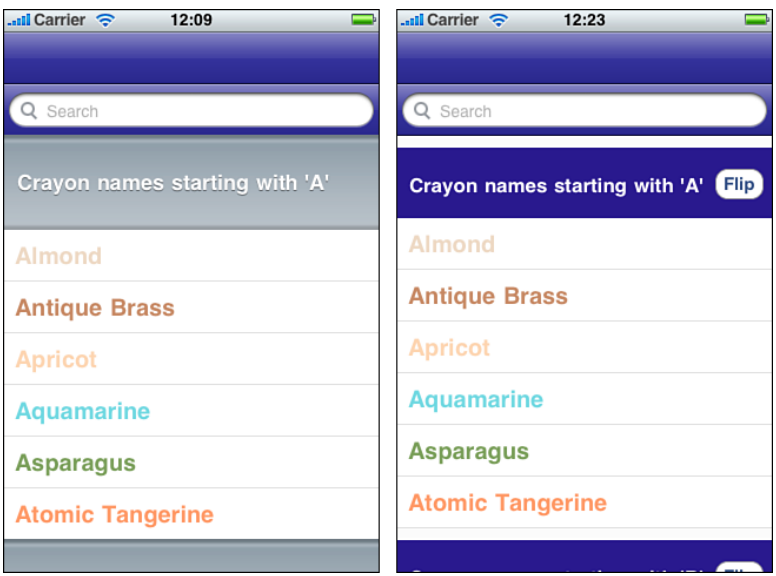


Figure 11-11 Table view delegate methods allow you to set the height and view of section headers and footers.

From the Library of Eric Berna

Here's how you might use these methods. The custom header is set at 50 points high and is loaded from an .xib file. This provides a trivial demonstration of a feature that is intrinsically much more powerful and more extensible than this simple example can express.

```
// Report the height for each section header
- (CGFloat)tableView:(UITableView *)tableView
    heightForHeaderInSection:(NSInteger)section
{
    return 50.0f;
}

- (UIView *)tableView:(UITableView *)tableView
    viewForHeaderInSection:(NSInteger)section
{
    // The nib contains a single object, the header view
    UIView *hView = [[[NSBundle mainBundle] loadNibNamed:@"HeaderView"
        owner:self options:nil] lastObject];

    return hView;
}
```

Recipe: Adding "Pull-to-Refresh" to Your Table

The App Store features many table-based applications that allow users to "pull down" in order to refresh table contents. I first encountered this behavior in the Echofon Twitter client, but it is by no means unique to that app. Pull-to-refresh is so intuitive that Apple really should have built it into its `UITableViewController` class. Since Apple did not, Recipe 11-7 shows you how to add it to your applications with just a few lines of code (see Figure 11-12).

Recipe 11-7 leverages the scroll view that underlies each table view. Instead of approaching this problem as a table challenge, it builds around scroll view delegate callbacks and content layout. It declares the `UIScrollViewDelegate` protocol and sets the table's `delegate` to the controller. This allows the control to catch scroll events.

The two events this recipe looks for are `scrollViewDidScroll:`, which allows you to detect whether the user has scrolled back far enough to trigger some reaction, and `scrollViewDidEndDecelerating:`, which is where you can implement some response to any trigger. In this example, when the user pulls back far enough (50 points in this case, adjust as desired), releasing touch triggers the controller to add new rows to the table. This trigger approach has two advantages. First, it lets you perform the refresh only once, at the conclusion of the interaction. Second, it lets you set the threshold that a user must commit to before allowing the refresh to proceed.

The recipe loads in a header view and adds it to the `tableView` as a child subview. The header's frame is offset by its height, so that the view ends at the y = 0 coordinate. This

From the Library of Eric Berna

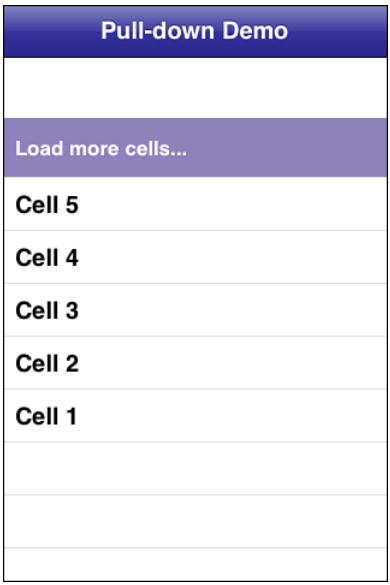


Figure 11-12 You can easily add a pull-to-refresh option to your tables by leveraging the underlying scroll view that powers tables.

leaves the header out of sight of the table. The only way to see the view is by pulling down on the table and using that built-in elasticity to reveal the otherwise hidden item.

Using pull-to-refresh allows your applications to delay performing expensive routines. For example, you might hold off fetching new information from the Internet or computing new table elements until the user triggers a request for those operations. Pull-to-refresh places your user in control of refresh operations and provides a great balance between information-on-demand and computational overhead.

Recipe 11-7 **Building Pull-to-Refresh into Your Tables**

```
@interface TestBedViewController : UITableViewController <UIScrollViewDelegate>
{
    int numberOfItems;
    BOOL addItemsTrigger;
}
@end

@implementation TestBedViewController
- (NSInteger)numberOfSectionsInTableView:(UITableView *)aTableView
{
    return 1; // a single section
}
```

From the Library of Eric Berna

```
- (NSInteger)tableView:(UITableView *)aTableView
    numberOfRowsInSection:(NSInteger)section
{
    return numberOfItems; // pulls increase this number
}

- (UITableViewCell *)tableView:(UITableView *)aTableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Dequeue or create a cell
    UITableViewCellStyle style =  UITableViewCellStyleDefault;
    UITableViewCell *cell = [aTableView
        dequeueReusableCellWithIdentifier:@"BaseCell"];
    if (!cell) cell = [[UITableViewCell alloc]
        initWithStyle:style reuseIdentifier:@"BaseCell"];

    cell.textLabel.text = [NSString stringWithFormat:@"Cell %d",
        numberOfItems - indexPath.row]; // reverse numbering
    return cell;
}

- (void)scrollViewDidEndDecelerating:(UIScrollView *)scrollView
{
    // Detect if the trigger has been set, if so add new items and reload
    if (addItemsTrigger)
    {
        numberOfItems += 2;
        [self.tableView reloadData];
    }

    // Reset the trigger
    addItemsTrigger = NO;
}

- (void) scrollViewDidScroll:(UIScrollView *)scrollView
{
    // Trigger the offset if the user has pulled back more than 50 points
    if (scrollView.contentOffset.y < -50.0f)
        addItemsTrigger = YES;
}

- (void) viewDidLoad
{
    self.title = @"Pull-down Demo";
    numberOfItems = 3; // start with a few seed cells
```

From the Library of Eric Berna

```
    // Add self as scroll view delegate to catch scroll events
    self.tableView.delegate = self;

    // Place the "Pull to Load" above the table
    UIView *pullView = [[[NSBundle mainBundle] loadNibNamed:
        @"HiddenHeaderView" owner:self options:nil] lastObject];
    pullView.frame =
        CGRectOffset(pullView.frame, 0.0f, -pullView.frame.size.height);
    [self.tableView addSubview:pullView];
}
@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 11 and open the project for this recipe.

Coding a Custom Group Table

If alphabetic section list tables are the M. C. Eschers of the iPhone table world, with each section block precisely fitting into the negative spaces provided by other sections in the list, then freeform group tables are the Marc Chagalls. Every bit is drawn as a freeform handcrafted work of art.

It's relatively easy to code up all the tables you've seen so far in this chapter once you've mastered the knack. Perfecting group table coding (usually called *preferences table* by devotees because that's the kind of table used in the Settings application) remains an illusion.

Building group tables in code is all about the collage. They're all about handcrafting a look, piece by piece. Figure 11-13 shows a simple preferences table that consists of two groups: a series of switches and a block with text. Creating a presentation like this in code involves a lot of detail work.

Creating Grouped Preferences Tables

There's nothing special involved in terms of laying out a new `UITableViewController` for a preferences table. You allocate it. You initialize it with the grouped table style. That's pretty much the end of it. It's the data source and delegate methods that provide the challenge. Here are the methods you'll need to define:

- **`numberOfSectionsInTableView:`**— All preferences tables contain groups of items. Each group is visually contained in a rounded rectangle. Return the number of groups you'll be defining as an integer.
- **`tableView:titleForHeaderInSection:`**— Add the titles for each section into this optional method. Return an `NSString` with the requested section name.

From the Library of Eric Berna

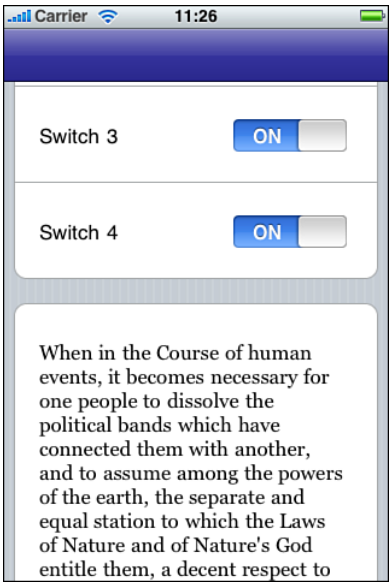


Figure 11-13 Hand-coded preferences tables must be laid out with each row and group specified through your data source methods.

- **`tableView:numberOfRowsInSection:`**—Each section may contain any number of cells. Have this method return an integer indicating the number of rows (that is, cells) for that group.
- **`tableView:heightForRowAtIndexPath:`**—Tables that use flexible row heights cost more in terms of computational intensity. If you need to use variable heights, implement this optional method to specify what those heights will be. Return the value by section and by row.
- **`tableView:cellForRowAtIndexPath:`**—This is the standard cell-for-row method you've seen throughout this chapter. What sets it apart is its implementation. Instead of using one kind of cell, you'll probably want to create different kinds of reusable cells (with different reuse tags) for each cell type. Make sure you manage your reuse queue carefully and use as many IB-integrated elements as possible.
- **`tableView:didSelectRowAtIndexPath:`**—You provide case-by-case reactions to cell selection in this optional delegate method depending on the cell type selected.

Note

The open-source llamasettings project at Google Code (http://llamasettings.googlecode.com) automatically produces grouped tables from property lists meant for iPhone settings bundles. It allows you to bring settings into your application without forcing your user to leave the app. The project can be freely added to commercial iOS SDK applications without licensing fees.

From the Library of Eric Berna

Recipe: Building a Multiwheel Table

Sometimes you'd like your users to pick from long lists or from several lists at once. That's where `UIPickerView` instances really excel. `UIPickerView` objects produce tables offering individually scrolling "wheels," as shown in Figure 11-14. Users interact with one or more wheels to build their selection.



Figure 11-14 `UIPickerView` instances enable users to select from independently scrolling wheels.

These tables, although superficially similar to standard `UITableView` instances, use distinct data and delegate protocols:

- **There is no `UIPickerViewController` class.** `UIPickerView` instances act as subviews to other views. They are not intended to be the central focus of an application view. You can build a `UIPickerView` instance onto another view like the action sheet shown in Figure 11-14.
- **Picker views use numbers not objects.** Components (that is to say, the wheels) are indexed by numbers and not by `NSIndexPath` instances. It's a more informal class than the `UITableView`.
- **The view height for pickers was historically static.** Until the 3.2 SDK you could not resize pickers the way you would a `UITableView` just by manipulating its frame. Standard iPhone portrait pickers are 320 by 216 points in size; landscape

From the Library of Eric Berna

pickers are 480 by 162. (These are the same dimensions used by the standard iPhone keyboard.) Pickers can now be resized, and it does look proper when performed, which is helpful when you're working on the iPad with its distinct proportions and screen size.

You can supply either titles or views via the data source. Picker views can handle both approaches.

Creating the `UIPickerView`

The joke used to go like this: "Use any frame size for your `UIPickerView` as long as your height is 216 points and your width is 320 points (portrait), or your height is 162 points and your width is 480 points (landscape)." That joke no longer holds true, although those remain the preferred dimensions for pickers, especially on iPhone and iPod touch devices.

When creating the picker, remember two key points. First, you want to enable the selection indicator. That is the blue bar that floats over the selected items. So set `showsSelectionIndicator` to `YES`. If you add the picker in Interface Builder, this is already set as the default.

Second, don't forget to assign the delegate and data source. Without this support, you cannot add data to the view, define its features, or respond to selection changes. Your primary view controller should implement the `UIPickerViewDelegate` and `UIPickerViewDataSource` protocols.

Implement three key data source methods for your `UIPickerView` to make it function properly at a minimum level. These methods are as follows:

- **`numberOfComponentsInPickerView`**—Return an integer, the number of columns.
- **`pickerView:numberOfRowsInComponent:`**—Return an integer, the maximum number of rows per wheel. These numbers do not need to be identical. You can have one wheel with many rows and another with very few.
- **`pickerView:titleForRow:forComponent`**—This method specifies the text used to label a row on a given component. Return an `NSString`. (Returning a view instead of a string is covered in the next section.)

In addition to these data source methods, you might want to supply one further delegate method. This method responds to user interactions via wheel selection:

- **`pickerView:didSelectRow:inComponent`**—Add any application-specific behavior to this method. If needed, you can query the `pickerView` to return the `selectedRowInComponent:` for any of the wheels in your view.

Recipe 11-8 creates the basic picker wheel shown in Figure 11-14. It presents a "combo-lock" picker, allowing users to enter a combination. Embedding the picker onto a `UIAlertSheet` instance allows the picker to slide in and out of view on the iPhone and display in a popover on the iPad.

From the Library of Eric Berna

Recipe 11-8 **Using a `UIPickerView` for Multicolumn Selection**

```
@interface TestBedViewController : UIViewController
    <UIPickerViewDelegate, UIActionSheetDelegate,
    UIPickerViewDataSource>
@end

@implementation TestBedViewController
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView
{
    return 3; // three wheels
}

- (NSInteger)pickerView:(UIPickerView *)pickerView
    numberOfRowsInComponent:(NSInteger)component
{
    return 20; // twenty items per row
}

- (NSString *)pickerView:(UIPickerView *)pickerView
    titleForRow:(NSInteger)row
    forComponent:(NSInteger)component
{
    return [NSString stringWithFormat:@"%@-%d",
        component == 1 ? @"R" : @"L", row];
}

- (void)actionSheet:(UIActionSheet *)actionSheet
     clickedButtonAtIndex:(NSInteger)buttonIndex
{
    UIPickerView *pickerView =
       (UIPickerView *)[actionSheet viewWithTag:101];
    self.title = [NSString stringWithFormat:@"L%d-R%d-L%d",
        [pickerView selectedRowInComponent:0],
        [pickerView selectedRowInComponent:1],
        [pickerView selectedRowInComponent:2]];
}

- (void)pickerView:(UIPickerView *)pickerView
    didSelectRow:(NSInteger)row inComponent:(NSInteger)component
{
    self.title = [NSString stringWithFormat:@"L%d-R%d-L%d",
        [pickerView selectedRowInComponent:0],
        [pickerView selectedRowInComponent:1],
        [pickerView selectedRowInComponent:2]];
}
```

From the Library of Eric Berna

```
- (void) action: (id) sender
{

    // Establish enough space for the picker
    NSString *title = @"\n\n\n\n\n\n\n\n\n\n\n\n";
    if (IS_IPHONE)
        title = UIDeviceOrientationIsLandscape(
            [UIDevice currentDevice].orientation) ?
                @"\n\n\n\n\n\n\n\n\n" : @"\n\n\n\n\n\n\n\n\n\n\n\n" ;

    // Create the base action sheet
    UIActionSheet *actionSheet = [[UIActionSheet alloc]
        initWithTitle:title delegate:self cancelButtonTitle:nil
        destructiveButtonTitle:nil otherButtonTitles:@"Set Combo", nil];

    // Display the action sheet
    if (IS_IPHONE)
        [actionSheet showInView:self.view];
    else
        [actionSheet showFromBarButtonItem:sender animated:NO];

    // Build the picker
    UIPickerView *pickerView = [[UIPickerView alloc] init];
    pickerView.tag = 101;
    pickerView.delegate = self;
    pickerView.dataSource = self;
    pickerView.showsSelectionIndicator = YES;

    // If working with an iPad, adjust the frames as needed
    if (!IS_IPHONE)
    {
        pickerView.frame = CGRectMake(0.0f, 0.0f, 272.0f, 216.0f);
        CGPoint center = actionSheet.center;
        actionSheet.frame = CGRectMake(0.0f, 0.0f, 272.0f, 253.0f);
        actionSheet.center = center;
    }

    // Embed the picker
    [actionSheet addSubview:pickerView];
}
@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 11 and open the project for this recipe.

From the Library of Eric Berna

Recipe: Using a View-based Picker

Picker views work just as well with views as they do with titles. Figure 11-15 shows a picker view that displays card suits. These images are returned by the `pickerView:viewForRow:forComponent:reusingView:` data source method. You can use any kind of view you like, including labels, sliders, buttons, and so forth.

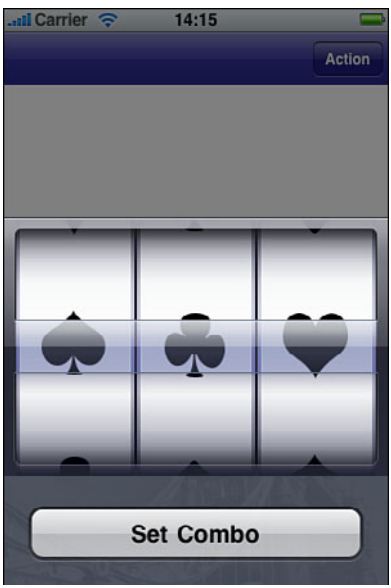


Figure 11-15 This `UIPickerView` presents a series of card suit images, allowing users to pick a combination of three items.

Picker views use a basic view-reuse scheme, caching the views supplied to it for possible reuse. When the final parameter for this callback method is not `nil`, you can reuse that view by updating its settings or contents. Check for the view and allocate a new one only if one has not been supplied.

The height need not match the actual view. Implement `pickerView:rowHeightForComponent:` to set the row height used by each component. Recipe 11-9 uses a row height of 120 points, providing plenty of room for each image and laying the groundwork for the illusion that the picker could be continuous rather than having a start and ending point as Recipe 11-8 did.

Notice the high number of components, namely one million. The reason for this high number lies in a desire to emulate real cylinders. Normally, picker views have a first element and a last, and that's where they end. This recipe takes another approach, asking "What if the components were actual cylinders, so the last element was connected to the first?"

From the Library of Eric Berna

To emulate this, the picker uses a far higher number of components than any user will ever be able to access. It initializes the picker to the middle of that number by calling `selectRow:inComponent:Animated:`. Each component "row" is derived by the modulo of the actual reported row and the number of individual elements to display (in this case, `% 4`). Although the code knows that the picker actually has a million rows per wheel, the user experience offers a cylindrical wheel of just four rows.

Recipe 11-9 **Creating the Illusion of a Repeating Cylinder**

```
- (NSInteger)pickerView:(UIPickerView *)pickerView
    numberOfRowsInComponent:(NSInteger)component
{
    // Return an insanely high number for the rows per wheel
    return 1000000;
}

- (CGFloat)pickerView:(UIPickerView *)pickerView
    rowHeightForComponent:(NSInteger)component
{
    // Produce a row height to match the art
    return 120.0f;
}

- (UIView *)pickerView:(UIPickerView *)pickerView
    viewForRow:(NSInteger)row forComponent:(NSInteger)component
    reusingView:(UIView *)view
{
    // Create a new view where needed, adding the art
    UIImageView *imageView;
    imageView = view ? (UIImageView *) view : [[UIImageView alloc]
        initWithFrame:CGRectMake(0.0f, 0.0f, 60.0f, 60.0f)];
    NSArray *names = [NSArray arrayWithObjects:@"club.png",
        @"diamond.png", @"heart.png", @"spade.png", nil];
    imageView.image = [UIImage imageNamed:
        [names objectAtIndex:(row % 4)]];
    return imageView;
}

- (void)actionSheet:(UIActionSheet *)actionSheet
    clickedButtonAtIndex:(NSInteger)buttonIndex
{
    // Set the title to match the current wheel selections
    UIPickerView *pickerView = (UIPickerView *)[actionSheet
        viewWithTag:101];
    NSArray *names = [NSArray arrayWithObjects:
        @"C", @"D", @"H", @"S", nil];
    self.title = [NSString stringWithFormat:@"%@•%@•%@",
```

From the Library of Eric Berna

```
        [names objectAtIndex:([pickerView
        selectedRowInComponent:0] % 4)],
        [names objectAtIndex:([pickerView
        selectedRowInComponent:1] % 4)],
        [names objectAtIndex:([pickerView
        selectedRowInComponent:2] % 4)]];
    [actionSheet release];
}
```

> **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 11 and open the project for this recipe.

Recipe: Using the `UIDatePicker`

When you want to ask your user to enter date information, Apple supplies a tidy subclass of `UIPickerView` to handle several kinds of time entry. Figure 11-16 shows the four built-in styles of `UIDatePicker`s you can choose from. These include selecting a time, selecting a date, selecting a combination of the two, and a countdown timer. Recipe 11-10 demonstrates all these styles.

Creating the Date Picker

Lay out a date picker exactly as you would a `UIPickerView`. The geometry is identical. After that, things get much, much easier. You need not set a delegate or define data source methods. You do not have to declare any protocols. Just assign a date picker mode. Choose from `UIDatePickerModeTime`, `UIDatePickerModeDate`, `UIDatePickerModeDateAndTime`, and `UIDatePickerModeCountDownTimer`.

Optionally, add a target for when the selection changes (`UIControlEventValueChanged`) and create the callback method for the target-action pair. Here are a few properties you'll want to take advantage of in the `UIDatePicker` class:

- **`date`**—Set the date property to initialize the picker or to retrieve the information set by the user as he or she manipulates the wheels.
- **`maximumDate` and `minimumDate`**—These properties set the bounds for date and time picking. Assign each one a standard `NSDate`. With these, you can constrain your user to pick a date from next year rather than just enter a date and then check whether it falls within an accepted time frame.
- **`minuteInterval`**—Sometimes you want to use 5-, 10-, 15-, or 30-minute intervals on your selections, such as for applications used to set appointments. Use the `minuteInterval` property to specify that value. Whatever number you pass, it has to be evenly divisible into 60.

From the Library of Eric Berna

- **`countDownDuration`**—Use this property to set the maximum available value for a countdown timer.You can go as high as 23 hours and 59 minutes (that is, 86,399 seconds).



Figure 11-16 The iPhone offers four stock date picker models. Use the `datePickerMode` property to select the picker you want to use in your application.

Recipe 11-10 Using the `UIDatePicker` to Select Dates and Times

```
- (void) update: (id) sender
{
    // Update the picker mode and reset the date to now
    [datePicker setDate:[NSDate date]];
    datePicker.datePickerMode = seg.selectedSegmentIndex;
}
```

From the Library of Eric Berna

```
- (void) action: (id) sender
{
    // Set the output format based on the current picker choice
    NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
    switch (seg.selectedSegmentIndex)
    {
        case 0:
            formatter.dateFormat = @"h:mm a";
            break;
        case 1:
            formatter.dateFormat = @"dd MMMM yyyy";
            break;
        case 2:
            formatter.dateFormat = @"MM/dd/YY h:mm a";
            break;
        case 3:
            formatter.dateFormat = @"HH:mm";
            break;
        default:
            break;
    }

    NSString *timestamp = [formatter stringFromDate:datePicker.date];
    NSLog(@"%@", timestamp);
}

- (void) loadView
{
    [super loadView];
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Action", @selector(action:));

    seg = [[UISegmentedControl alloc] initWithItems:
        [@"Time Date DT Count" componentsSeparatedByString:@" "]];
    seg.segmentedControlStyle = UISegmentedControlStyleBar;
    seg.selectedSegmentIndex = 0;
    [seg addTarget:self action:@selector(update:)
        forControlEvents:UIControlEventValueChanged];
    self.navigationItem.titleView = seg;

    datePicker = [[UIDatePicker alloc] init];
    [self.view addSubview:datePicker];
}
```

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 11 and open the project for this recipe.

From the Library of Eric Berna

One More Thing: Formatting Dates

Although the `NSDateFormatter` class has evolved a great deal from its early days and now offers highly customizable elements that can be localized for various calendars and cultures, it helps to have a quick reference on hand for the most common date and time formats. Table 11-1 provides that reference, listing the most commonly used default codes. These codes are like the ones used in Recipe 11-10 to format the results from the date picker's `date` property for the midscreen label. Listing 11-1 uses the formats from Table 11-1 to create an `NSString` utility that converts a string into a date using a format of Month-Day-Year (for example, `@"05-22-1934"`).

Table 11-1 **Default Format Codes for the `NSDateFormatter` Class**

Type	Code	Notes
Day of month	`d`	1 to 31, no leading zeros
	`dd`	01 to 31, uses leading zeros
Month	`M`	1 to 12, numeric month value, no leading zeros
	`L`	
	`MM`	01 to 12, numeric month value, leading zeros
	`LL`	
	`MMM`	Jan to Dec, three-letter month abbreviation
	`LLL`	
	`MMMM`	January to December, full month name
	`LLLL`	
Year	`y`	Four-digit year (e.g., 2009)
	`u`	
	`yy`	Two-digit year (e.g., 09)
Hour	`h`	1 to 12, no leading zeros
	`K`	
	`hh`	01 to 12, leading zeros
	`KK`	
	`H`	0 to 23, 24-hour clock, no leading zeros
	`k`	
	`HH`	00 to 23, 24-hour clock, leading zeros
	`kk`	
Minutes	`m`	0 to 59, no leading zeros
	`mm`	00 to 59, leading zeros
Seconds	`s`	0 to 59, no leading zeros
	`ss`	0 to 59, leading zeros
AM/PM	`a`	

From the Library of Eric Berna

Table 11-1 **Default Format Codes for the `NSDateFormatter` Class**

Type	Code	Notes
Day of Week	`ccc`	Sun through Sat, three-letter abbreviations
	`EEE`	
	`cccc`	Sunday through Saturday, full names
	`EEEE`	
	`c`	Ordinal day of week (0–7)
	`e`	
	`cc`	Ordinal day of week (00–07)
	`ee`	
Week of Month	`F`	1–5, no leading zeros
	`FF`	01–05, leading zeros
Day of Year	`D`	1–366, no leading zeros
	`DD`	01–366, one leading zero
	`DDD`	001–366, two leading zeros
Week of Year	`w`	1–52, no leading zeros
	`ww`	01-51, one leading zero
Millisecond of Day	`A`	0–86399999, no padding
Astronomical Julian Day Number	`g`	Number of days since 1 January 4713 BCE
Era	`G`	BC, AD—Christian year notation
	`GGGG`	Before Christ, Anno Domini—Christian year notation
Quarter	`q`	1–4, quarter of year
	`Q`	
	`qq`	01–04, one leading zero
	`QQ`	
	`qqq`	Q1, Q2, Q3, Q4
	`QQQ`	
	`qqqq`	1st quarter, 2nd quarter, 3rd quarter, 4th quarter
	`QQQQ`	
Time Zone	`v`	Two-letter time zone (e.g., MT)
	`V`	Three-letter time zone (e.g., MDT)
	`z`	
	`vv`	RFC 822 time zone offset from Greenwich Mean Time (e.g., GMT-06:00)
	`VV`	
	`ZZZZ`	
	`vvvv`	Time zone name (e.g., Mountain Time)

From the Library of Eric Berna

Table 11-1 **Default Format Codes for the `NSDateFormatter` Class**

Type	Code	Notes
	`vvvv`	Time zone location (e.g., United States [Denver])
	`zzzz`	Full time zone name (e.g., Mountain Daylight Time)
	`Z`	GMT offset (e.g., -0600)
Other characters	`:`	Colon, hyphen, slash
	`-`	
	`/`	

Listing 11-1 **Using Date Formats to Convert a String to a Date**

```
@implementation (NSString-DateUtility)
- (NSDate *) date
{
    // Return a date from a string
    NSDateFormatter *formatter =
        [[NSDateFormatter alloc] init];
    formatter.dateFormat = @"MM-dd-yyyy";
    NSDate *date = [formatter dateFromString:aString];
    return date;
}
@end
```

Summary

This chapter introduced iOS tables from the simple to the complex. You saw all the basic iOS table features—from simple tables, to edits, to reordering and undo. You also learned about a variety of advanced elements—from custom XIB-based cells, to indexed alphabetic listings, to picker views. The skills covered in this chapter enable you to build a wealth of table-based applications for the iPhone, iPad, and iPod touch. Here are some key points to take away from this chapter:

- When it comes to understanding tables, make sure you know the difference between data sources and delegate methods. Data sources fill up your tables with meaningful cells. Delegate methods respond to user interactions.
- `UITableViewControllers` simplify applications built around a central `UITableView`. Do not hesitate to use `UITableView` instances directly, however, if your application requires them—especially in popovers or with split view controllers. Just make sure to explicitly support the `UITableViewDelegate` and `UITableViewDataSource` protocols when needed.

From the Library of Eric Berna

- Index controls provide a great way to navigate quickly through large ordered lists. Take advantage of their power when working with tables that would otherwise become unnavigable. Stylistically, it's best to avoid index controls when working with grouped tables.
- Dive into edits. Giving the user control over the table data is easy to do, and your code can be reused over many projects. Don't hesitate to design for undo support from the start. Even if you think you may not need undo at first, you may change your mind over time.
- It's easy to convert flat tables into sectioned ones. Don't hesitate to use the predicate approach introduced in this chapter to create sections from simple arrays. Sectioned tables allow you to present data in a more structured fashion, with index support and easy search integration.
- Date pickers are highly specialized and very good at what they do: soliciting your users for dates and times. Picker views provide a less-specialized solution but require more work on your end to bring them to life.
- This chapter introduced the `NSPredicate` class. This class provides flexible and powerful solutions that extend well beyond tables and are explored further in Chapter 12.

From the Library of Eric Berna

This page intentionally left blank

From the Library of Eric Berna

12

A Taste of Core Data

iOS's Core Data framework provides persistent data solutions. It offers managed data stores that can be queried and updated from your application. With Core Data, you gain a Cocoa Touch–based object interface that brings relational data management out from SQL queries and into the Objective-C world of iOS development. This chapter introduces Core Data. It provides just enough recipes to give you a taste of the technology, offering a jumping-off point for further Core Data learning. By the time you finish reading through this chapter, you'll have seen Core Data in action. You'll gain an overview of the technology and will have worked through several common Core Data scenarios. This chapter cannot provide a full review of Core Data—there are standalone books for that—but it does offer a jumping-off point for those who want to integrate this technology into their iOS applications.

Introducing Core Data

Core Data simplifies the way your applications create and use managed objects. Until the 3.*x* SDK, all data management and SQL access were left to fairly low-level libraries. It wasn't pretty, and it wasn't easy to use. Since then, Core Data has joined the Cocoa Touch framework family, bringing powerful data management solutions to iOS. Core Data provides a flexible object management infrastructure. It offers tools for working with persistent data stores, generating solutions for the complete object life cycle.

Core Data lives in the Model portion of the Model-View-Controller paradigm. Core Data understands that application-specific data must be defined and controlled outside the application's GUI. Because of that, the application delegate, view controller instances, and custom model classes provide the most natural homes for Core Data functionality. Where you place the ownership depends on how you'll use the data.

As a rule, your application delegate usually owns any shared database that gets used throughout the application. Simpler applications may get by with a single view controller that manages the same data access. The key lies in understanding that the owner controls all data access—reading, writing, and updating. Any part of your application that works with Core Data must coordinate with that owner.

From the Library of Eric Berna

While it's agreed that the data model portion of the application exists separately from its interface, Apple understands that data does not exist in a vacuum. The iOS SDK integrates seamlessly with `UITableView` instances. Cocoa Touch's fetched-results controller class was designed and built with tables in mind. It offers useful properties and methods that support table integration. You see this integration in action via recipes later in this chapter.

Creating and Editing Model Files

Model files define how Core Data objects are structured. Each project that links against the Core Data framework includes one or more model files. These .xcdatamodel files define the objects, their attributes, and their relationships.

Each object may own any number of properties, which are called "attributes." Attribute types include strings, dates, numbers, and data. Each object can also have relationships, which are links between one object and another. These relationships can be single, using a one-to-one relationship, or they can be multiple, using a one-to-many relationship. In addition, those relationships can be one-way, or they can be reciprocal, providing an inverse relationship.

Define your model in Xcode by laying out a new data model file. Some iOS templates allow you to include Core Data as part of the project. Otherwise, you can create these Xcode model files by selecting File > New > New File > iOS > Core Data > Data Model > Next. Enter a name for your new file, check Add to targets: for your project, and click Save. Xcode creates and then adds the new model file to your project. If Xcode has not done so already, you may want to then drag it into your project's Resources group.

Click the .xcdatamodel file to open it in the editor window, as shown in Figure 12-1. Add new object entities (basically "classes") in the left list at the left column in the editor window; define and view attributes and relationships (essentially "instance variables") in the right column of the editor window. The Core Data data model inspector appears in the Utility pane to the right of the editor. You may toggle between a table view and the object graph shown in Figure 12-1 by tapping the buttons at the bottom-right of the Editor pane. The object graph offers a grid-based visual presentation of the entities you have defined.

Tap the Add Entity button at the bottom-left of the Editor pane to add a new entity (a class definition that acts as an object skeleton) to your model. By default, all new entities are instantiated at runtime as instances of the `NSManagedObject` class. Edit the word "Entity" to give your new object a name (for example, Person).

With the entity selected, you can add attributes. Tap the Add Attribute button at the bottom-right of the Editor pane. You can tap-and-hold this button to choose between Add Attribute, Add Relationship, and Add Fetched Property. Each attribute has a name and is typed, just as you would define an instance variable. Relationships are pointers to other objects. You can define a single pointer for a one-to-one relation (the single manager for a department) or a set for a one-to-many relation (all the members of a department). Take note of the inspector at the top-right. Here, you can edit an object's name

From the Library of Eric Berna

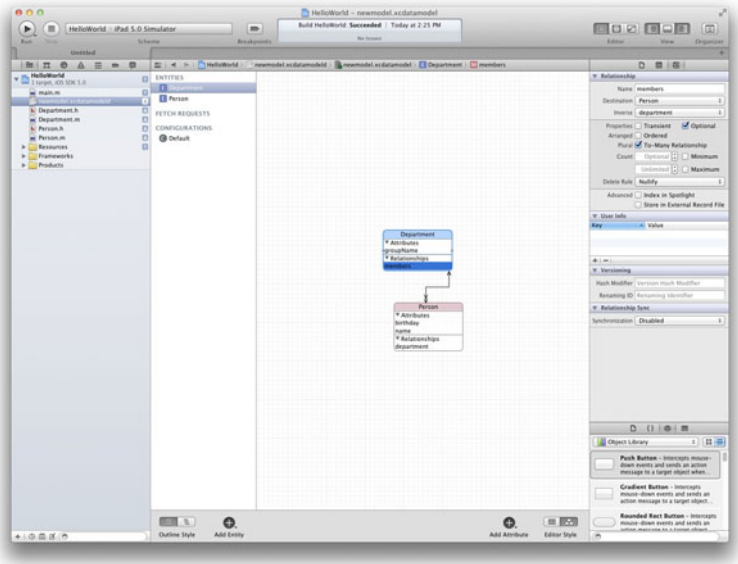


Figure 12-1 Xcode's editor allows you to build managed object definitions for your Core Data applications.

and type, set its "to-many" relationship option, define its default value, limit its legal value range, and more.

Both the entity list and the object graph show the entities you have defined. The graph provides arrows that represent the relationships between the various kinds of entities in your project. In this model, every person belongs to a department. Each department has a manager (a one-to-one relationship) and any number of members (a one-to-many relationship).

To build this model, create a Person entity and add two attributes: birthday and name. Set the birthday type to date and the name type to string. Create a Department entity. Add a single attribute, `groupName`, as a string type.

Switch from the list view to the graph view by clicking the right of the two Editor Style buttons at the bottom of the pane. Hover the mouse over the Person entity proxy and press the Control key. The cursor switches to a crosshair. With the crosshair, drag from Person > Relationships to Department and release the mouse. A two-way arrowed line appears.

If you quickly switch from the graph to the table editor, you'll see that a new relationship called "newRelationship" has been added to both the Department and the Person.

Return to the graph. Click Department > Relationships > newRelationship. Open the Data Model inspector in the Utilities pane. Edit the name to "members" and check Relationship > Plural > To-Many Relationship. Click Person > Relationships > newRelationship. Edit the name to "department" and then save your changes.

From the Library of Eric Berna

Generating Class Files

Once you have laid out and saved your model, you need to generate files for each entity. Select each entity and choose Editor > Create Managed Object Subclass. Save into your project folder; select the group you want to add the classes to. Click Create. Xcode automatically builds your class files for you.

For example, here is the Department header for the project shown in Figure 12-1:

```
#import <CoreData/CoreData.h>
#import <Foundation/Foundation.h>

@interface Department : NSManagedObject

@property (nonatomic, retain) NSString * groupName;
@property (nonatomic, retain) NSSet* members;
@end

@interface Department (CoreDataGeneratedAccessors)

- (void)addMembersObject:(NSManagedObject *)value;
- (void)removeMembersObject:(NSManagedObject *)value;
- (void)addMembers:(NSSet *)values;
- (void)removeMembers:(NSSet *)values;

@end
```

You can see that the group name is a string and that the members' one-to-many relationship is defined as a set.

Although this looks like a standard Objective-C class header file, importantly there are few actual implementation details you have to work with. Core Data takes care of most of those for you using the `@dynamic` compiler keywords. In addition, Xcode adds boilerplate property methods, which you are free to either modify or remove according to your needs.

```
@implementation Department


@dynamic groupName;
@dynamic members;

@end
```

An important reason for generating Core Data files lies in their ability to add new behavior and transient attributes—that is, attributes not saved in the persistent store. For example, you might create a `fullName` attribute returning a name built from a person's `firstName` and `lastName`. Plus, there's nothing to stop you from using a managed object class like any other class—that is, using and manipulating all kinds of data. You can bundle

From the Library of Eric Berna

any kind of Objective-C behavior into a managed object instance by editing its implementation file. You can add instance and class methods as needed.

Creating a Core Data Context

After designing the managed object model, it's time to build code that accesses a data file. To work with Core Data you need to programmatically create a managed object context. A context performs all the access and update functions needed to coordinate your model with a file.

The following method initializes the context for an application. This work is all done for you when you use a prebuilt Core Data template; this method shows how to do the same work by hand. It starts by reading in all the model files from the application bundle and merging them into a central model. It then initializes a persistent store coordinator—here, using SQLite (`NSSQLiteStoreType`) to store the data. This coordinator provides low-level file access using the central model. You supply a URL that points to the file you want to use to store the model's data. Finally, this method initializes a new context using the coordinator and stores it as a strong (retained) instance variable:

```
- (void) initCoreData
{
    NSError *error;

    // Path to data file.
    NSString *path = [NSHomeDirectory()
        stringByAppendingPathComponent:@"Documents/data.db"];
    NSURL *url = [NSURL fileURLWithPath:path];

    // Init the model, coordinator, context
    NSManagedObjectModel *managedObjectModel =
        [NSManagedObjectModel mergedModelFromBundles:nil];
    NSPersistentStoreCoordinator *persistentStoreCoordinator =
        [[NSPersistentStoreCoordinator alloc]
        initWithManagedObjectModel:managedObjectModel];
    if (![persistentStoreCoordinator
        addPersistentStoreWithType:NSSQLiteStoreType
        configuration:nil URL:url options:nil error:&error])
        NSLog(@"Error: %@", [error localizedFailureReason]);
    else
    {
        context = [[NSManagedObjectContext alloc] init];
        [context setPersistentStoreCoordinator:
            persistentStoreCoordinator];
    }
}
```

From the Library of Eric Berna

It's important to maintain a context instance that you can refer to, whether from a view controller (for a simple Core Data application) or from your application delegate (for more complex applications). The context is used for all read, search, and update operations in your application.

Adding Objects

Create new objects by inserting entities into your managed context. The following snippet builds three new items: a department and two people. After inserting the object, which returns the new instance, you set the managed object's properties (its attributes and relationships) by assignment. Each person belongs to a department. Each department has a set of members and one manager. This code reflects the design built in Figure 12-1. You do not have to explicitly set the department's members. The inverse relationship takes care of that for you, adding the members into the department when you set the person's department attribute.

```
- (void) addObjects
{
    // Insert objects for department and several people,
    // setting their properties

    Department *department =
        (Department *)[NSEntityDescription
            insertNewObjectForEntityForName:@"Department"
            inManagedObjectContext:context];
    department.groupName = @"Office of Personnel Management";

    for (NSString *name in [@"John Smith*Jane Doe*Fred Wilkins"
        componentsSeparatedByString:@"*"])
    {
        Person *person = (Person *)[NSEntityDescription
            insertNewObjectForEntityForName:@"Person"
            inManagedObjectContext:context];
        person.name = name;
        person.birthday = [NSDate date];
        person.department = department;
    }

    // Save the data
    NSError *error;
    if (![context save:&error])
      NSLog(@"Error: %@", [error localizedFailureReason]);
}
```

No changes to the persistent store file take effect until you save. A save operation brings the database file up to date with the model stored in memory. The single save

From the Library of Eric Berna

request in this code tells the context to synchronize its state with the persistent store, writing out all changes to the database file.

If you run this code in the simulator, you can easily inspect the .sqlite file that's created. Navigate to the simulator folder (~/Library/Application Support/iPhone Simulator/*Firmware*/User/Applications, where *Firmware* is the current firmware release—for example, 5.0) and into the folder for the application itself. Stored in the Library folder (depending on the URL used to create the persistent store), a .sqlite file contains the database representation you've created.

Use the command-line sqlite3 utility to inspect the contents by performing a `.dump` operation. Here, you see the two SQL table definitions (department and person) that store the information for each object plus the insert commands used to store the instances built in your code. Although you are thoroughly cautioned against directly manipulating the Core Data store with sqlite3, it offers a valuable insight into what's going on under the Core Data hood.

```
% sqlite3 data.db
SQLite version 3.7.5
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .dump
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE ZDEPARTMENT ( Z_PK INTEGER PRIMARY KEY,
Z_ENT INTEGER, Z_OPT INTEGER, ZGROUPNAME VARCHAR );
INSERT INTO "ZDEPARTMENT" VALUES(1,1,1,
'Office of Personnel Management');
CREATE TABLE ZPERSON ( Z_PK INTEGER PRIMARY KEY, Z_ENT INTEGER,
Z_OPT INTEGER, ZDEPARTMENT INTEGER, ZBIRTHDAY TIMESTAMP,
ZNAME VARCHAR );
INSERT INTO "ZPERSON" VALUES(2,2,1,1,333663713.877964,'Fred Wilkins');
INSERT INTO "ZPERSON" VALUES(3,2,1,1,333663713.87454,'John Smith');
INSERT INTO "ZPERSON" VALUES(5,2,1,1,333663713.877087,'Jane Doe');
CREATE TABLE Z_PRIMARYKEY (Z_ENT INTEGER PRIMARY KEY,
Z_NAME VARCHAR, Z_SUPER INTEGER, Z_MAX INTEGER);
INSERT INTO "Z_PRIMARYKEY" VALUES(1,'Department',0,1);
INSERT INTO "Z_PRIMARYKEY" VALUES(2,'Person',0,5);
CREATE TABLE Z_METADATA (Z_VERSION INTEGER PRIMARY KEY,
Z_UUID VARCHAR(255), Z_PLIST BLOB);
INSERT INTO "Z_METADATA" VALUES(1,'DD1D5AAB-F5EC-40A2-BCA5-
83A54411422C',X'62706C6973743030D60102030405060708090A0F10115F101E4E5353746F72654D6F6
4656C56657273696F6E4964656E746966696572735F101D4E535065727369737465 6E63654672616D65
776F726B56657273696F6E5F10194E5353746F72654D6F64656C56657273696F6E4861736865735B4E5
353746F7265547970655F10125F4E534175746F56616375756D4C6576656C5F10204E5353746F72654D
6F64656C56657273696F6E4861736865735656572736963696F6EA1084011016ED20B0C0D0E56506572736F6
E5A4465706172746D656E744F1020DE27A38E814A2A5F72418573563732F83CBC1CACAADF39FA559420
B155E5A9734F1020B60AD00AD230B4BF739A6F856B8B72BAC945B4A5C63BF9DA6F6F7A8AFA40D321565
```

From the Library of Eric Berna

```
3514C69746551321003000800150036005600720007E009300B600B800B900BC00C100C800D300F60119
012001220000000000000201000000000000001200000000000000000000000000000124');
CREATE INDEX ZPERSON_ZDEPARTMENT_INDEX ON ZPERSON (ZDEPARTMENT);
COMMIT;
sqlite> .quit
%
```

Querying the Database

Retrieve objects from the database by performing fetch requests. A fetch request describes your search criteria. It's passed through and used to initialize a results object that contains a pointer to the managed objects that meet those criteria. The results controller executes the fetch before passing back the array of managed object results.

The following `fetchPeople` method creates a new request, setting its entity type to Person. This search looks for `Person` objects in the shared managed store. Each request must contain at least one sort descriptor. For this example, the search returns a list of `Person` records sorted in ascending order by their name field. Although you can produce more complicated queries, this example shows the simplest "please return all managed items of a given type" request. The results are saved locally into a strong (retained) instance variable called `fetchedResultsController`.

```
- (void) fetchPeople
{
    // Create a basic fetch request
    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    [fetchRequest setEntity:
        [NSEntityDescription entityForName:@"Person"
            inManagedObjectContext:context]];

    // Add a sort descriptor
    NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
        initWithKey:@"name" ascending:YES selector:nil];
    NSArray *descriptors = [NSArray arrayWithObject:sortDescriptor];
    [fetchRequest setSortDescriptors:descriptors];

    // Init the fetched results controller
    NSError __autoreleasing *error;
    fetchedResultsController =
        [[NSFetchedResultsController alloc]
            initWithFetchRequest:fetchRequest
            managedObjectContext:context
            sectionNameKeyPath:nil cacheName:@"Root"];
    if (![fetchedResultsController performFetch:&error])
        NSLog(@"Error: %@", [error localizedFailureReason]);
}
```

From the Library of Eric Berna

The fetched results controller handles the results returned from a Core Data fetch. Fetched results provide concrete access to the data model objects. You can access retrieved objects through the controller's `fetchedObjects` property.

Detecting Changes

Fetched results might be used as a table data source or to fill out an object settings form, or for any other purpose you might think of. Whether you're retrieving just one object or many, the fetched results controller offers you direct access to those managed objects on request.

So how do you make sure that your fetched data remains current? After adding new objects or otherwise changing the data store, you want to fetch a fresh set of results. Subscribe to the results controller's `controllerDidChangeContent:` callback. This method notifies your class when changes affect your fetched objects. To subscribe, declare the `NSFetchedResultsControllerDelegate` protocol and assign the controller's delegate as follows. After setting the results' delegate, you receive a callback each time the data store updates.

```
fetchedResultsController.delegate = self;
```

Removing Objects

Retaining consistency can prove slightly harder than you might first expect, especially when deleting relationship objects. Consider the following code. It iterates through each person in the fetched object results and deletes them before saving the context.

```
- (void) removeObjects
{
    NSError *error = nil;
    for (Person *person in
    fetchedResultsController.fetchedObjects)
        [context deleteObject:person];

    if (![context save:&error])
        NSLog(@"Error %@ (%@)", [error localizedFailureReason]);
    [self fetchObjects];
}
```

Core Data ensures internal consistency before writing data out, throwing an error if it cannot. Some models that use cross-references are more complicated than the simple one shown in Figure 12-1. In some data models, you must clear lingering references before the object can safely be removed from the persistent store. If not, objects may point to deleted items, which is a situation that can lead to bad references.

You can set Core Data delete rules in the data model inspector. Delete rules control how an object responds to an attempted delete. You can "Deny" delete requests, ensuring that a relationship has no connection before allowing object deletion. Nullify resets

From the Library of Eric Berna

inverse relationships before deleting an object. Cascade deletes an object plus all its relationships; for example, you could delete an entire department (including its members) all at once with a cascade. No Action provides that the objects pointed to by a relationship remain unaffected, even if those objects point back to the item about to be deleted.

In the sample code that accompanies this chapter, the introductory project (essentially Recipe 0 for this chapter) nullifies its connections. The department/members relationship represents an inverse relationship. By using Nullify, the default delete rule, you do not need to remove the member from the department list before deleting a person.

Xcode issues warnings when it detects nonreciprocal relationships. Avoid unbalanced relationships to simplify your code and provide better internal consistency. If you cannot avoid nonreciprocal items, you need to take them into account when you create your delete methods.

Recipe: Using Core Data for a Table Data Source

Core Data on iOS works closely with table views. The `NSFetchedResultsController` class includes features that simplify the integration of Core Data objects with table data sources. As you can see in the following list, many of the fetched results class's properties and methods are designed from the ground up for table support:

- **Index path access**—The fetched results class offers object-index path integration in two directions. You can recover objects from a fetched object array using index paths by calling `objectAtIndexPath:`. You can query for the index path associated with a fetched object by calling `indexPathForObject:`. These two methods work with both sectioned tables and those tables that are flat—that is, that only use a single section for all their data.
- **Section key path**—The `sectionNameKeyPath` property links a managed object attribute to section names. This property helps determine which section each managed object belongs to. You can set this property directly at any time or you can initialize it when you set up your fetched results controller.

  Recipe 12-1 uses an attribute named `section` to distinguish sections, although you can use any attribute name for this key path. For this example, this attribute is set to the first character of each object name to assign a managed object to a section. Set the key path to `nil` to produce a flat table without sections.
- **Section groups**—Recover section subgroups with the `sections` property. This property returns an array of sections, each of which stores the managed objects whose section attribute maps to the same letter.

  Each returned section implements the `NSFetchedResultsSectionInfo` protocol. This protocol ensures that sections can report their `objects` and `numberOfObjects`, their `name`, and an `indexTitle`—that is, the title that appears on the quick reference index optionally shown above and at the right of the table.

From the Library of Eric Berna

- **Index titles**—The `sectionIndexTitles` property generates a list of section titles from the sections within the fetched data. For Recipe 12-1, that array includes single-letter titles. The default implementation uses the value of each section key to return a list of all known sections.

Two further instance methods, `sectionIndexTitleForSectionName:` and `sectionForSectionIndexTitle:atIndex:`, provide section title lookup features. The first returns a title for a section name. The second looks up a section via its title. Override these to use section titles that do not match the data stored in the section name key.

As these properties and methods reveal, fetched results instances are both table aware and table ready for use. Recipe 12-1 uses these features to duplicate the indexed color name table first introduced in Chapter 11, "Creating and Managing Table Views." The code in this recipe recovers data from the fetched results using index paths, as shown in the method that produces a cell for a given row and the method that tints the navigation bar with the color from the selected row.

Each method used for creating and managing sections is tiny. The built-in Core Data access features reduce these methods to one or two lines each. That's because all the work in creating and accessing the sections is handed over directly to Core Data. The call that initializes each fetched data request specifies what data attribute to use for the sections. Core Data then takes over and performs the rest of the work.

```
fetchedResultsController = [[NSFetchedResultsController alloc]
    initWithFetchRequest:fetchRequest
    managedObjectContext:context
    sectionNameKeyPath:@"section" cacheName:nil];
```

Caching reduces overhead associated with producing data that's structured with sections and indices. Multiple fetch requests are ignored when the data has not changed, minimizing the cost associated with fetch requests over the lifetime of an application. The name used for the cache is arbitrary. Either use `nil` to prevent caching or supply a name in the form of an `NSString`. The preceding snippet uses `nil` to avoid errors related to mutating a fetch request.

Recipe 12-1 **Building a Sectioned Table with Core Data**

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView
{
    return [[fetchedResultsController sections] count];
}

- (NSString *)tableView:(UITableView *)aTableView
    titleForHeaderInSection:(NSInteger)section
{
    // Return the title for a given section
    NSArray *titles = [fetchedResultsController sectionIndexTitles];
```

From the Library of Eric Berna

```
    if (titles.count <= section) return @"Error";
    return [titles objectAtIndex:section];
}

- (NSArray *)sectionIndexTitlesForTableView:(UITableView *)aTableView
{
    return [fetchedResultsController sectionIndexTitles];
}

- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
{
    return [[[fetchedResultsController sections]
        objectAtIndex:section] numberOfObjects];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Retrieve or create a cell
    UITableViewCell *cell =
        [tableView dequeueReusableCellWithIdentifier:@"basic cell"];
    if (!cell) cell = [[UITableViewCell alloc]
        initWithStyle:UITableViewCellStyleDefault
        reuseIdentifier:@"basic cell"];

    // Recover object from fetched results
    NSManagedObject *managedObject =
        [fetchedResultsController objectAtIndexPath:indexPath];
    cell.textLabel.text = [managedObject valueForKey:@"name"];
    UIColor *color = [self getColor:[managedObject valueForKey:@"color"]];

    // Force "white" to display as black
    cell.textLabel.textColor = ([[managedObject valueForKey:@"color"]
        hasPrefix:@"FFFFFF"]) ? [UIColor blackColor] : color;

    return cell;
}

- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    // When a row is selected, color the navigation bar accordingly
    NSManagedObject *managedObject =
        [fetchedResultsController objectAtIndexPath:indexPath];
```

From the Library of Eric Berna

```
    UIColor *color = [self getColor:[managedObject valueForKey:@"color"]];
    self.navigationController.navigationBar.tintColor = color;
}

- (BOOL)tableView:(UITableView *)tableView
    canMoveRowAtIndexPath:(NSIndexPath *)indexPath
{
    return NO;      // no reordering allowed
}
```

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 12 and open the project for this recipe.

Recipe: Search Tables and Core Data

Core Data stores are designed to work efficiently with `NSPredicates`. Predicates allow you to create fetch requests that select only those managed objects that match the predicate's rule or rules. Adding a predicate to a fetch request limits the fetched results to matching objects.

Recipe 12-2 adapts the search table from Chapter 11 to build a Core Data–based solution. This recipe uses a search bar to select data from the persistent store, displaying the results in a table's search sheet. Figure 12-2 shows a search in progress.

As the text in the search bar at the top of the table changes, the search bar's delegate receives a `searchBar:textDidChange:` callback. In turn, that callback method performs a new fetch.

The recipe's `performFetch` method creates a simple predicate based on the text in the search bar. It sets the request's `predicate` property to limit matches to names that contain the text, using a case-insensitive match. `contains` matches text anywhere in a string. The `[cd]` after `contains` refers to case- and diacritic-insensitive matching. Diacritics are small marks that accompany a letter, such as the dots of an umlaut or the tilde above a Spanish *n*.

For more complex queries, assign a compound predicate. Compound predicates allow you to combine simple predicates using standard logical operations such as AND, OR, and NOT. Use the `NSCompoundPredicate` class to build a compound predicate out of a series of component predicates, or include the AND, OR, and NOT notation directly in `NSPredicate` text.

None of the methods from Recipe 12-1 need updating for use with Recipe 12-2's `performFetch` method. All the cell and section methods are tied to the `results` object and its properties, simplifying implementation even when adding these search table features.

From the Library of Eric Berna

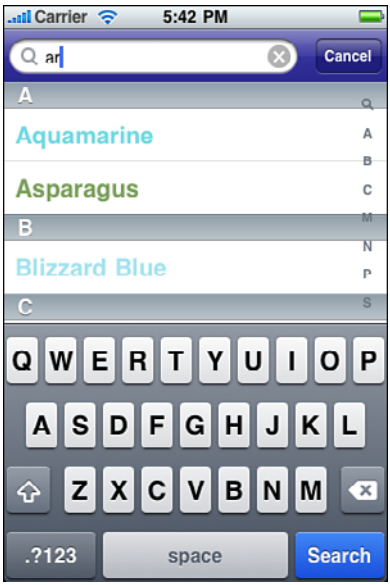


Figure 12-2 To power this search table with Core Data, the fetched results must update each time the text in the search box changes.

Recipe 12-2 **Using Fetch Requests with Predicates**

```
- (void) performFetch
{
    // Init a fetch request
    NSFetchRequest *fetchRequest =
        [[NSFetchRequest alloc] init];
    NSEntityDescription *entity =
        [NSEntityDescription entityForName:@"Crayon"
            inManagedObjectContext:context];
    [fetchRequest setEntity:entity];

    // More than needed for this example
    [fetchRequest setFetchBatchSize:999];

    // Apply an ascending sort for the items
    NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
        initWithKey:@"name" ascending:YES selector:nil];
    NSArray *descriptors = [NSArray arrayWithObject:sortDescriptor];
    [fetchRequest setSortDescriptors:descriptors];
```

From the Library of Eric Berna

```
    // Recover query
    NSString *query = searchBar.text;
    if (query && query.length) fetchRequest.predicate =
        [NSPredicate predicateWithFormat:
            @"name contains[cd] %@", query];

    // Init the fetched results controller
    NSError *error;
    fetchedResultsController = [[NSFetchedResultsController alloc]
        initWithFetchRequest:fetchRequest
        managedObjectContext:context
        sectionNameKeyPath:@"section" cacheName:nil];
    fetchedResultsController.delegate = self;
    if (![fetchedResultsController performFetch:&error])
        NSLog(@"Error: %@", [error localizedFailureReason]);
}
```

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 12 and open the project for this recipe.

Recipe: Integrating Core Data Table Views with Live Data Edits

Recipe 12-3 demonstrates how to move basic table editing tasks into the Core Data world. Its code is based on the basic edits of Recipe 11-4. There are, however, real changes that must be made to provide its Core Data solution. These changes include the following adaptations:

- **Adding and deleting items are restricted to the data source.** Methods that commit an editing style (that is, perform deletes) and that add new cells do not directly address the table view. In the original recipe, each method reloaded the table view data after adds and deletes. Recipe 12-3 saves data to the managed context but does *not* call `reloadData`.
- **Data updates can trigger table reloads.** Subscribe to the delegate `controllerDidChangeContent:` callback. This callback indicates that "fetched results" has updated when data changes have been saved via the managed object context. Use this callback to reload your data and update user options whenever your data changes. This becomes more important as you add undo and redo support, as in Recipe 12-4.

From the Library of Eric Berna

- **The table forbids reordering.** Recipe 12-3's `tableView:canMoveRowAtIndexPath:` method hard-codes its result to `NO`. When working with sorted fetched data sources, users may not reorder that data. This method reflects that reality.

Together, these changes allow your table to work with add and delete edits, as well as content edits. Although content edits are not addressed in this recipe, they involve a similar approach.

The actual add and delete code follows the approach detailed at the start of this chapter. Objects are added by inserting a new entity description. Their attributes are set and the context saved. Objects are deleted from the context, and again the context is saved. These updates trigger the content-changed callbacks for the fetched results delegate.

As this recipe shows, the Core Data interaction simplifies the integration between the data model and the user interface. And that's due in large part to Apple's thoughtful class designs that handle the managed object responsibilities. Recipe 12-3 highlights this design, showcasing the code parsimony that results from using Core Data.

Recipe 12-3 **Adapting Table Edits to Core Data**

```
- (void) setBarButtonItems
{
    // left item is always add
    self.navigationItem.leftBarButtonItem =
        SYSBARBUTTON(UIBarButtonSystemItemAdd, @selector(add));

    // right (edit/done) item depends on both edit mode and item count
    int count = [[fetchedResultsController fetchedObjects] count];
    if (self.tableView.isEditing)
        self.navigationItem.rightBarButtonItem =
            SYSBARBUTTON(UIBarButtonSystemItemDone,
            @selector(leaveEditMode));
    else
        self.navigationItem.rightBarButtonItem =  count ?
            SYSBARBUTTON(UIBarButtonSystemItemEdit,
            @selector(enterEditMode)) : nil;
}

- (BOOL)tableView:(UITableView *)tableView
    canMoveRowAtIndexPath:(NSIndexPath *)indexPath
{
    return NO;      // no reordering allowed
}

-(void)enterEditMode
{
    // Start editing
    [self.tableView deselectRowAtIndexPath:
        [self.tableView indexPathForSelectedRow] animated:YES];
```

From the Library of Eric Berna

```
    [self.tableView setEditing:YES animated:YES];
    [self setBarButtonItems];
}

-(void)leaveEditMode
{
    // finish editing
    [self.tableView setEditing:NO animated:YES];
    [self setBarButtonItems];
}

// Handle deletions
- (void)tableView:(UITableView *)tableView
    commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
    forRowAtIndexPath:(NSIndexPath *)indexPath
{
    // delete request
    if (editingStyle == UITableViewCellEditingStyleDelete)
    {
        NSError *error = nil;
        [context deleteObject:[fetchedResultsController
            objectAtIndexPath:indexPath]];
        if (![context save:&error])
            NSLog(@"Error: %@", [error localizedFailureReason]);
    }

    [self performFetch];
}

// Retrieve the new item and add it
- (void)alertView:(UIAlertView *)alertView
    didDismissWithButtonIndex:(NSInteger)buttonIndex
{
    if (buttonIndex == 0) return;

    NSString *todoAction = [alertView textFieldAtIndex:0].text;
    if (!todoAction || todoAction.length == 0) return;

    ToDoItem *item = (ToDoItem *)[NSEntityDescription
        insertNewObjectForEntityForName:@"ToDoItem"
        inManagedObjectContext:context];
    item.action = todoAction;
    item.sectionName =
        [[todoAction substringToIndex:1] uppercaseString];

    // save the new item
    NSError *error;
    if (![context save:&error])
```

From the Library of Eric Berna

```
        NSLog(@"Error: %@", [error localizedFailureReason]);

    [self performFetch];
}

// Present an alert to solicit the new To Do item
- (void) add
{
    UIAlertView *av = [[UIAlertView alloc]
        initWithTitle:@"To Do" message:nil delegate:self
        cancelButtonTitle:@"Cancel" otherButtonTitles:@"Okay", nil];
    av.alertViewStyle = UIAlertViewStylePlainTextInput;
    [av show];
}
```

> **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 12 and open the project for this recipe.

Recipe: Implementing Undo/Redo Support with Core Data

Core Data simplifies table undo/redo support to an astonishing degree. It provides automatic support for these operations with little programming effort. Here are the steps you need to take to add undo/redo to your table-based application.

First, add an undo manager to the managed object context. After establishing a managed object context, set its undo manager to a newly allocated instance:

```
context.undoManager = self.view.window.undoManager;
context.undoManager.levelsOfUndo = 999;
```

Then, ensure that your view controller becomes the first responder when it is onscreen. Provide the following suite of methods, allowing the view responder to become first responder whenever it appears and resign that first responder status when it moves offscreen:

```
- (BOOL)canBecomeFirstResponder {
    return YES;
}

- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
    [self becomeFirstResponder];
}
```

From the Library of Eric Berna

```
- (void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];
    [self resignFirstResponder];
}
```

These steps provide all the setup needed to use undo management in your table. Recipe 12-4 integrates that undo management into the actual delete and add methods for the table. To make this happen, it brackets the Core Data access with an undo grouping. The `beginUndoGrouping` and `endUndoGrouping` calls appear before and after the context updates and saves with changes. An action name describes the operation that just took place.

These three calls (begin, undo, and setting the action name) comprise all the work needed to ensure that Core Data can reverse its operations. For this minimal effort, your application gains a fully realized undo management system, courtesy of Core Data. Be aware that any undo/redo data will not survive quitting your application. This works just as you'd expect with manual undo/redo support.

Recipe 12-4 **Expanding Cell Management for Undo/Redo Support**

```
- (void) setBarButtonItems
{
    NSMutableArray *items = [NSMutableArray array];
    UIBarButtonItem *spacer =
        SYSBARBUTTON(UIBarButtonSystemItemFlexibleSpace, nil);

    // Add is always visible
    [items addObject:
        SYSBARBUTTON(UIBarButtonSystemItemAdd, @selector(add))];
    [items addObject:spacer];

    // Undo is only enabled with available undo actions
    UIBarButtonItem *undo =
        SYSBARBUTTON(UIBarButtonSystemItemUndo, @selector(undo));
    undo.enabled = context.undoManager.canUndo;
    [items addObject:undo];
    [items addObject:spacer];

    // Ditto for redo
    UIBarButtonItem *redo =
        SYSBARBUTTON(UIBarButtonSystemItemRedo, @selector(redo));
    redo.enabled = context.undoManager.canRedo;
    [items addObject:redo];
    [items addObject:spacer];

    // Disable "Edit" for zero-item lists
    UIBarButtonItem *item;
    int count = [[fetchedResultsController fetchedObjects] count];
```

From the Library of Eric Berna

```
    if (self.tableView.isEditing)
        item = SYSBARBUTTON(UIBarButtonSystemItemDone,
            @selector(leaveEditMode));
    else
    {
        item = SYSBARBUTTON(UIBarButtonSystemItemEdit,
            @selector(enterEditMode));
        item.enabled = (count != 0);
    }

    [items addObject:item];
    toolbar.items = items;
}

// Update the buttons after undo and redo
- (void) undo
{
    [context.undoManager undo];
    [self setBarButtonItems];
}

- (void) redo
{
    [context.undoManager redo];
    [self setBarButtonItems];
}

// Handle deletions
- (void)tableView:(UITableView *)tableView
    commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
    forRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Start the undo grouping
    [context.undoManager beginUndoGrouping];

    // Delete request
    if (editingStyle == UITableViewCellEditingStyleDelete)
    {
        NSError *error = nil;
        [context deleteObject:
            [fetchedResultsController objectAtIndexPath:indexPath]];
        if (![context save:&error])
            NSLog(@"Error: %@", [error localizedFailureReason]);
    }

    // Complete the undo grouping
    [context.undoManager endUndoGrouping];
```

From the Library of Eric Berna

```
    [context.undoManager setActionName:@"Delete"];

    [self performFetch];
}

// Handle additions
- (void)alertView:(UIAlertView *)alertView
    didDismissWithButtonIndex:(NSInteger)buttonIndex
{
    if (buttonIndex == 0) return;

    NSString *todoAction = [alertView textFieldAtIndex:0].text;
    if (!todoAction || todoAction.length == 0) return;

    // Start the undo grouping
    [context.undoManager beginUndoGrouping];

    ToDoItem *item = (ToDoItem *)[NSEntityDescription
        insertNewObjectForEntityForName:@"ToDoItem"
        inManagedObjectContext:context];
    item.action = todoAction;
    item.sectionName =
        [[todoAction substringToIndex:1] uppercaseString];

    // Save the new item
    NSError *error;
    if (![context save:&error])
        NSLog(@"Error: %@", [error localizedFailureReason]);

    // Complete the undo grouping
    [context.undoManager endUndoGrouping];
    [context.undoManager setActionName:@"Add"];

    [self performFetch];
}

// Begin adding a new item
- (void) add
{
    UIAlertView *av = [[UIAlertView alloc]
        initWithTitle:@"To Do" message:nil delegate:self
        cancelButtonTitle:@"Cancel" otherButtonTitles:@"Okay", nil];
    av.alertViewStyle = UIAlertViewStylePlainTextInput;
    [av show];
}
```

From the Library of Eric Berna

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 12 and open the project for this recipe.

Summary

This chapter offered just a taste of Core Data's capabilities. These recipes showed you how to design and implement basic Core Data applications. They used Core Data features to work with its managed object models. You read about defining a model and implementing fetch requests. You saw how to add objects, delete them, modify them, and save them. You learned about predicates and undo operations, and discovered how to integrate Core Data with table views. After reading through this chapter, here are a few final thoughts to take away with you:

- Xcode issues a standard compiler warning when it encounters relationships that are not reciprocal. Nonreciprocal relationships add an extra layer of work, preventing you from taking advantage of simple delete rules such as `Nullify`. Avoid these relationships when possible.
- When moving data from a pre–Core Data store (for example, from object archives or flat text files) into a new database, be sure to use some sort of flag in your user defaults. Check whether you've already performed a data upgrade. You want to migrate user data once when the application is upgraded but not thereafter.
- Predicates are one of my favorite SDK features. Spend some time learning how to construct them and use them with all kinds of objects such as arrays and sets, not just with Core Data.
- If you're not using Core Data with tables, you're missing out on some of the most elegant ways to populate and control your tabular data.
- iCloud provides the perfect match between Core Data and ubiquitous data, extending iOS data to the user's desktop, to each of his or her devices, and to the cloud as a whole. Look up `UIManagedDocument` to learn more about iCloud and Core Data integration.
- Core Data's capabilities go way beyond the basic recipes you've seen in this chapter. Check out Tim Isted and Tom Harrington's *Core Data for iOS: Developing Data-Driven Applications for the iPad, iPhone, and iPod touch*, available from Pearson Education/InformIT/Addison-Wesley for an in-depth exploration of Core Data and its features.

From the Library of Eric Berna

13

Alerting the User

At times, you need to grab your user's attention. New data might arrive or status might change. You might want to tell your user that there's going to be a wait before anything more happens—or that the wait is over and it's time to come back and pay attention. iOS offers many ways to provide that heads-up to the user: from alerts and progress bars to audio pings. In this chapter, you discover how to build these indications into your applications and expand your user-alert vocabulary. You see real-life examples that showcase these classes and discover how to make sure your user pays attention at the right time.

Talking Directly to Your User Through Alerts

Alerts speak to your user. Members of the `UIAlertView` and `UIActionSheet` classes pop up or scroll in above other views to deliver their messages. These lightweight classes add two-way dialog to your apps. Alerts visually "speak" to users and can prompt them to reply. You present your alert onscreen, get user acknowledgment, and then dismiss the alert to move on with other tasks.

If you think that alerts are nothing more than messages with an attached OK button, think again. Alert objects provide incredible versatility. With alert sheets, you can actually build menus, text input, make queries, and more. In this chapter's recipes, you see how to create a wide range of useful alerts that you can use in your own programs.

Building Simple Alerts

To create alert sheets, allocate a `UIAlertView` object. Initialize it with a title and a button array. The title is an `NSString`, as are the buttons. In the button array, each string represents a single button that should be shown.

The method snippet shown here creates and displays the simplest alert scenario. It shows a message with a single OK button. The alert doesn't bother with delegates or callbacks, so its lifetime ends when the user taps a button.

From the Library of Eric Berna

```
- (void) showAlert: (NSString *) theMessage
{
    UIAlertView *av = [[UIAlertView alloc] initWithTitle:@"Title"
        message:theMessage
        delegate:nil
        cancelButtonTitle:@"OK"
        otherButtonTitles:nil];
    [av show];
}
```

Add buttons by introducing them as parameters to `otherButtonTitles:`. Make sure you end your arbitrary list of buttons with `nil`. Adding `nil` tells the method where your list finishes. The following snippet creates an alert with three buttons (Cancel, Option, and OK). Because this code does not declare a delegate, there's no way to recover the alert and determine which of the three buttons was pushed. The alert displays until a user taps and then it automatically dismisses without any further effect.

```
- (void) showAlert: (NSString *) theMessage
{
    UIAlertView *av = [[UIAlertView alloc] initWithTitle:@"Title"
        message:theMessage
        delegate:nil
        cancelButtonTitle:@"Cancel"
        otherButtonTitles: @"Option", @"OK", nil];
    [av show];
}
```

When working with alerts, space is often at a premium. Adding more than two buttons causes the alert to display in multiline mode. Figure 13-1 shows a pair of alerts depicting both two-button (side-by-side display) and three-button (line-by-line display) presentations. Limit the number of alert buttons you add at any time to no more than three or four. Fewer buttons work better; one or two is ideal. If you need to use more buttons, consider using action sheet objects, which are discussed later in this chapter, rather than alert views.

`UIAlertView` objects provide no visual "default" button highlights. The only highlighting is for the Cancel button, as you can see in Figure 13-1. As a rule, Cancel buttons appear at the bottom or left of alerts.

Alert Delegates

Alerts use delegates to recover user choices. In normal use, you often set the delegate to your primary (active) `UIViewController` object, which should declare the `UIAlertViewDelegate` protocol. `UIAlertView` instances allow the delegate to respond to button taps using a simple callback.

From the Library of Eric Berna

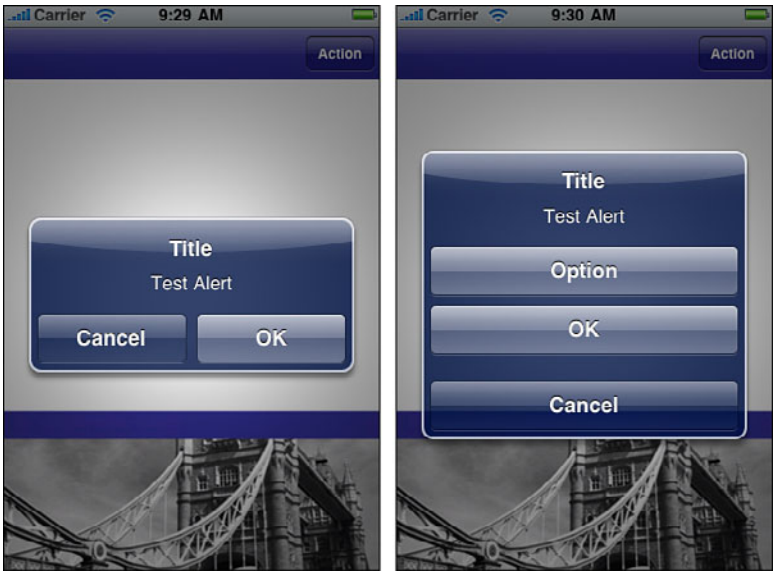


Figure 13-1 Alerts work best with one or two buttons (left). Alerts with more than two buttons stack the buttons as a list, producing a less elegant presentation (right).

Delegate methods enable you to provide meaningful responses as different buttons are pressed. As you've already seen, you can omit that delegate support if all you need to do is show some message with an OK button.

After the user has seen and interacted with your alert, the delegate receives an `alertView:clickedButtonAtIndex:` callback. The second parameter passed to this method indicates which button was pressed. Button numbering begins with zero. The Cancel button, when defined, is always button 0. Even though it appears at the left in some views and the bottom at others, its button numbering remains the same. This is not true for action sheet objects, which are discussed later in this chapter.

Here is a simple example of an alert presentation and callback, which prints out the selected button number to the debugging console:

```
@interface TestBedViewController : UIViewController
    <UIAlertViewDelegate>
@end

@implementation TestBedViewController
- (void) alertView:(UIAlertView *) alertView
    clickedButtonAtIndex: (int) index
{
    NSLog(@"User selected button %d\n", index);
}
```

From the Library of Eric Berna

```
- (void) showAlert
{
    UIAlertView *av = [[UIAlertView alloc]
        initWithTitle:@"Alert View Sample"
        message:@"Select a Button"
        delegate:self
        cancelButtonTitle:@"Cancel"
        otherButtonTitles:@"One", @"Two", @"Three", nil];
    av.tag = MAIN_ALERT;
    [av show];
}
@end
```

If your controller produces multiple alert types, tags can help identify which alert produced a given callback. Unlike controls that use target-action pairs, all alerts trigger the same methods. Adding an alert-tag-based switch statement lets you differentiate your responses to each alert.

Displaying the Alert

As you've seen, the `show` method tells your alert to appear onscreen. When shown, the alert works in a modal fashion. That is, it dims the screen behind it and blocks user interaction with your application behind the modal window. This modal interaction continues until your user acknowledges the alert through a button tap, typically by selecting OK or Cancel. Upon doing so, control passes to any alert delegate, allowing that delegate to finish working with the alert and respond to the selected button.

The alert sheet properties remain modifiable after creation. You may customize the alert by updating its `title` or `message` properties. The message is the optional text that appears below the alert title and above its buttons. You can also change the alert's frame and add subviews.

Kinds of Alerts

Starting with iOS 5.0, the `alertViewStyle` property allows you to create several alert styles. The default style (`UIAlertViewStyleDefault`) is a standard alert, with a title, message text, followed by buttons, as shown in Figure 13-1. It is the bread and butter of the alert world, allowing you to query for button presses such as Yes/No, Cancel/Okay, and other simple choices.

iOS 5.0 introduced three more styles, specifically for entering text:

- **`UIAlertViewStylePlainTextInput`**—This alert style enables users to enter text.
- **`UIAlertViewStyleSecureTextInput`**—When security is an issue, this alert style allows users to enter text, which is automatically obscured as they type it. The text

From the Library of Eric Berna

appears as a series of large dots, but the input can be read programmatically by the delegate callback.

- **`UIAlertViewStyleLoginAndPasswordInput`**—This alert style offers two entry fields, including a plain-text user account login field and an obscured text password field.

When working with text entry alerts, keep your button choices simple. Use no more than two side-by-side buttons—typically OK and Cancel. Too many buttons will create improper visuals, with text fields floating off above or to the sides of the alert.

You can recover the entered text from the alert view by retrieving each text field. The `textFieldAtIndex:` method takes one argument, an integer index starting at 0, and returns the text field at that index. In real use, the only text field that is not at index 0 is the password field, which uses index 1. Once you've retrieved a text field, you can query its contents using its `text` property.

"Please Wait": Showing Progress to Your User

Waiting is an intrinsic part of the computing experience and will remain so for the foreseeable future. It's your job as a developer to communicate that fact to your users. Cocoa Touch provides classes that tell your users to wait for a process to complete. These progress indicators come in two forms: as a spinning wheel that persists for the duration of its presentation and as a bar that fills from left to right as your process moves forward from start to end. The classes that provide these indications are as follows:

- **`UIActivityIndicatorView`**—A progress indicator offers a spinning circle that tells your user to wait without providing specific information about its degree of completion. iOS's activity indicator is small, but its live animation catches the user's eye and is best suited for quick disruptions in a normal application. Recipe 13-1 presents a simple alert embedded as an activity indicator.
- **`UIProgressView`**—This view presents a progress bar. The bar provides concrete feedback as to how much work has been done and how much remains while occupying a relatively small onscreen space. It presents as a thin, horizontal rectangle that fills itself from left to right as progress takes place. This classic user interface element works best for long delays, where users want to know to what degree the job has finished. See Recipe 13-2 for an example of it in use.

Be aware of blocking. Both of these classes must be used on your main thread, as is the rule with GUI objects. Computationally heavy code can block, keeping views from updating in real time. If your code blocks, your progress view may not update in real time as progress is actually made, getting stuck on its initial value instead.

Should you need to display asynchronous feedback, use threading. The edge-detection discussed in Chapter 7, "Working with Images," provides a good example. It uses a

From the Library of Eric Berna

`UIActivityIndicatorView` on the main thread and performs its computation on a second thread. Your threaded computations can then perform view updates on the main thread to provide a steady stream of progress notifications that will keep your user in sync with the actual work being done.

Using `UIActivityIndicatorView`

`UIActivityIndicatorView` instances offer lightweight views that display a standard rotating progress wheel, as shown previously in Figure 13-2. The key word to keep in mind when working with these views is "small." All activity indicators are tiny and do not look right when zoomed past their natural size.

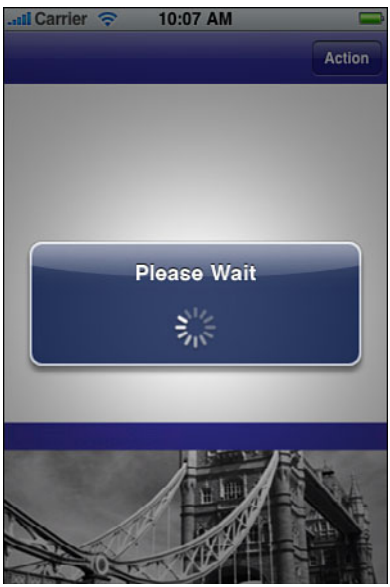


Figure 13-2 Removing buttons from an alert lets you create heads-up displays about ongoing actions.

iOS offers several different styles of the `UIActivityIndicatorView` class. `UIActivityIndicatorViewStyleWhite` and `UIActivityIndicatorViewStyleGray` are 20×20 pixels in size. The white version looks best against a black background, and the gray looks best against white. It's a thin, sharp style.

Take care when choosing whether to use white or gray. An all-white presentation does not show at all against a white backdrop. Unfortunately,

From the Library of Eric Berna

`UIActivityIndicatorViewStyle-WhiteLarge` is available only for use on dark backgrounds. It provides the largest, clearest indicator at 37×37 pixels in size.

```
UIActivityIndicatorView *aiv = [[UIActivityIndicatorView alloc]
    initWithActivityIndicatorStyle:
        UIActivityIndicatorViewStyleWhiteLarge];
```

You need not center indicators on the screen. Place them wherever they work best for you. As a clear-backed view, the indicator blends over whatever backdrop view lies behind it. The predominant color of that backdrop helps select which style of indicator to use.

For general use, just add the activity indicator as a subview to the window, view, toolbar, or navigation bar you want to overlay. Allocate the indicator and initialize it with a frame, preferably centered within whatever parent view you're using. Start the indicator in action by sending `startAnimating`. To stop, call `stopAnimating`. Cocoa Touch takes care of the rest, hiding the view when not in use.

Using `UIProgressView`

Progress views enable your users to follow task progress as it happens rather than just saying "Please wait." They present bars that fill from left to right. The bars indicate the degree to which a task has finished. Progress bars work best for long waits where providing state feedback enables your users to retain the feel of control.

To create a progress view, allocate a `UIProgressView` instance and set its frame. To use the bar, issue `setProgress:`. This takes one argument, a floating-point number that ranges between 0.0 (no progress) and 1.0 (finished). Progress view bars come in two styles: basic white or light gray. The `setStyle:` method chooses the kind you prefer, either `UIProgressViewStyleDefault` or `UIProgressViewStyleBar`. The latter is meant for use in toolbars.

Recipe: No-Button Alerts

Alert views offer a simple way to display an asynchronous message without involving user interaction. You can create a `UIAlertView` instance without buttons and use it to create a heads-up display about ongoing actions in your application. Because alerts are modal, they prevent any other user interaction during their tenure. This allows you to block users from touching the screen when the program must pause to handle critical operations.

You can build a HUD alert and show it just as you would a normal buttoned version. No-button alerts provide an excellent way to throw up a "Please Wait" message, as shown in Figure 13-2.

No-button alerts present a special challenge because they cannot call back to a delegate method. They do not auto-dismiss, even when tapped. Instead, you must manually dismiss the alert when you are done displaying it. Call `dismissWithClickedButtonIndex:animated:` to do so.

From the Library of Eric Berna

Recipe 13-1 builds a custom `ActivityAlert` class, with a static alert view instance. No more than one alert can ever be active at a time, so this implementation uses class methods to ensure that the class acts as a singleton instance.

A `UIActivityIndicator` appears below the alert title. This creates the progress wheel you see at the bottom of the alert in Figure 13-2. This provides visual feedback to the user that some activity or process is ongoing that prevents user interaction.

Once an alert is created, it works like any other view, and you can add subviews and otherwise update its look. Unfortunately, Interface Builder does not offer alert views in its library, so all customization must be done in code, as shown here. Recipe 13-1 builds the subview and adds it to the alert after first presenting the alert with `show`. Showing the alert allows it to build a real onscreen view that you can modify and customize.

Be aware that alerts display in a separate window—that is, not the same window that contains the primary view, navigation bar, and bar button that you see in the sample app. An alert's view is not part of your main window's hierarchy.

Another thing to note is that removing buttons can create an imbalance in the overall presentation geometry. The space that the buttons normally occupy does not go away. In Recipe 13-1, that space is used for the activity indicator. When you're just using text, adding a carriage return (`@"\n"`) to the start of your message helps balance the bottom where buttons normally go with the spacing at the top.

Recipe 13-1 **Displaying and Dismissing a No-Button Alert**

```
@implementation ActivityAlert
+ (void) presentWithText: (NSString *) alertText
{
    if (alertView)
    {
        // With an existing alert, update the text and re-show it
        alertView.title = alertText;
        [alertView show];
    }
    else
    {
        // Create an alert with plenty of room
        alertView = [[UIAlertView alloc]
            initWithTitle:alertText
            message:@"\n\n\n\n\n\n"
            delegate:nil
        cancelButtonTitle:nil
        otherButtonTitles: nil];
        [alertView show];

        // Build a new activity indicator and animate it
        activity = [[UIActivityIndicatorView alloc]
```

From the Library of Eric Berna

```
            initWithActivityIndicatorStyle:
                UIActivityIndicatorViewStyleWhiteLarge];
        activity.center = CGPointMake(
            CGRectGetMidX(alertView.bounds),
            CGRectGetMidY(alertView.bounds));
        [activity startAnimating];

        // Add it to the alert
        [alertView addSubview: activity];
    }
}

// Update the alert's title
+ (void) setTitle: (NSString *) aTitle
{
    alertView.title = aTitle;
}

// Update the alert's message, making sure to pad it
// to the proper number of lines. Keep the message short.
+ (void) setMessage: (NSString *) aMessage;
{
    NSString *message = aMessage;
    while ([message componentsSeparatedByString:@"\n"].count < 7)
        message = [message stringByAppendingString:@"\n"];
    alertView.message = message;
}

// Dismiss the alert and reset the static variables
+ (void) dismiss
{
    if (alertView)
    {
        [alertView dismissWithClickedButtonIndex:0 animated:YES];

        [activity removeFromSuperview];
        activity = nil;
        alertView = nil;
    }
}
@end
```

Get This Recipe's Code

To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from the book, go to the folder for Chapter 13 and open the project for this recipe.

From the Library of Eric Berna

Building a Floating Progress Monitor

In the rare case that you must block user interaction for an extended period of time, progress bars on a modal alert allow you to provide status updates to the user. Whenever possible avoid this scenario. Most long-term operations can and should be sent to another thread, with progress updating with nonblocking feedback. For those rare times when you must, you may embed a progress view onto an alert just as you embedded an activity monitor.

To support alert-based progress bars, you'll need to be able to update the embedded bar as progress occurs. This is easily added with an extra method call. You'll probably want to provide text updates as well. Use the message field for this. Finally, a well designed application will allow users to cancel any extended operation like this, so consider providing a cancel button in your alert as well.

```
+ (void) setProgress: (float) amount
{
    progress.progress = amount;
}

// Use short messages
+ (void) setMessage: (NSString *) aMessage;
{
    NSString *message = aMessage;
    while ([message componentsSeparatedByString:@"\n"].count < 7)
        message = [message stringByAppendingString:@"\n"];
    alertView.message = message;
}
```

Recipe: Creating Modal Alerts with Run Loops

The indirect nature of the alert (namely its delegate callback approach) can produce unnecessarily complex code. It's easy to build a custom class that directly returns a button choice value. Consider the following code. It requests an answer from the alert shown in Figure 13-3 and then uses the answer that the class method returns.

```
- (void) alertText: (id) sender
{
    // OK = 1, Cancel = 0
    UIAlertView *alertView = [[UIAlertView alloc]
        initWithTitle:@"What is your name?"
        message:@"Please enter your name"
        delegate:nil
        cancelButtonTitle:@"Cancel"
        otherButtonTitles:@"Okay", nil];
    alertView.alertViewStyle = UIAlertViewStylePlainTextInput;
```

From the Library of Eric Berna

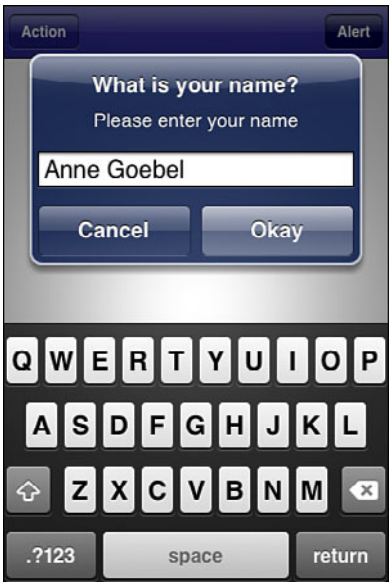


Figure 13-3 This modal alert returns immediate answers because it's built using its own run loop.

```
    ModalAlertDelegate *delegate =
        [ModalAlertDelegate delegateWithAlert:alertView];

    // Build a response from the text field
    NSString *response = @"No worries. I'm shy too.";
    if ([delegate show])
        response = [NSString stringWithFormat:@"Hello %@",
            [alertView textFieldAtIndex:0].text];

    // Display the response
    [[[UIAlertView alloc]
        initWithTitle:nil message:response
        delegate:nil cancelButtonTitle:nil
        otherButtonTitles:@"Okay", nil] show];
}
```

To create an alert that returns an immediate result requires a bit of ingenuity. A custom class called `ModalAlertDelegate` can handle things for you. Create an instance, passing it the alert view you wish to work with. It assigns itself as the alert's delegate and takes responsibility for presenting the alert. It implements its own version of `show`, which you normally call directly on the alert. As you can see in Recipe 13-2, the code calls `CFRunLoopRun()`, which makes the method sit and wait until the user finishes interacting with the alert. The method goes no further as the run loop runs.

From the Library of Eric Berna

The delegate cancels that run loop on a button click and returns the value of the selected item. When the user finishes interacting, the calling method can finally proceed past the run loop.

Be aware that although you can run one alert after another using this method, sometimes the calls may crowd each other. Leave enough time for the previous alert to disappear before presenting the next. Should an alert fail to show onscreen, it's probably due to this overlap issue. In such a case, use a delayed selector to call the next alert request. A tenth of a second offers plenty of time to allow the new alert to show.

Recipe 13-2 **Creating Alerts That Return Immediate Results**

```
@interface ModalAlertDelegate : NSObject <UIAlertViewDelegate>
{
    UIAlertView *alertView;
    int index;
}
+ (id) delegateWithAlert: (UIAlertView *) anAlert;
- (int) show;
@end

@implementation ModalAlertDelegate
- (id)initWithAlert: (UIAlertView *) anAlert
{
    if (!(self = [super init])) return self;
    alertView = anAlert;
    return self;
}

-(void)alertView:(UIAlertView*)aView
    clickedButtonAtIndex:(NSInteger)anIndex
{
    // Store the selected button index
    index = anIndex;

    // Done with the alert view
    alertView = nil;

    // Stop the ongoing run loop and return control
    CFRunLoopStop(CFRunLoopGetCurrent());
}

- (int) show
{
    // Act as the alert's delegate, and show it
    [alertView setDelegate:self];
    [alertView show];
```

From the Library of Eric Berna

```
    // Wait until the user interacts
    CFRunLoopRun();

    return index;
}

+ (id) delegateWithAlert: (UIAlertView *) anAlert
{
    ModalAlertDelegate *mad = [[self alloc] initWithAlert:anAlert];
    return mad;
}
@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 13 and open the project for this recipe.

Recipe: Using Variadic Arguments with Alert Views

Methods that can take a variable number of arguments are called "variadic." They are declared using ellipses after the last parameter. Both `NSLog` and `printf` are variadic. You can supply them with a format string along with any number of arguments.

Because most alerts center on text, it's handy to build methods that create alerts from format strings. Recipe 13-3 creates a `say:` method that collects the arguments passed to it and builds a string with them. The string is then passed to an alert view, which is then shown, providing a handy instant display.

The `say:` method does not parse or otherwise analyze its parameters. Instead, it grabs the first argument, uses that as the format string, and passes the remaining items to the `NSString initWithFormat:arguments:` method. This builds a string, which is then passed to a one-button alert view as its title.

Defining your own utility methods with variadic arguments lets you skip several steps where you have to build a string with a format and then call a method. With `say:` you can combine this into a single call, as follows:

```
[NotificationAlert say:
    @"I am so happy to meet you, %@", yourName];
```

Recipe 13-3 **Using a Variadic Method for `UIAlertView` Creation**

```
+ (void) say: (id)formatstring,...
{
    if (!formatstring) return;

    va_list arglist;
    va_start(arglist, formatstring);
```

From the Library of Eric Berna

```
    id statement = [[NSString alloc]
        initWithFormat:formatstring arguments:arglist];
    va_end(arglist);

    UIAlertView *av = [[UIAlertView alloc]
        initWithTitle:statement message:nil
        delegate:nil cancelButtonTitle:@"Okay"
        otherButtonTitles:nil];
    [av show];
}
```

> **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 13 and open the project for this recipe.

Presenting Simple Menus

`UIActionSheet` instances create simple iOS menus. On the iPhone and iPod touch, they slide choices, basically a list of buttons representing possible actions, onto the screen and wait for the user to respond. On the iPad, they appear in popovers. Action sheets are different from alerts. Alerts stand apart from the interface and are better used for demanding attention. Menus slide into a view and better integrate with ongoing application work. Cocoa Touch supplies five ways to present menus:

- **`showInView`**—On the iPhone and iPod touch, this method slides the menu up from the bottom of the view. On the iPad, the action sheet is centered in the middle of the screen.
- **`showFromToolBar:` and `showFromTabBar:`**—For the iPhone and iPod touch, when you're working with toolbars, tab bars, or any other kinds of bars that provide those horizontally grouped buttons that you see at the bottom of many applications, these methods align the menu with the top of the bar and slide it out exactly where it should be. On the iPad, the action sheet is centered in the middle of the screen.
- **`showFromBarbuttonItem:animated:`**—On the iPad, this method presents the action sheet as a popover from the specified bar button.
- **`showFromRect:inView:animated:`**—Shows the action sheet originating from the rectangle you specify in the coordinates of the view you specify.

The following snippet shows how to initialize and present a simple `UIActionSheet` instance. Its initialization method introduces a concept missing from `UIAlertView`: the Destructive button. Colored in red, a Destructive button indicates an action from which there is no return, such as permanently deleting a file (see Figure 13-4, left). Its bright red color warns the user about the choice. Use this option sparingly.

From the Library of Eric Berna



Figure 13-4 On the iPhone and iPod touch, action sheet menus slide in from the bottom of the view. Although the Destructive menu button appears gray here (left), it is red on iOS devices and indicates permanent actions with possible negative consequences to your users. Adding many menu items produces the scrolling list on the right.

Action sheet values are returned in button order. In the left Figure 13-4 example, the Destructive button is number 0 and the Cancel button is number 4. This behavior contradicts alert view values, where the Cancel button returns 0. With action sheets, the Cancel button's position sets its number. This may vary, depending on how you add your buttons. In some configurations (no Destructive button), Cancel is presented as the first item and corresponds to choice 0. Always test your action sheets to confirm layout and button order.

```
- (void) actionSheet:(UIActionSheet *)actionSheet
    didDismissWithButtonIndex:(NSInteger)buttonIndex
{
    self.title = [NSString stringWithFormat:@"Button %d", buttonIndex];
}

- (void) action: (UIBarButtonItem *) sender
{
    // Destructive = 0, One = 1, Two = 2, Three = 3, Cancel = 4
    UIActionSheet *actionSheet = [[UIActionSheet alloc]
        initWithTitle:@"Title"
        delegate:self
        cancelButtonTitle:@"Cancel"
```

From the Library of Eric Berna

```
        destructiveButtonTitle:@"Destructive"
        otherButtonTitles:@"One", @"Two", @"Three", nil];
    [actionSheet showFromBarButtonItem:sender animated:YES];
}
```

Avoid using Cancel buttons on the iPad. Allow users to tap outside the action sheet to cancel interaction after presenting the sheet.

```
UIActionSheet *actionSheet = [[UIActionSheet alloc]
    initWithTitle:theTitle delegate:nil
    cancelButtonTitle:IS_IPAD ? nil : @"Cancel"
    destructiveButtonTitle:nil otherButtonTitles:nil];
```

Cancelling an iPad action sheet returns a (default) value of –1. You can override this, but I cannot recommend doing so.

Scrolling Menus

As a rough rule of thumb, you can fit a maximum of about seven buttons (including Cancel) into a portrait orientation and about four buttons into landscape on the iPhone and iPod touch. (There's quite a bit more room on the iPad.) Going beyond this number triggers the scrolling presentation shown in Figure 13-4 (right). Notice that the Cancel button is presented below the list, although its numbering remains consistent with shorter menu presentations. The Cancel button is always numbered after any previous buttons. As Figure 13-4 demonstrates, this presentation falls fairly low on the aesthetics scale and should be avoided where possible.

> **Note**
>
> You can use the same second run loop approach shown in Recipe 13-2 to retrieve results with action sheets as you can with alert views.

Displaying Text in Action Sheets

Action sheets offer many of the same text presentation features as alert views, but they do so with a much bigger canvas. The following snippet demonstrates how to display a text message using a `UIActionSheet` object. It provides a handy way to present a lot of text at once.

```
- (void) show: (id)formatstring,...
{
    if (!formatstring) return;

    va_list arglist;
    va_start(arglist, formatstring);
    id statement = [[NSString alloc]
        initWithFormat:formatstring arguments:arglist];
    va_end(arglist);
```

From the Library of Eric Berna

```
    UIActionSheet *actionSheet = [[UIActionSheet alloc]
        initWithTitle:statement
        delegate:nil cancelButtonTitle:nil
        destructiveButtonTitle:nil
        otherButtonTitles:@"OK", nil];

   [actionSheet showInView:self.view];
}
```

Recipe: Building Custom Overlays

Although `UIAlertView` and `UIActionSheet` provide excellent modal progress indicators, you can also roll your own completely from scratch. Recipe 13-4 uses a simple tinted `UIView` overlay with a `UIActivityIndicatorView`.

The overlay view occupies the entire screen size. Using the entire screen lets the overlay fit over the navigation bar. That's because the overlay view must be added to the application window and not, as you might think, to the main `UIViewController`'s view. That view only occupies the space under the navigation bar (the "application frame" in `UIScreen` terms), allowing continued access to any buttons and other control items in the bar. Filling the window helps block that access.

To restrict any user touches with the screen, the overlay sets its `userInteractionEnabled` property to `YES`. This catches any touch events, preventing them from reaching the GUI below the alert, creating a modal presentation where interaction cannot continue until the alert has finished. You can easily adapt this approach to dismiss an overlay with a touch, but be aware when creating alerts like this that the view does not belong to a view controller. It will not update itself during device orientation changes. If you need to work with a landscape/portrait-aware system, you can catch that value before showing the overlay and subscribe to reorientation notifications.

Recipe 13-4 **Presenting and Hiding a Custom Alert Overlay**

```
- (void) removeOverlay: (UIView *) overlayView
{
    [overlayView removeFromSuperview];
}

- (void) action: (id) sender
{
    UIWindow *window = self.view.window;

    // Create a tinted overlay, sized to the window
    UIView *overlayView = [[UIView alloc] initWithFrame:window.bounds];
    overlayView.backgroundColor =
        [[UIColor blackColor] colorWithAlphaComponent:0.5f];
    overlayView.userInteractionEnabled = YES;
```

From the Library of Eric Berna

```
    // Add an activity indicator
    UIActivityIndicatorView *aiv = [[UIActivityIndicatorView alloc]
        initWithActivityIndicatorStyle:
            UIActivityIndicatorViewStyleWhiteLarge];
    aiv.center = CGPointMake(
        CGRectGetMidX(overlayView.bounds),
        CGRectGetMidY(overlayView.bounds));
    [aiv startAnimating];

    [overlayView addSubview:aiv];
    [window addSubview:overlayView];

    // Use a time delay to simulate a task finishing
    [self performSelector:@selector(removeOverlay:)
        withObject:overlayView afterDelay:5.0f];
}
```

> **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 13 and open the project for this recipe.

Tappable Overlays

A custom overlay can present information as well as limit interaction. You can easily expand the overlay approach from Recipe 13-4 to dismiss itself on a touch. When tapped, a view removes itself from the screen. This behavior makes it particularly suitable for showing information in a way normally reserved for the `UIAlertView` class.

```
@interface TappableOverlay : UIView
@end
@implementation TappableOverlay
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    // Remove this view when it is touched
    [self removeFromSuperview];
}
@end
```

Recipe: Basic Popovers

At the time this book is being written, popovers remain an iPad-only feature. Often you'll want to present information using a popover as an alternative to presenting a modal view. There are several basic rules of popovers that you need to incorporate into your day-to-day development:

From the Library of Eric Berna

- **Always hang onto your popovers.** Create strong local variables that retain your popovers until they are no longer needed. In Recipe 13-5, the variable is reset once the popover is dismissed.
- **Always check for existing popovers and dismiss them.** This is especially important if you create popovers with different roles in your apps. For example, you may provide popovers for more than one bar button item. Before you present any new popover, dismiss the existing one.
- **Always set your content size.** The default iPad popover is long and thin and may not appeal to your design aesthetics. Setting the `contentSizeForViewInPopover` property of your view controllers allows you to specify exactly what dimensions the popover should use.
- **Think carefully about your permitted arrow directions.** With bar button items (as in Recipe 13-5), I *always* use an up direction because my bar buttons are consistently at the top of my screen. Pick an arrow direction that makes sense, always pointing toward the calling object. If you have a master controller on the left of your screen and the popover goes out to its right, use a left arrow. Arrow directions can be OR'ed together as needed when space is tight, to pick the best possible solution.
- **Always provide an iPhone option.** Don't sacrifice functionality when changing platforms. Instead, provide an iPhone alternative, usually a modally presented controller instead of a popover.
- **Never add a Done button to popovers.** Although you always add a Done button to modal presentations, skip them in popovers. Tapping outside of the popover is conventionally understood to dismiss the popover. A Done button is simply redundant.

Recipe 13-5 **Basic Popovers**

```
- (void) popoverControllerDidDismissPopover:
    (UIPopoverController *)popoverController
{
    // Stop holding onto the popover
    popover = nil;
}

- (void) action: (id) sender
{
    // Always check for existing popover
    if (popover)
        [popover dismissPopoverAnimated:YES];

    // Retrieve the nav controller from the storyboard
    UIStoryboard *storyboard =
        [UIStoryboard storyboardWithName:@"Storyboard"
```

From the Library of Eric Berna

```
            bundle:[NSBundle mainBundle]];
    UINavigationController *controller =
        [storyboard instantiateInitialViewController];

    // Present either modally or as a popover
    if (IS_IPHONE)
    {
        [self.navigationController
            presentModalViewController:controller animated:YES];
    }
    else
    {
        // No done button on iPads
        UIViewController *vc = controller.topViewController;
        vc.navigationItem.rightBarButtonItem = nil;

        // Set the content size to iPhone-sized
        vc.contentSizeForViewInPopover =
            CGSizeMake(320.0f, 480.0f - 44.0f);

        // Create and deploy the popover
        popover = [[UIPopoverController alloc]
            initWithContentViewController:controller];
        [popover presentPopoverFromBarButtonItem:sender
            permittedArrowDirections:UIPopoverArrowDirectionUp
            animated:YES];
    }
}
```

Get This Recipe's Code

To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from the book, go to the folder for Chapter 13 and open the project for this recipe.

Recipe: Local Notifications

Local notifications can alert the user when your application is not running. They offer a simple way to schedule an alert that presents itself at a specific date and time. Unlike push notifications, local notifications do not require any network access and do not communicate with remote servers. As their name suggests, they are handled entirely on a local level.

Local notifications are meant to be used with schedules, such as calendar and to-do list utilities. You can also use them with multitasking applications to provide updates when the application is not running in the foreground. For example, a location-based app might pop up a notification to let a user know that the app has detected that the user is nearby the local library and that books are ready to be picked up.

From the Library of Eric Berna

The system does not present local notifications when the application is active, only when it's suspended or running in the background. Recipe 13-6 forces the app to quit as it schedules the notification for 5 seconds in the future to allow the notification to appear properly. Don't *ever* do this in App Store applications, but if you don't do it here, you'll miss the notification.

As with push notifications, tapping the action button will relaunch the application, moving control back into the `application:didFinishLaunchingWithOptions:` method. If you retrieve the options dictionary, the notification object can be found via the `UIApplicationLaunchOptionsLocalNotificationKey` key.

Recipe 13-6 **Scheduling Local Notifications**

```
- (void) action: (id) sender
{
    UIApplication *app = [UIApplication sharedApplication];

    // Remove all prior notifications
    NSArray *scheduled = [app scheduledLocalNotifications];
    if (scheduled.count)
        [app cancelAllLocalNotifications];

    // Create a new notification
    UILocalNotification* alarm =
        [[UILocalNotification alloc] init];
    if (alarm)
    {
        alarm.fireDate =
            [NSDate dateWithTimeIntervalSinceNow:5.0f];
        alarm.timeZone = [NSTimeZone defaultTimeZone];
        alarm.repeatInterval = 0;
        alarm.alertBody = @"Five Seconds Have Passed";
        [app scheduleLocalNotification:alarm];

        // Force quit. Never do this in App Store code.
        exit(0);
    }
}
```

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 13 and open the project for this recipe.

From the Library of Eric Berna

Alert Indicators

When your application accesses the Internet from behind the scenes, it's polite to let your user know what's going on. Rather than create a full-screen alert, Cocoa Touch provides a simple application property that controls a spinning network activity indicator in the status bar. Figure 13-5 shows this indicator in action, to the right of the Wi-Fi indicator and to the left of the current time display.

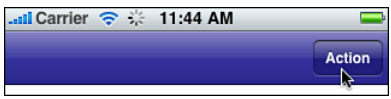


Figure 13-5 The network activity indicator is controlled by a `UIApplication` property.

The following snippet demonstrates how to access this property, doing little more than toggling the indicator on or off. In real-world use, you'll likely perform your network activities on a secondary thread. Make sure you perform this property change on the main thread so the GUI can properly update itself.

```
- (void) action: (id) sender
{
    // Toggle the network activity indicator
    UIApplication *app = [UIApplication sharedApplication];
    app.networkActivityIndicatorVisible =
        !app.networkActivityIndicatorVisible;
}
```

Badging Applications

If you've used iOS for any time, you've likely seen the small, red badges that appear over applications on the home screen. These might indicate the number of missed phone calls or unread e-mails that have accumulated since the user last opened Phone or Mail.

To set an application badge from within the program itself, set the `applicationIconBadgeNumber` property to an integer. To hide badges, set `applicationIconBadgeNumber` to 0.

Recipe: Simple Audio Alerts

Audio alerts "speak" directly to your users. They produce instant feedback—assuming users are not hearing impaired. Fortunately, Apple built basic sound playback into the Cocoa Touch SDK through System Audio services. This works very much like system audio on a Macintosh.

From the Library of Eric Berna

The alternatives include using Audio Queue calls or `AVAudioPlayer`. Audio Queue playback is expensive to program and involves much more complexity than simple alert sounds need. In contrast, you can load and play system audio with just a few lines of code. `AVAudioPlayer` also has its drawbacks. It interferes with iPod audio. In contrast, System Audio can perform a sound without interrupting any music that's currently playing, although that may admittedly not be the result you're looking for, as alerts can get lost in the music.

Alert sounds work best when kept short, preferably 30 seconds or shorter according to Apple. System Audio plays PCM and IMA audio only. That means limiting your sounds to AIFF, WAV, and CAF formats.

System Sounds

To build a system sound, call `AudioServicesCreateSystemSoundID` with a file URL pointing to the sound file. This call returns an initialized system sound object, which you can then play at will. Just call `AudioServicesPlaySystemSound` with the sound object. That single call does all the work.

```
AudioServicesPlaySystemSound(mySound);
```

The default implementation of system sounds allows them to be controlled by the Sound Effects preference in Settings. When effects are disabled, the sound will not play. To override this preference and always play the sound, you can set a property flag as such:

```
// Identify it as a non UI Sound
AudioServicesCreateSystemSoundID(baseURL, &mysound);
AudioServicesPropertyID flag = 0;  // 0 means always play
AudioServicesSetProperty(
    kAudioServicesPropertyIsUISound,
    sizeof(SystemSoundID),
    &mysound,
    sizeof(AudioServicesPropertyID),
    &flag);
```

When iPod audio is playing, the system sound generally plays back at the same volume, so users may miss your alert. Consider using vibration in addition to or in place of music. You can check the current playback state by testing as follows. Make sure you include `MediaPlayer/MediaPlayer.h` and link to the MediaPlayer framework.

```
if ([MPMusicPlayerController iPodMusicPlayer].playbackState ==
     MPMusicPlaybackStatePlaying)
```

Add an optional system sound completion callback to notify your program when a sound finishes playing by calling `AudioServicesAddSystemSoundCompletion`. Unless you use short sounds that are chained one after another, this is a step you can generally skip.

Clean up your sounds by calling `AudioServicesDisposeSystemSoundID` with the sound in question. This frees the sound object and all its associated resources.

From the Library of Eric Berna

> **Note**
> To use these system sound services, make sure to include `AudioToolbox/AudioServices.h` in your code and link to the Audio Toolbox framework.

Vibration

As with audio sounds, vibration immediately grabs a user's attention. What's more, vibration works for nearly all users, including those who are hearing or visually impaired. Using the same System Audio services, you can vibrate as well as play a sound. All you need is the following one-line call to accomplish it, as used in Recipe 13-7:

```
AudioServicesPlaySystemSound(kSystemSoundID_Vibrate);
```

You cannot vary the vibration parameters. Each call produces a short 1-to-2-second buzz. On platforms without vibration support (such as the iPod touch and iPad), this call does nothing—but will not produce an error.

Alerts

Audio Services provides a vibration/sound mashup called an alert sound, which is invoked as follows:

```
AudioServicesPlayAlertSound(mySound);
```

This call, which is also demonstrated in Recipe 13-7, plays the requested sound and, possibly, vibrates or plays a second alert. On iPhones, when the user has set Settings > Sound > Ring > Vibrate to ON, it vibrates the phone. Second-generation and later iPod touch units play the sound sans vibration (which is unavailable on those units) through the onboard speaker. First-generation iPod touches (see if you can find one these days!) play a short alert melody in place of the sound on the device speaker while playing the requested audio through to the headphones.

Delays

The first time you play back a system sound on iOS, you may encounter delays. You may want to play a silent sound on application initialization to avoid a delay on subsequent playback.

> **Note**
> When testing on iPhones, make sure you have not enabled the silent ringer switch on the left side of the unit. This oversight has tripped up many iPhone developers. If your alert sounds must always play, consider using the `AVAudioPlayer` class.

From the Library of Eric Berna

Recipe 13-7 **Playing Sounds, Alerts, and Vibrations Using Audio Services**

```
@interface TestBedViewController : UIViewController
{
    SystemSoundID mysound;
}
@end

@implementation TestBedViewController
- (void) dealloc
{
    // Always dispose of allocated system sounds
    if (mysound) AudioServicesDisposeSystemSoundID(mysound);
}

- (void) playSound
{
    // Choose how to present the sound
    if ([MPMusicPlayerController iPodMusicPlayer].playbackState
        ==  MPMusicPlaybackStatePlaying)
        AudioServicesPlayAlertSound(mysound);
    else
        AudioServicesPlaySystemSound(mysound);
}

- (void) vibrate
{
    // Vibrate only works on iPhones
    AudioServicesPlaySystemSound (kSystemSoundID_Vibrate);
}

- (void) loadView
{
    [super loadView];

    // Create the sound
    NSString *sndpath = [[NSBundle mainBundle]
        pathForResource:@"basicsound" ofType:@"wav"];
    CFURLRef baseURL =
        (__bridge CFURLRef)[NSURL fileURLWithPath:sndpath];

    // Identify it as not a UI Sound
    AudioServicesCreateSystemSoundID(baseURL, &mysound);
    AudioServicesPropertyID flag = 0;  // 0 means always play
    AudioServicesSetProperty(
        kAudioServicesPropertyIsUISound,
        sizeof(SystemSoundID),
```

From the Library of Eric Berna

```
        &mysound,
        sizeof(AudioServicesPropertyID),
        &flag);

    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Sound", @selector(playSound));
    self.navigationItem.leftBarButtonItem =
        BARBUTTON(@"Vibrate", @selector(vibrate));
}
@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 13 and open the project for this recipe.

One More Thing: Showing the Volume Alert

iOS offers a built-in alert that you can display to allow users to adjust the system volume. Figure 13-6 shows this alert, which consists of a slider and a Done button. Invoke this alert by issuing the following Media Player function:

```
- (void) action
{
    // Show the Media Player volume settings alert
    MPVolumeSettingsAlertShow();
}
```

Test whether this alert is visible by issuing `MPVolumeSettingsAlertIsVisible()`. This returns a Boolean value reflecting whether the alert is already onscreen. Hide the alert with `MPVolumeSettingsAlertHide()`, which dismisses the alert regardless of whether the user taps Done. For these functions to work, you must link to the MediaPlayer framework and import the Media Player headers.

From the Library of Eric Berna

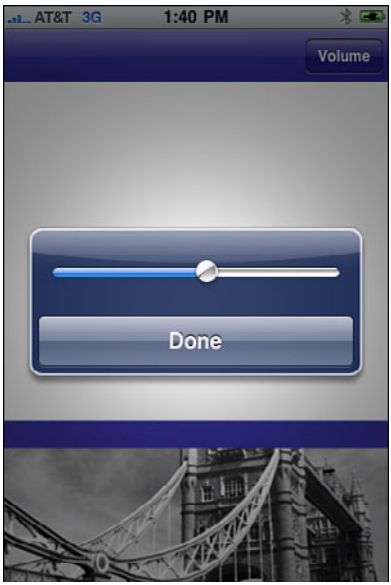


Figure 13-6 The Media Player class's utility volume alert panel.

Summary

This chapter introduced ways to interact directly with your user. You learned how to build alerts—visual, auditory, and tactile—that grab your user's attention and can request immediate feedback. Use these examples to enhance the interactive appeal of your programs and leverage some unique iPhone-only features. Here are a few thoughts to carry away from this chapter:

- Whenever any task will take a noticeable amount of time, be courteous to your user and display some kind of progress feedback. iOS offers many ways to do this, from heads-up displays to status bar indicators and beyond. You may need to divert the non-GUI elements of your task to a new thread to avoid blocking.
- Alerts take users into the moment. They're designed to elicit responses while communicating information. And, as you saw in this chapter, they're almost insanely customizable. It's possible to build entire applications around the simple `UIAlertView`.

From the Library of Eric Berna

- Don't be afraid of the run loop. A modal response from an alert or action sheet lets you poll users for immediate choices without being dependent on asynchronous callbacks. To be fair, don't be afraid of blocks and asynchronous callbacks either. Using runloops may startle some of your coworkers unless you evangelize heavily. If you run your own shop, you probably have more freedom to make executive decisions that include runloop implementations.
- Use local notifications sparingly. Never display them unless there's a compelling reason why the user would *want* them to be displayed. It's very easy to alienate your user and get your app kicked off the device by overusing local notification alerts.
- If blue-colored system-supplied features do not match your application design needs, skip them. You can easily build your own alerts and menus using `UIView` instances and animation.
- Audio feedback, including beeps and vibration, can enhance your programs and make your interaction richer. Using system sound calls means that your sounds play nicely with iPod functionality and won't ruin the ongoing listening experience. At the same time, don't be obnoxious. Use alert sounds sparingly and meaningfully to avoid annoying your users.

From the Library of Eric Berna

14

Device Capabilities

Each iPhone device represents a meld of unique, shared, momentary, and persistent properties. These properties include the device's current physical orientation, its model name, its battery state, and its access to onboard hardware. This chapter looks at the device from its build configuration to its active onboard sensors. It provides recipes that return a variety of information items about the unit in use. 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 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.

Accessing Basic Device Information

The `UIDevice` class exposes key device-specific properties, including the iPhone or iPod touch model being used, the device name, and the OS name and version. It's a one-stop solution for pulling out certain system details. Each method is an instance method, which is called using the `UIDevice` singleton, via `[UIDevice currentDevice]`.

The system information you can retrieve from `UIDevice` includes these items:

- **`systemName`**—This returns the name of the operating system currently in use. For current generations of iOS devices, there is only one OS that runs on the platform: iPhone OS. Apple has not yet updated this name to match the general iOS rebranding.
- **`systemVersion`**—This value lists the firmware version currently installed on the unit: for example, 4.2, 4.3, 5.0, and so on.
- **`model`**—The iPhone model returns a string that describes its platform—namely iPhone, iPad, and iPod touch. Should iOS be extended to new devices, additional strings will describe those models. `localizedModel` provides a localized version of this property.

From the Library of Eric Berna

- **userInterfaceIdiom**—This property represents the interface style used on the current device, namely either iPhone (for iPhone and iPod touch) or iPad. Other idioms may be introduced as Apple offers additional platform styles.
- **name**—This string presents the iPhone name assigned by the user in iTunes, such as "Joe's iPhone" or "Binky." This name is also used to create the local host name for the device.

Here are a few examples of these properties in use:

```
UIDevice *device = [UIDevice currentDevice];
NSLog(@"System name: %@", device.systemName);
NSLog(@"Model: %@", device.model);
NSLog(@"Name: %@", device.name);
```

For current iOS releases, you can use the idiom check with a simple Boolean test. Here's an example:

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

You can assume for now that if the test fails, it falls back to the only other idiom, in this case iPhone/iPod touch. If and when Apple releases a new family of devices, you'll need to update your code accordingly. Here's an example:

```
if (IS_IPAD)
    //  iPad-specific code
else if (IS_IPHONE)
     //  iPhone/iPod touch-specific code
else if (IS_OTHER_DEVICE)
     // other device-specific code
```

Adding Device Capability Restrictions

The Info.plist property list allows you to specify application requirements when you submit applications to iTunes. These restrictions enable you to tell iTunes what device features your application needs.

Each iOS unit provides a unique feature set. Some devices offer cameras and GPS capabilities. Others don't. Some have onboard gyros, autofocus, and other powerful options. You specify what features are needed to run your application on a device.

When you include the `UIRequiredDeviceCapabilities` key in your Info.plist file, iTunes limits application installation to devices that offer the required capabilities. Provide this list as either an array of strings or a dictionary.

An array specifies each required capability; each item in that array must be present on your device. A dictionary allows you to explicitly require or prohibit a feature. The dictionary keys are the capabilities. The dictionary values set whether the feature must be present (Boolean true) or omitted (Boolean false).

From the Library of Eric Berna

The current keys are detailed in Table 14-1. Only include those features that your application absolutely requires or cannot support. If your application can provide workarounds, do not add restrictions in this way. Table 14-1 discusses each feature in a positive sense. When using a prohibition rather than a requirement, reverse the meaning—for example, that an autofocus camera or gyro cannot be onboard, or that Game Center access cannot be supported.

Table 14-1 **Required Device Capabilities**

Key	Use
telephony	Application requires the Phone application or uses tel:// URLs.
wifi	Application requires local 802.11-based network access.
sms	Application requires the Messages application or uses sms:// URLs.
still-camera	Application requires an onboard still camera and can use the image picker interface to capture photos from that still camera.
auto-focus-camera	Application requires extra focus capabilities for macro photography or especially sharp images for in-image data detection.
opengles-1	Application requires OpenGL ES 1.1.
camera-flash	Application requires a camera flash feature.
video-camera	Application requires a video-capable camera.
accelerometer	Application requires accelerometer-specific feedback beyond simple `UIViewController` orientation events.
gyroscope	Application requires an onboard gyroscope on the device.
location-services	Application uses Core Location of any kind.
gps	Application uses Core Location and requires the additional accuracy of GPS positioning.
magnetometer	Application uses Core Location and requires heading-related events—that is, the direction of travel. (The magnetometer is the built-in compass.)
gamekit	Application requires Game Center access (iOS 4.1 and later).
microphone	Application uses either built-in microphones or (approved) accessories that provide a microphone.
opengles-1	Application requires OpenGL ES 1.1.
opengles-2	Application requires OpenGL ES 2.0.
armv6	Application is compiled *only* for the armv6 instruction set (3.1 or later).
armv7	Application is compiled *only* for the armv7 instruction set (3.1 or later).
peer-peer	Application uses GameKit peer-to-peer connectivity over Bluetooth (3.1 or later).

From the Library of Eric Berna

For example, consider an application that offers an option for taking pictures when run on a camera-ready device. If the application otherwise works on pre-camera iPod touch units, do not include the still-camera restriction. Instead, use check for camera capability from within the application and present the camera option when appropriate. Adding a still-camera restriction eliminates many early iPod touch (first through third generation) and iPad (first generation) owners from your potential customer pool.

Recipe: Recovering Additional Device Information

Both `sysctl()` and `sysctlbyname()` allow you to retrieve system information. These standard UNIX functions query the operating system about hardware and OS details. You can get a sense of the kind of scope on offer by glancing at the /usr/include/sys/sysctl.h include file on the Macintosh. There you find an exhaustive list of constants that can be used as parameters to these functions.

These constants allow you to check for core information such as the system's CPU frequency, the amount of available memory, and more. Recipe 14-1 demonstrates this. It introduces a `UIDevice` category that gathers system information and returns it via a series of method calls.

You might wonder why this category includes a platform method, when the standard `UIDevice` class returns device models on demand. The answer lies in distinguishing different types of units.

An iPhone 3GS's model is simply "iPhone," as is the model of an iPhone 4. In contrast, this recipe returns a platform value of "iPhone2,1" for the 3GS and "iPhone 3,1" for the 4. This allows you to programmatically differentiate the unit from a first-generation iPhone ("iPhone1,1") or iPhone 3G ("iPhone1,2").

Each model offers distinct built-in capabilities. Knowing exactly which iPhone you're dealing with helps you determine whether that unit likely supports features such as accessibility, GPS, and magnetometers.

Recipe 14-1 **Extending Device Information Gathering**

```
@implementation UIDevice (Hardware)
+ (NSString *) getSysInfoByName:(char *)typeSpecifier
{
    // Recover sysctl information by name
    size_t size;
    sysctlbyname(typeSpecifier, NULL, &size, NULL, 0);

    char *answer = malloc(size);
    sysctlbyname(typeSpecifier, answer, &size, NULL, 0);

    NSString *results = [NSString stringWithCString:answer
        encoding: NSUTF8StringEncoding];
    free(answer);
```

From the Library of Eric Berna

```
    return results;
}

- (NSString *) platform
{
    return [UIDevice getSysInfoByName:"hw.machine"];
}

- (NSUInteger) getSysInfo: (uint) typeSpecifier
{
    size_t size = sizeof(int);
    int results;
    int mib[2] = {CTL_HW, typeSpecifier};
    sysctl(mib, 2, &results, &size, NULL, 0);
    return (NSUInteger) results;
}

- (NSUInteger) cpuFrequency
{
    return [UIDevice getSysInfo:HW_CPU_FREQ];
}

- (NSUInteger) busFrequency
{
    return [UIDevice getSysInfo:HW_BUS_FREQ];
}

- (NSUInteger) totalMemory
{
    return [UIDevice getSysInfo:HW_PHYSMEM];
}

- (NSUInteger) userMemory
{
    return [UIDevice getSysInfo:HW_USERMEM];
}

- (NSUInteger) maxSocketBufferSize
{
    return [UIDevice getSysInfo:KIPC_MAXSOCKBUF];
}
@end
```

Get This Recipe's Code

To get the code used for this recipe, go 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 14 and open the project for this recipe.

From the Library of Eric Berna

Monitoring the iPhone Battery State

You can programmatically keep track of the iPhone's battery level and charge state. The battery level is a floating-point value that ranges between 1.0 (fully charged) and 0.0 (fully discharged). It provides an approximate discharge level that you can use to query before performing operations that put unusual strain on the device.

For example, you might want to caution your user about performing a large series of mathematical computations and suggest that the user plug in to a power source. You retrieve the battery level via this `UIDevice` call. The value returned is produced in 5% increments.

```
NSLog(@"Battery level: %0.2f%",
    [[UIDevice currentDevice] batteryLevel] * 100);
```

The iPhone charge state has four possible values. The unit can be charging (that is, connected to a power source), full, unplugged, and a catchall "unknown." Recover the state using the `UIDevice batteryState` property.

```
NSArray *stateArray = [NSArray arrayWithObjects:
    @"Battery state is unknown",
    @"Battery is not plugged into a charging source",
    @"Battery is charging",
    @"Battery state is full", nil];

UIDevice *device = [UIDevice currentDevice];
NSLog(@"Battery state: %@",
    [stateArray objectAtIndex:device.batteryState]);
```

Don't think of these choices as persistent states. Instead, think of them as momentary reflections of what is actually happening to the device. They are not flags. They are not OR'ed together to form a general battery description. Instead, these values reflect the most recent state change.

You can easily monitor state changes by responding to notifications that the battery state has changed. In this way, you can catch momentary events, such as when the battery finally recharges fully, when the user has plugged in to a power source to recharge, and when the user disconnects from that power source.

To start monitoring, set the `batteryMonitoringEnabled` property to `YES`. During monitoring, the `UIDevice` class produces notifications when the battery state or level changes. The following method subscribes to both notifications. Please note that you can also check these values directly, without waiting for notifications. Apple provides no guarantees about the frequency of level change updates, but as you can tell by testing this recipe, they arrive in a fairly regular fashion.

```
- (void) viewDidLoad
{
    // Enable battery monitoring
    [[UIDevice currentDevice] setBatteryMonitoringEnabled:YES];
```

From the Library of Eric Berna

```
    // Add observers for battery state and level changes
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(checkBattery)
        name:UIDeviceBatteryStateDidChangeNotification
        object:nil];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(checkBattery)
        name:UIDeviceBatteryLevelDidChangeNotification
        object:nil];
}
```

Enabling and Disabling the Proximity Sensor

Proximity is an iPhone-specific feature at this time. The iPod touch and iPad do not offer proximity sensors. Unless you have some pressing reason to hold an iPhone against body parts (or vice versa), enabling the proximity sensor accomplishes little.

When enabled, it has one primary task. It detects whether there's a large object right in front of it. If so, it switches the screen off and sends off a general notification. Move the blocking object away and the screen switches back on. This prevents you from pressing buttons or dialing the phone with your ear when you are on a call. Some poorly designed protective cases keep the iPhone's proximity sensors from working properly.

The Google Mobile application on the App Store used this feature to start a voice recording session. When you held the phone up to your head, it would record your query, sending it off to be interpreted when moved away from your head. The developers didn't mind that the screen blanked because the voice-recording interface did not depend on a visual GUI to operate.

The following methods demonstrate how to work with proximity sensing on the iPhone. This code uses the `UIDevice` class to toggle proximity monitoring and subscribes to `UIDeviceProximityStateDidChangeNotification` to catch state changes. The two states are on and off. When the `UIDevice proximityState` property returns `YES`, the proximity sensor has been activated.

```
- (void) toggle: (id) sender
{
    UIDevice *device = [UIDevice currentDevice];

    // Determine the current proximity monitoring and toggle it
    BOOL isEnabled = device.proximityMonitoringEnabled;
    device.proximityMonitoringEnabled = ! isEnabled;
}

- (void) stateChange: (NSNotificationCenter *) notification
{
    // Log the notifications
    NSLog(@"The proximity sensor %@",
```

From the Library of Eric Berna

```
        [UIDevice currentDevice].proximityState ?
            @"will now blank the screen" :
            @"will now restore the screen");
}

- (void) viewDidLoad
{
    // Add proximity state observer
    [[NSNotificationCenter defaultCenter]
        addObserver:self
        selector:@selector(stateChange)
        name:@"UIDeviceProximityStateDidChangeNotification"
        object:nil];
}
```

Recipe: Using Acceleration to Locate "Up"

The iPhone provides three onboard sensors that measure acceleration along the iPhone's perpendicular axes: left/right (`x`), up/down (`y`), and front/back (`z`). These values indicate the forces affecting the iPhone, from both gravity and user movement. You can get some really neat force feedback by swinging the iPhone around your head (centripetal force) or dropping it from a tall building (freefall). Unfortunately, you might not be able to recover that data after your iPhone becomes an expensive bit of scrap metal.

To subscribe an object to iPhone accelerometer updates, set it as the delegate. The object set as the delegate must implement the `UIAccelerometerDelegate` protocol.

```
[[UIAccelerometer sharedAccelerometer] setDelegate:self]
```

Once assigned, your delegate receives `accelerometer:didAccelerate:` callback messages, which you can track and respond to. The `UIAcceleration` structure sent to the delegate method consists of floating-point values for the x, y, and z axes. Each value ranges from –1.0 to 1.0.

```
float x = acceleration.x;
float y = acceleration.y;
float z = acceleration.z;
```

Recipe 14-2 uses these values to help determine the "up" direction. It calculates the arctangent between the X and Y acceleration vectors, returning the up-offset angle. As new acceleration messages are received, the recipe rotates a `UIImageView` instance with its picture of an arrow, which you can see in Figure 14-1, to point up. The real-time response to user actions ensures that the arrow continues pointing upward, no matter how the user reorients the phone.

From the Library of Eric Berna

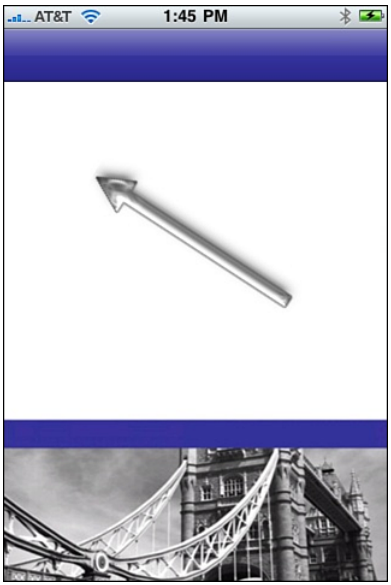


Figure 14-1 A little math recovers the “up” direction by performing an arctan function using the x and y force vectors. In this example, the arrow always points up, no matter how the user reorients the iPhone.

Recipe 14-2 **Catching Acceleration Events**

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
    didAccelerate:(UIAcceleration *)acceleration
{
    // Determine up from the x and y acceleration components
    float xx = -acceleration.x;
    float yy = acceleration.y;
    float angle = atan2(yy, xx);
    [arrow setTransform:
        CGAffineTransformMakeRotation(angle)];
}


- (void) viewDidLoad
{
    // Init the delegate to start catching accelerometer events
    [UIAccelerometer sharedAccelerometer].delegate = self;
}
```

From the Library of Eric Berna

> **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 14 and open the project for this recipe.

Retrieving the Current Accelerometer Angle Synchronously

At times you may want to query the accelerometer without setting yourself up as a full delegate. The following methods, which are meant to be used within a `UIDevice` category, allow you to synchronously return the current device angle along the x/y plane—the front face plane of the iOS device. Accomplish this by entering a new run loop, wait for an accelerometer event, retrieve the current angle from that callback, and then leave the run loop to return that angle.

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
    didAccelerate:(UIAcceleration *)acceleration
{
    float xx = acceleration.x;
    float yy = -acceleration.y;
    device_angle = M_PI / 2.0f - atan2(yy, xx);

    if (device_angle > M_PI)
        device_angle -= 2 * M_PI;

    CFRunLoopStop(CFRunLoopGetCurrent());
}

- (float) orientationAngle
{
    // Supercede current delegate
    id priorDelegate = [UIAccelerometer sharedAccelerometer].delegate;
    [UIAccelerometer sharedAccelerometer].delegate = self;

    // Wait for a reading
    CFRunLoopRun();

    // Restore delegate
    [UIAccelerometer sharedAccelerometer].delegate = priorDelegate;

    return device_angle;
}
```

This is not an approach to use for continuous polling—use the callbacks directly for that. But for an occasional angle query, these methods provide simple and direct access to the current screen angle.

From the Library of Eric Berna

Calculate a Relative Angle

Screen reorientation support means that an interface's relationship to a given device angle must be supported in quarters, one for each possible front-facing screen orientation. As the `UIViewController` automatically rotates its onscreen view, the math needs to catch up to account for those reorientations.

The following method, which is written for use in a `UIDevice` category, calculates angles so the angle remains in synchrony with the device orientation. This creates simple offsets from vertical that match the way the GUI is currently presented.

```
- (float) orientationAngleRelativeToOrientation:
    (UIDeviceOrientation) someOrientation
{
     float dOrientation = 0.0f;
    switch (someOrientation)
    {
        case UIDeviceOrientationPortraitUpsideDown:
            {dOrientation = M_PI; break;}
        case UIDeviceOrientationLandscapeLeft:
            {dOrientation = -(M_PI/2.0f); break;}
        case UIDeviceOrientationLandscapeRight:
            {dOrientation = (M_PI/2.0f); break;}
        default: break;
    }

    float adjustedAngle =
        fmod(self.orientationAngle - dOrientation, 2.0f * M_PI);
    if (adjustedAngle > (M_PI + 0.01f))
        adjustedAngle = (adjustedAngle - 2.0f * M_PI);
    return adjustedAngle;
}
```

This method uses a floating-point modulo to retrieve the difference between the actual screen angle and the interface orientation angular offset to return that all-important vertical angular offset.

Working with Basic Orientation

The `UIDevice` class uses the built-in `orientation` property to retrieve the physical orientation of the device. iOS devices support seven possible values for this property:

- **`UIDeviceOrientationUnknown`**—The orientation is currently unknown.
- **`UIDeviceOrientationPortrait`**—The home button is down.
- **`UIDeviceOrientationPortraitUpsideDown`**—The home button is up.
- **`UIDeviceOrientationLandscapeLeft`**—The home button is to the right.
- **`UIDeviceOrientationLandscapeRight`**—The home button is to the left.

From the Library of Eric Berna

- **UIDeviceOrientationFaceUp**—The screen is face up.
- **UIDeviceOrientationFaceDown**—The screen is face down.

The device can pass through any or all these orientations during a typical application session. Although orientation is created in concert with the onboard accelerometer, these orientations are not tied in any way to a built-in angular value.

iOS offers two built-in macros to help determine if a device orientation enumerated value is portrait or landscape: namely `UIDeviceOrientationIsPortrait()` and `UIDeviceOrientationIsLandscape()`. I find it convenient to extend the `UIDevice` class to offer these tests as built-in device properties.

```
@property (nonatomic, readonly) BOOL isLandscape;
@property (nonatomic, readonly) BOOL isPortrait;

- (BOOL) isLandscape
{
    return UIDeviceOrientationIsLandscape(self.orientation);
}

- (BOOL) isPortrait
{
    return UIDeviceOrientationIsPortrait(self.orientation);
}
```

Your code can subscribe directly to device reorientation notifications. To accomplish this, send `beginGeneratingDeviceOrientationNotifications` to the `currentDevice` singleton. Then add an observer to catch the ensuing `UIDeviceOrientationDidChangeNotification` updates. As you would expect, you can finish listening by calling `endGeneratingDeviceOrientationNotification`.

> **Note**
>
> At the time of writing, iOS does not report a proper orientation when applications are first launched. It updates the orientation only after the device has moved into a new position or `UIViewController` methods kick in. An application launched in portrait orientation may not read as "portrait" until the user moves the device out of and then back into the proper orientation. This bug exists on the simulator as well as on the iPhone device and is easily tested. For a workaround, consider using the angular orientation recovered from recipes in this chapter.

Recipe: Using Acceleration to Move Onscreen Objects

With a bit of programming, the iPhone's onboard accelerometer can make objects "move" around the screen, responding in real time to the way the user tilts the phone. Recipe 14-3 builds an animated butterfly that users can slide across the screen.

The secret to making this work lies in adding what I call a "physics timer" to the program. Instead of responding directly to changes in acceleration, the way Recipe 14-2 did,

From the Library of Eric Berna

the accelerometer callback measures the current forces. It's up to the timer routine to apply those forces to the butterfly over time by changing its frame. Here are some key points to keep in mind:

- As long as the direction of force remains the same, the butterfly accelerates. Her velocity increases, scaled according to the degree of acceleration force in the X or Y direction.
- The `tick` routine, called by the timer, moves the butterfly by adding the velocity vector to the butterfly's origin.
- The butterfly's range is bounded. So when she hits an edge, she stops moving in that direction. This keeps the butterfly onscreen at all times. The `tick` method checks for boundary conditions. For example, if the butterfly hits a vertical edge, she can still move horizontally.
- The butterfly reorients herself so she's always falling "down." This happens by applying a simple rotation transform in the `tick` method. Be careful when using transforms in addition to frame or center offsets. Always reset the math before applying offsets, and then reapply any angular changes. Failing to do so may cause your frames to zoom, shrink, or skew unexpectedly.

Recipe 14-3 **Sliding an Onscreen Object Based on Accelerometer Feedback**

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
    didAccelerate:(UIAcceleration *)acceleration
{
    // Extract the acceleration components
    float xx = -acceleration.x;
    float yy = acceleration.y;

    // Store the most recent angular offset
    mostRecentAngle = atan2(yy, xx);

    // Has the direction changed?
    float accelDirX = SIGN(xvelocity) * -1.0f;
    float newDirX = SIGN(xx);
    float accelDirY = SIGN(yvelocity) * -1.0f;
    float newDirY = SIGN(yy);

    // Accelerate. To increase viscosity lower the additive value
    if (accelDirX == newDirX) xaccel =
        (abs(xaccel) + 0.85f) * SIGN(xaccel);
    if (accelDirY == newDirY) yaccel =
        (abs(yaccel) + 0.85f) * SIGN(yaccel);

    // Apply acceleration changes to the current velocity
    xvelocity = -xaccel * xx;
```

From the Library of Eric Berna

```
    yvelocity = -yaccel * yy;
}

- (void) tick
{
    // Reset the transform before changing position
    butterfly.transform = CGAffineTransformIdentity;

    // Move the butterfly according to the current velocity vector
    CGRect rect = CGRectOffset(butterfly.frame, xvelocity, 0.0f);
    if (CGRectContainsRect(self.view.bounds, rect))
        butterfly.frame = rect;

    rect = CGRectOffset(butterfly.frame, 0.0f, yvelocity);
    if (CGRectContainsRect(self.view.bounds, rect))
        butterfly.frame = rect;

    // Rotate the butterfly independently of position
    butterfly.transform =
        CGAffineTransformMakeRotation(mostRecentAngle + M_PI_2);
}

- (void) initButterfly
{
    CGSize size;

    // Load the animation cells
    NSMutableArray *butterflies = [NSMutableArray array];
    for (int i = 1; i <= 17; i++)
    {
        NSString *fileName = [NSString stringWithFormat:@"bf_%d.png", i];
        UIImage *image = [UIImage imageNamed:fileName];
        size = image.size;
        [butterflies addObject:image];
    }

    // Begin the animation
    butterfly = [[UIImageView alloc]
        initWithFrame:(CGRect){.size=size}];
    [butterfly setAnimationImages:butterflies];
    butterfly.animationDuration = 0.75f;
    [butterfly startAnimating];

    // Set the butterfly's initial speed and acceleration
    xaccel = 2.0f;
    yaccel = 2.0f;
    xvelocity = 0.0f;
    yvelocity = 0.0f;
```

From the Library of Eric Berna

```
    // Add the butterfly
    butterfly.center = RECTCENTER(self.view.bounds);
    [self.view addSubview:butterfly];


    // Activate the accelerometer
    [[UIAccelerometer sharedAccelerometer] setDelegate:self];

    // Start the physics timer
    [NSTimer scheduledTimerWithTimeInterval: 0.03f
        target: self selector: @selector(tick)
        userInfo: nil repeats: YES];
}
```

> **Get This Recipe's Code**
>
> To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from the book, go to the folder for Chapter 14 and open the project for this recipe.

Adding a Little Sparkle

Although Recipe 14–3's butterfly animation draws the eye, you can expand its visual interest by adding a Core Animation emitter to the butterfly's layer. Emitters render particles in real time using a suite of custom properties that you adjust. The following snippet creates a soft orange cloud of dust-like particles that moves with the butterfly and appears to be created by the butterfly's beating wings.

```
// Add an emitter for sparklage
float multiplier = 0.25f;

CAEmitterLayer *emitter = [CAEmitterLayer layer];
emitter.emitterPosition = RECTCENTER(butterfly.bounds);
emitter.emitterMode = kCAEmitterLayerOutline;
emitter.emitterShape = kCAEmitterLayerCircle;
emitter.renderMode = kCAEmitterLayerAdditive;
emitter.emitterSize = CGSizeMake(100 * multiplier, 0);

// Create the emitter cell
CAEmitterCell* particle = [CAEmitterCell emitterCell];
particle.emissionLongitude = M_PI;
particle.birthRate = multiplier * 100.0;
particle.lifetime = multiplier;
particle.lifetimeRange = multiplier * 0.35;
particle.velocity = 180;
particle.velocityRange = 130;
```

From the Library of Eric Berna

```
particle.emissionRange = 1.1;
particle.scaleSpeed = 1.0;
particle.color =
    [[UIColor orangeColor] colorWithAlphaComponent:0.1f].CGColor;
particle.contents =
    (__bridge id)[UIImage imageNamed:@"spark.png"].CGImage;
particle.name = @"particle";

emitter.emitterCells = [NSArray arrayWithObject:particle];
[butterfly.layer addSublayer:emitter];
```

Recipe: Core Motion Basics

The Core Motion framework centralizes motion data processing. Introduced in the iOS 4 SDK, Core Motion supersedes the direct accelerometer access you've just read about. It provides centralized monitoring of three key onboard sensors. These sensors are composed of the gyroscope, which measures device rotation, the magnetometer, which provides a way to measure compass bearings, and the accelerometer, which detects gravitational changes along three axes. A fourth entry point called "device motion" combines all three of these sensors into a single monitoring system.

Core Motion uses raw values from these sensors to create readable measurements, primarily in the form of force vectors. Measurable items include the following properties:

- **Device attitude (`attitude`)**—The device's orientation relative to some frame of reference. The attitude is represented as a triplet of roll, pitch, and yaw angles, each measured in radians.
- **Rotation rate (`rotationRate`)**—The rate at which the device is rotating around each of its three axes. The rotation includes x, y, and z angular velocity values measured in radians per second.
- **Gravity (`gravity`)**—The device's current acceleration vector as imparted by the normal gravitational field. Gravity is measured in g's, along the x, y, and z axes. Each unit represents the standard gravitational force imparted by Earth (namely 32 feet per second per second, or 9.8 meters per second per second).
- **User acceleration (`userAcceleration`)**—The acceleration vector being imparted by the user. Like `gravity`, user acceleration is measured in g's along the x, y, and z axes. When added together, the user vector and the gravity vector represent the total acceleration imparted to the device.
- **Magnetic field (`magneticField`)**—The vector representing the overall magnetic field values in the device's vicinity. The field is measured in microteslas along the x, y, and z axes. A calibration accuracy is also provided, to inform your application of the field measurements quality.

From the Library of Eric Berna

Testing for Sensors

As you read earlier in this chapter, you can use the application's Info.plist file to require or exclude onboard sensors. You can also test in-app for each kind of possible Core Motion support.

```
if (motionManager.gyroAvailable)
    [motionManager startGyroUpdates];

if (motionManager.magnetometerAvailable)
    [motionManager startMagnetometerUpdates];

if (motionManager.accelerometerAvailable)
    [motionManager startAccelerometerUpdates];

if (motionManager.deviceMotionAvailable)
    [motionManager startDeviceMotionUpdates];
```

Starting updates does not produce a delegate callback mechanism like you encountered with the `UIAccelerometer` class. Instead, you are responsible for polling each value or you can use a block-based update mechanism that executes a block that you provide at each update (for example, `startAccelerometerUpdatesToQueue:withHandler:`).

Handler Blocks

Recipe 14-4 adapts Recipe 14-3 for use with Core Motion. The acceleration callback has been moved into a handler block and the x and y values are read from the data's acceleration property. Otherwise, the code remains unchanged. Here, you see the Core Motion basics: A new motion manager is created. It tests for accelerometer availability. It then starts updates using a new operation queue, which persists for the duration of the application run.

The `establishMotionManager` and `shutDownMotionManager` methods allow your application to start up and shut down the motion manager on demand. These methods are called from the application delegate when the application becomes active and when it suspends.

```
- (void) applicationWillResignActive:(UIApplication *)application
{
    [tbvc shutDownMotionManager];
}

- (void) applicationDidBecomeActive:(UIApplication *)application
{
    [tbvc establishMotionManager];
}
```

These methods provide a clean way to shut down and resume motion services in response to the current application state.

From the Library of Eric Berna

Recipe 14-4 **Basic Core Motion**

```
@implementation TestBedViewController
- (void) tick
{
    butterfly.transform = CGAffineTransformIdentity;

    // Move the butterfly according to the current velocity vector
    CGRect rect = CGRectOffset(butterfly.frame, xvelocity, 0.0f);
    if (CGRectContainsRect(self.view.bounds, rect))
        butterfly.frame = rect;

    rect = CGRectOffset(butterfly.frame, 0.0f, yvelocity);
    if (CGRectContainsRect(self.view.bounds, rect))
        butterfly.frame = rect;

    butterfly.transform =
       CGAffineTransformMakeRotation(mostRecentAngle + M_PI_2);
}

- (void) shutDownMotionManager
{
    NSLog(@"Shutting down motion manager");
    [motionManager stopAccelerometerUpdates];
    motionManager = nil;

    [timer invalidate];
    timer = nil;
}

- (void) establishMotionManager
{
    if (motionManager)
        [self shutDownMotionManager];

    NSLog(@"Establishing motion manager");

    // Establish the motion manager
    motionManager = [[CMMotionManager alloc] init];
    if (motionManager.accelerometerAvailable)
        [motionManager
         startAccelerometerUpdatesToQueue:
             [[NSOperationQueue alloc] init]
         withHandler:^(CMAccelerometerData *data, NSError *error)
         {
             // Extract the acceleration components
             float xx = -data.acceleration.x;
```

From the Library of Eric Berna

```
            float yy = data.acceleration.y;
            mostRecentAngle = atan2(yy, xx);

            // Has the direction changed?
            float accelDirX = SIGN(xvelocity) * -1.0f;
            float newDirX = SIGN(xx);
            float accelDirY = SIGN(yvelocity) * -1.0f;
            float newDirY = SIGN(yy);

            // Accelerate. To increase viscosity,
            // lower the additive value
            if (accelDirX == newDirX)
                xaccel = (abs(xaccel) + 0.85f) * SIGN(xaccel);
            if (accelDirY == newDirY)
                yaccel = (abs(yaccel) + 0.85f) * SIGN(yaccel);

            // Apply acceleration changes to the current velocity
            xvelocity = -xaccel * xx;
            yvelocity = -yaccel * yy;
        }];

    // Start the physics timer
    timer = [NSTimer scheduledTimerWithTimeInterval: 0.03f
        target: self selector: @selector(tick)
        userInfo: nil repeats: YES];
}

- (void) initButterfly
{
    CGSize size;

    // Load the animation cells
    NSMutableArray *butterflies = [NSMutableArray array];
    for (int i = 1; i <= 17; i++)
    {
        NSString *fileName =
            [NSString stringWithFormat:@"bf_%d.png", i];
        UIImage *image = [UIImage imageNamed:fileName];
        size = image.size;
        [butterflies addObject:image];
    }

    // Begin the animation
    butterfly = [[UIImageView alloc]
        initWithFrame:(CGRect){.size=size}];
    [butterfly setAnimationImages:butterflies];
    butterfly.animationDuration = 0.75f;
```

From the Library of Eric Berna

```
    [butterfly startAnimating];

    // Set the butterfly's initial speed and acceleration
    xaccel = 2.0f;
    yaccel = 2.0f;
    xvelocity = 0.0f;
    yvelocity = 0.0f;

    // Add the butterfly
    butterfly.center = RECTCENTER(self.view.bounds);
    [self.view addSubview:butterfly];
}

- (void) loadView
{
    [super loadView];
    self.view.backgroundColor = [UIColor whiteColor];
    [self initButterfly];
}
@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 14 and open the project for this recipe.

Recipe: Retrieving and Using Device Attitude

Imagine an iPad sitting on a desk. There's an image displayed on the iPad, which you can bend over and look at. Now imagine rotating that iPad as it lays flat on the desk, but as the iPad moves, the image does not. It maintains a perfect alignment with the world around it. Regardless of how you spin the iPad, the image doesn't "move" as the view updates itself to balance the physical movement. That's how Recipe 14-5 works.

The image adjusts itself however you hold the device. In addition to that flat manipulation, you can pick up the device and orient it in space. If you flip the device and look at it over your head, you see the reversed "bottom" of the image. You can also tilt it along both axes: the one that runs from the home button to the camera, and the other that runs along the surface of the iPad, from the midpoints between the camera and home button. The other axis, the one you first explore, is coming out of the device from its middle, pointing to the air above the device and passing through that middle point to behind it. As you manipulate the device, the image responds to create a virtual still world within that iPad.

Recipe 14-5 demonstrates how to do this with just a few simple geometric transformations. It establishes a motion manager, subscribes to device motion updates, and then applies image transforms based on the roll, pitch, and yaw returned by the motion manager.

From the Library of Eric Berna

Recipe 14-5 **Using Device Motion Updates to Fix an Image in Space**

```
- (void) shutDownMotionManager
{
    NSLog(@"Shutting down motion manager");
    [motionManager stopDeviceMotionUpdates];
    motionManager = nil;
}

- (void) establishMotionManager
{
    if (motionManager)
        [self shutDownMotionManager];

    NSLog(@"Establishing motion manager");

    // Establish the motion manager
    motionManager = [[CMMotionManager alloc] init];
    if (motionManager.deviceMotionAvailable)
        [motionManager
         startDeviceMotionUpdatesToQueue:
             [NSOperationQueue currentQueue]
         withHandler: ^(CMDeviceMotion *motion, NSError *error) {
             CATransform3D transform;
             transform = CATransform3DMakeRotation(
                 motion.attitude.pitch, 1, 0, 0);
             transform = CATransform3DRotate(transform,
                 motion.attitude.roll, 0, 1, 0);
             transform = CATransform3DRotate(transform,
                 motion.attitude.yaw, 0, 0, 1);
             imageView.layer.transform = transform;
     }];
}
```

> **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 14 and open the project for this recipe.

Detecting Shakes Using Motion Events

When the iPhone detects a motion event, it passes that event to the current first responder, the primary object in the responder chain. Responders are objects that can handle events. All views and windows are responders and so is the application object.

The responder chain provides a hierarchy of objects, all of which can respond to events. When an object toward the start of the chain receives an event, that event does not

From the Library of Eric Berna

get passed further down. The object handles it. If it cannot, that event can move on to the next responder.

Objects often become first responder by declaring themselves to be so, via `becomeFirstResponder`. In this snippet, a `UIViewController` ensures that it becomes first responder whenever its view appears onscreen. Upon disappearing, it resigns the first responder position.

```
- (BOOL)canBecomeFirstResponder {
    return YES;
}

// Become first responder whenever the view appears
- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
    [self becomeFirstResponder];
}

// Resign first responder whenever the view disappears
- (void)viewWillDisappear:(BOOL)animated {
    [super viewWillDisappear:animated];
    [self resignFirstResponder];
}
```

First responders receive all touch and motion events. The motion callbacks mirror the touch callbacks discussed in Chapter 8, "Gestures and Touches." They are as follows:

- **`motionBegan:withEvent:`**—This callback indicates the start of a motion event. At the time of writing this book, there was only one kind of motion event recognized: a shake. This may not hold true for the future, so you might want to check the motion type in your code.
- **`motionEnded:withEvent:`**—The first responder receives this callback at the end of the motion event.
- **`motionCancelled:withEvent:`**—As with touches, motions can be cancelled by incoming phone calls and other system events. Apple recommends that you implement all three motion event callbacks (and, similarly, all four touch event callbacks) in production code.

The following snippet shows a pair of motion callback examples. If you test this on a device, you'll notice several things. First, the began and ended events happen almost simultaneously from a user perspective. Playing sounds for both types is overkill. Second, there is a bias toward side-to-side shake detection. The iPhone is better at detecting side-to-side shakes than the front-to-back and up-down versions. Finally, Apple's motion implementation uses a slight lockout approach. You cannot generate a new motion event until a second or so after the previous one was processed. This is the same lockout used by Shake to Shuffle and Shake to Undo events.

From the Library of Eric Berna

```
- (void)motionBegan:(UIEventSubtype)motion
    withEvent:(UIEvent *)event {

    // Play a sound whenever a shake motion starts
    if (motion != UIEventSubtypeMotionShake) return;
    [self playSound:startSound];
}

- (void)motionEnded:(UIEventSubtype)motion withEvent:(UIEvent *)event
{
    // Play a sound whenever a shake motion ends
     if (motion != UIEventSubtypeMotionShake) return;
    [self playSound:endSound];
}
```

Recipe: Detecting Shakes via the Accelerometer

Recipe 14-6 mimics the Apple motion detection system while avoiding the need for the event consumer to be the first responder. It's built on two key parameters: a sensitivity level that provides a threshold that must be met before a shake is acknowledged and a lockout time that limits how often a new shake can be generated.

This `AccelerometerHelper` class stores a triplet of acceleration values. Each value represents a force vector in 3D space. Each successive pair of that triplet can be analyzed to determine the angle between the two vectors. In this example, the angles between the first two items and the second two help determine when a shake happens. This code looks for a pair whose second angle exceeds the first angle. If the angular movement has increased enough between the two (that is, an acceleration of angular velocity, basically a "jerk"), a shake is detected.

The helper generates no delegate callbacks until a second hurdle is passed. A lockout prevents any new callbacks until a certain amount of time expires. This is implemented by storing a trigger time for the last shake event. All shakes that occur before the lockout time expires are ignored. New shakes can be generated after.

Apple's built-in shake detection is calculated with more complex accelerometer data analysis. It analyzes and looks for oscillation in approximately eight to ten consecutive data points, according to a technical expert informed on this topic. Recipe 14-6 provides a less complicated approach, demonstrating how to work with raw acceleration data to provide a computed result from those values.

Recipe 14-6 **Detecting Shakes with the Accelerometer Helper**

```
@implementation AccelerometerHelper
- (id) init
{
    if (!(self = [super init])) return self;

    self.triggerTime = [NSDate date];
```

From the Library of Eric Berna

```
    // Current force vector
    cx = UNDEFINED_VALUE;
    cy = UNDEFINED_VALUE;
    cz = UNDEFINED_VALUE;

    // Last force vector
    lx = UNDEFINED_VALUE;
    ly = UNDEFINED_VALUE;
    lz = UNDEFINED_VALUE;

    // Previous force vector
    px = UNDEFINED_VALUE;
    py = UNDEFINED_VALUE;
    pz = UNDEFINED_VALUE;

    self.sensitivity = 0.5f;
    self.lockout = 0.5f;

    // Start the accelerometer going
    [[UIAccelerometer sharedAccelerometer] setDelegate:self];

    return self;
}

- (void) setX: (float) aValue
{
    px = lx;
    lx = cx;
    cx = aValue;
}

- (void) setY: (float) aValue
{
    py = ly;
    ly = cy;
    cy = aValue;
}

- (void) setZ: (float) aValue
{
    pz = lz;
    lz = cz;
    cz = aValue;
}

- (float) dAngle
{
```

From the Library of Eric Berna

```
    if (cx == UNDEFINED_VALUE) return UNDEFINED_VALUE;
    if (lx == UNDEFINED_VALUE) return UNDEFINED_VALUE;
    if (px == UNDEFINED_VALUE) return UNDEFINED_VALUE;

    // Calculate the dot product of the first pair
    float dot1 = cx * lx + cy * ly + cz * lz;
    float a = ABS(sqrt(cx * cx + cy * cy + cz * cz));
    float b = ABS(sqrt(lx * lx + ly * ly + lz * lz));
    dot1 /= (a * b);

    // Calculate the dot product of the second pair
    float dot2 = lx * px + ly * py + lz * pz;
    a = ABS(sqrt(px * px + py * py + pz * pz));
    dot2 /= a * b;

    // Return the difference between the vector angles
    return acos(dot2) - acos(dot1);
}

- (BOOL) checkTrigger
{
    if (lx == UNDEFINED_VALUE) return NO;

    // Check to see if the new data can be triggered
    if ([[NSDate date] timeIntervalSinceDate:self.triggerTime]
        < self.lockout) return NO;

    // Get the current angular change
    float change = [self dAngle];

    // If we have not yet gathered two samples, return NO
    if (change == UNDEFINED_VALUE) return NO;

    // Does the dot product exceed the trigger?
    if (change > self.sensitivity)
    {
        self.triggerTime = [NSDate date];
        return YES;
    }
    else return NO;
}

- (void)accelerometer:(UIAccelerometer *)accelerometer
    didAccelerate:(UIAcceleration *)acceleration
{
    // Adapt values for a standard coordinate system
    [self setX:-acceleration x];
```

From the Library of Eric Berna

```
    [self setY:acceleration.y];
    [self setZ:acceleration.z];

    // All accelerometer events
    if (self.delegate &&
        [self.delegate respondsToSelector:@selector(ping)])
        [self.delegate performSelector:@selector(ping)];

    // All shake events
    if ([self checkTrigger] && self.delegate &&
        [self.delegate respondsToSelector:@selector(shake)])
    {
        [self.delegate performSelector:@selector(shake)];
    }
}

@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 14 and open the project for this recipe.

Recipe: Using External Screens

There are many ways to use external screens with iOS 5. Take the iPad 2, for example. It offers built-in screen mirroring. Attach a VGA or HDMI cable and your content can be shown on external displays as well as on the built-in screen. Starting with iOS 5, certain devices also allow you to mirror screens wirelessly to Apple TV using AirPlay. These mirroring features are extremely handy, but you're not limited to simply copying content from one screen to another in iOS.

The `UIScreen` class allows you to detect and write to external screens independently. You can treat any connected display as a new window and create content for that display separate from any view you're showing on the primary device screen. You can do this for any wired screen, and starting with iOS 5 and the iPad 2, you can do so wirelessly using AirPlay to Apple TV 2.

Geometry is important. Here's why. iOS devices currently include the 320×480 old-style iPhone displays, the 640×960-pixel Retina display units, and the 1024×768-pixel iPads. Typical composite/component output is produced at 720×480 pixels (480i and 480p), VGA at 1024×768 and 1280×720 (720p), and then there's the higher quality HDMI output available as well.

Add to this the issues of overscan and other target display limitations, and Video Out quickly becomes a geometric challenge. Fortunately, Apple has really responded to this challenge in iOS 5 with some very handy real-world adaptations. Instead of trying to create one-to-one correspondences with the output screen and your built-in device screen,

From the Library of Eric Berna

you can build content based on the available properties of your output display. You just create a window, populate it, and display it.

As a rule, if you intend to develop Video Out applications, make sure you have at least one of each type of cable on-hand (composite, component, VGA, HDMI) as well as an AirPlay-ready iPad, too, so you can thoroughly test on each output configuration. Third-party imported cables do not work, so make sure you purchase Apple-branded items.

Detecting Screens

The `UIScreen` class reports how many screens are connected. You know that an external screen is connected whenever this count goes above 1. The first item in the screens array is always your primary device screen.

```
#define SCREEN_CONNECTED ([UIScreen screens].count > 1)
```

Each screen can report its bounds (that is, its physical dimensions in points) and its screen scale (relating the points to pixels). Two standard notifications allow you to observe when screens have been connected to and disconnected from the device.

```
// Register for connect/disconnect notifications
[[NSNotificationCenter defaultCenter]
    addObserver:self selector:@selector(screenDidConnect:)
    name:UIScreenDidConnectNotification object:nil];
[[NSNotificationCenter defaultCenter]
    addObserver:self selector:@selector(screenDidDisconnect:)
    name:UIScreenDidDisconnectNotification object:nil];
```

Connection means *any* kind of connection, whether by cable or via AirPlay. Whenever you receive an update of this type, make sure you count your screens and adjust your user interface to match the new conditions.

It's your responsibility to set up windows whenever new screens are attached and tear them down upon detach events. Each screen should have its own window to manage content for that output display. Don't hold onto windows upon detaching screens. Let them release and then re-create them when new screens appear.

Retrieving Screen Resolutions

Each screen provides an `availableModes` property. This is an array of resolution objects ordered from least to highest resolution. Each mode has a `size` property indicating a target pixel size resolution. Many screens support multiple modes. For example, a VGA display might have as many as a half dozen or more different resolutions it offers. The number of supported resolutions varies by hardware. There will always be at least one resolution available, but you should be able to offer choices to users when there are more. Program accordingly.

From the Library of Eric Berna

Setting Up Video Out

After retrieving an external screen object from the `[UIScreens screens]` array, query the available modes and select a size to use. As a rule, you can get away with selecting the last mode in the list to always use the highest possible resolution, or the first mode for the lowest resolution.

To start a Video Out stream, create a new `UIWindow` and size it to the selected mode. You'll want to add a new view to that window for drawing on. Then assign the window to the external screen and make it key and visible. This orders the window to display itself and prepares it for use. Once you do that, make the original window key again. This allows the user to continue interacting with the primary screen. Don't skip this step. Nothing makes end users more cranky than discovering their expensive device no longer responds to their touches.

```
self.outwindow = [[UIWindow alloc] initWithFrame:theFrame];
outwindow.screen = secondaryScreen;
[outwindow makeKeyAndVisible];
[delegate.view.window makeKeyAndVisible];
```

Adding a Display Link

Display links are a kind of timer that synchronizes drawing to a display's refresh rate. You can adjust this frame refresh time by changing the display link's `frameInterval` property. It defaults to 1. A higher number slows down the refresh rate. Setting it to 2 halves your frame rate. Create the display link when a screen connects to your device. The `UIScreen` class implements a method that returns a display link object for its screen. You specify the target for the display link and a selector to call.

The display link fires on a regular basis, letting you know when to update the Video Out screen. You can adjust the interval up for less of a CPU load, but you will lose frame rates. This is an important tradeoff, especially for direct manipulation interfaces that require a high level of CPU response on the device side.

The code you see in Recipe 14-7 uses common modes for the run loop, providing the least latency. You `invalidate` your display link when you are done with it, removing it from the run loop.

Overscanning Compensation

The `UIScreen` class allows you to compensate for pixel loss at the edge of display screens by assigning a value to the `overscanCompensation` property. The techniques you can assign are described in Apple's documentation but basically correspond to whether you want to clip content or pad it with black space.

VIDEOkit

Recipe 14-7 introduces VIDEOkit, a basic external screen client. It demonstrates all the features needed to get up and going with wired and wireless external screens. You establish

From the Library of Eric Berna

screen monitoring by calling `startupWithDelegate:`. Pass it the primary view controller whose job it will be to create external content.

The internal `init` method starts listening for screen attach and detach events and builds and tears down windows as needed. An informal delegate method (`updateExternalView:`) is called each time the display link fires. It passes a view that lives on the external window that the delegate can draw onto as needed.

In the sample code that accompanies this recipe, the view controller delegate stores a local color value and uses it to color the external display.

```
- (void) updateExternalView: (UIImageView *) aView
{
    aView.backgroundColor = color;
}

- (void) action: (id) sender
{
    color = [UIColor randomColor];
}
```

Each time the action button is pressed, the view controller generates a new color. When VIDEOkit queries the controller to update the external view, it sets this as the background color. You can see the external screen instantly update to a new random color.

Recipe 14-7 **VIDEOkit**

```
@interface VIDEOkit : NSObject
{
    UIImageView *baseView;
}
@property (nonatomic, weak)   UIViewController *delegate;
@property (nonatomic, strong) UIWindow *outwindow;
@property (nonatomic, strong) CADisplayLink *displayLink;
+ (void) startupWithDelegate: (id) aDelegate;
@end

@implementation VIDEOkit
@synthesize delegate;
@synthesize outwindow, displayLink;

static VIDEOkit *sharedInstance = nil;

- (void) setupExternalScreen
{
    // Check for missing screen
    if (!SCREEN_CONNECTED) return;

    // Set up external screen
    UIScreen *secondaryScreen =
```

From the Library of Eric Berna

```
        [[UIScreen screens] objectAtIndex:1];
    UIScreenMode *screenMode =
        [[secondaryScreen availableModes] lastObject];
    CGRect rect = CGRectMake(0.0f, 0.0f,
        screenMode.size.width, screenMode.size.height);

    // Create new outwindow
    self.outwindow = [[UIWindow alloc] initWithFrame:CGRectZero];
    outwindow.screen = secondaryScreen;
    outwindow.screen.currentMode = screenMode; // Thanks Scott Lawrence
    [outwindow makeKeyAndVisible];
    outwindow.frame = rect;

    // Add base video view to outwindow
    baseView = [[UIImageView alloc] initWithFrame:rect];
    baseView.backgroundColor = [UIColor clearColor];
    [outwindow addSubview:baseView];

    // Restore primacy of main window
    [delegate.view.window makeKeyAndVisible];
}

- (void) updateScreen
{
    // Abort if the screen has been disconnected
    if (!SCREEN_CONNECTED && outwindow)
        self.outwindow = nil;

    // (Re)initialize if there's no output window
    if (SCREEN_CONNECTED && !outwindow)
        [self setupExternalScreen];

    // Abort if we have encountered some weird error
    if (!self.outwindow) return;

    // Go ahead and update
    SAFE_PERFORM_WITH_ARG(delegate,
        @selector(updateExternalView:), baseView);
}

- (void) screenDidConnect: (NSNotification *) notification
{
    NSLog(@"Screen connected");
    UIScreen *screen = [[UIScreen screens] lastObject];

    if (displayLink)
```

From the Library of Eric Berna

```
    {
        [displayLink
            removeFromRunLoop:[NSRunLoop currentRunLoop]
            forMode:NSRunLoopCommonModes];
        [displayLink invalidate];
        self.displayLink = nil;
    }

    // Check for current display link
    if (!displayLink)
    {
        self.displayLink =
            [screen displayLinkWithTarget:self
               selector:@selector(updateScreen)];
        [displayLink addToRunLoop:[NSRunLoop currentRunLoop]
            forMode:NSRunLoopCommonModes];
    }
}

- (void) screenDidDisconnect: (NSNotification *) notification
{
    NSLog(@"Screen disconnected.");
    if (displayLink)
    {
        [displayLink
            removeFromRunLoop:[NSRunLoop currentRunLoop]
            forMode:NSRunLoopCommonModes];
        [displayLink invalidate];
        self.displayLink = nil;
    }
}

- (id) init
{
    if (!(self = [super init])) return self;

    // Handle output window creation
    if (SCREEN_CONNECTED)
        [self screenDidConnect:nil];

    // Register for connect/disconnect notifications
    [[NSNotificationCenter defaultCenter]
        addObserver:self selector:@selector(screenDidConnect:)
        name:UIScreenDidConnectNotification object:nil];
    [[NSNotificationCenter defaultCenter]
       addObserver:self selector:@selector(screenDidDisconnect:)
       name:UIScreenDidDisconnectNotification object:nil];
```

From the Library of Eric Berna

```
    return self;
}

- (void) dealloc
{
    [self screenDidDisconnect:nil];
    self.outwindow = nil;
}

+ (VIDEOkit *) sharedInstance
{
    if (!sharedInstance)
        sharedInstance = [[self alloc] init];
    return sharedInstance;
}

+ (void) startupWithDelegate: (id) delegate
{
    [[self sharedInstance] setDelegate:delegate];
}
@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 14 and open the project for this recipe.

One More Thing: Checking for Available Disk Space

The `NSFileManager` class allows you to determine how much space is free on the iPhone as well as how much space is provided on the device as a whole. Listing 14-1 demonstrates how to check for these values and show the results using a friendly comma-formatted string. The values returned represent the free space in bytes.

Listing 14-1 **Recovering File System Size and File System Free Size**

```
- (NSString *) commaFormattedStringWithLongLong: (long long) num
{
    // Produce a properly formatted number string
    // Alternatively use NSNumberFormatter
    if (num < 1000)
        return [NSString stringWithFormat:@"%d", num];
    return    [[self commasForNumber:num/1000]
```

From the Library of Eric Berna

```
        stringByAppendingFormat:@",%03d", (num % 1000)];
}

- (void) action: (UIBarButtonItem *) bbi
{
    NSFileManager *fm = [NSFileManager defaultManager];
    NSDictionary *fattributes =
        [fm fileSystemAttributesAtPath:NSHomeDirectory()];
    NSLog(@"System space: %@",
        [self commaFormattedStringWithLongLong:[[fattributes
        objectForKey:NSFileSystemSize] longLongValue]]);
    NSLog(@"System free space: %@",
        [self commasForNumber:[[fattributes
        objectForKey:NSFileSystemFreeSize] longLongValue]]);
}
```

Summary

This chapter introduced core ways to interact with an iPhone device. You saw how to recover device info, check the battery state, and subscribe to proximity events. You learned how to differentiate the iPod touch from the iPhone and iPad and determine which model you're working with. You discovered the accelerometer and saw it in use through several examples, from the simple "finding up" to the more complex shake detection algorithm. You jumped into Core Motion and learned how to create update blocks to respond to device events in real time. Finally, you saw how to add external screen support to your applications. Here are a few parting thoughts about the recipes you just encountered:

- The iPhone's accelerometer provides a novel way to complement its touch-based interface. Use acceleration data to expand user interactions beyond the "touch here" basics and to introduce tilt-aware feedback.
- Low-level calls can be App Store friendly. They don't depend on Apple APIs that may change based on the current firmware release. UNIX system calls may seem daunting, but many are fully supported by the iOS device family.
- Remember device limitations. You may want to check for free disk space before performing file-intensive work and for battery charge before running the CPU at full steam.
- Dive into Core Motion. The real-time device feedback it provides is the foundation for integrating iOS devices into real-world experiences.

From the Library of Eric Berna

- Now that AirPlay has cut the cord for external display tethering, you can use Video Out for many more exciting projects than you might have previously imagined. AirPlay and external video screens mean you can transform your iOS device into a remote control for games and utilities that display on big screens and are controlled on small ones.
- When submitting to iTunes, use your Info.plist file to determine which device capabilities are required. iTunes uses this list of required capabilities to determine whether an application can be downloaded to a given device and run properly on that device.

From the Library of Eric Berna

15

Networking

As an Internet-connected device, the iPhone and its other iOS family members are particularly well suited to retrieving remote data and accessing web-based services. Apple has lavished the platform with a solid grounding in all kinds of network computing and its supporting technologies. The iOS SDK handles sockets, password keychains, XML processing, and more. This chapter surveys common techniques for network computing, offering recipes that simplify day-to-day tasks. You read about checking the network status, monitoring that status for changes, and testing site reachability. You also learn how to download resources asynchronously and how to respond to authentication challenges. By the time you finish this chapter, you'll have discovered how to build an FTP client, a custom iPhone-based web browser, and more.

Checking Your Network Status

Networked applications need a live connection to communicate with the Internet or other nearby devices. Applications should know whether that connection exists before reaching out to send or retrieve data. Checking the network status lets the application communicate with users and explain why certain functions might be disabled.

Apple has and will reject applications that do not check the network status before providing download options to the user. Apple reviewers are trained to check whether you properly notify the user, especially in the case of network errors. Always verify your network status and alert the user accordingly.

Apple also may reject applications based on "excessive data usage." If you plan to stream large quantities of data in your application, such as voice or data, you'll want to test for the current connection type. Provide lower-quality data streams for users on a cell network connection and higher-quality data for users with a Wi-Fi connection. Apple has had little tolerance for applications that place high demands on cell network data. Keep in mind that unlimited data has given way to metered accounts. You can quickly alienate your users as well as Apple.

From the Library of Eric Berna

iOS can currently test for the following configuration states: some (that is, any kind of) network connection available, Wi-Fi available, and cell service available. There are no App Store–safe APIs that allow the iPhone to test for Bluetooth connectivity at this time (although you can limit your application to run only on Bluetooth-enabled devices), nor can you check to see whether a user is roaming before offering data access.

The System Configuration framework offers many networking aware functions. Among these, `SCNetworkReachabilityCreateWithAddress` checks whether an IP address is reachable. Listing 15-1 shows a simple example of this test in action.

It provides a basic detector that determines whether your device has outgoing connectivity, which it defines as having both access and a live connection. This method, based on Apple sample code, returns `YES` when the network is available and `NO` otherwise. The flags used here indicate both that the network is reachable (`kSCNetworkFlagsReachable`) and that no further connection is required (`kSCNetworkFlagsConnectionRequired`). Other flags you may use are as follows:

- **`kSCNetworkReachabilityFlagsIsWWAN`**—Tests whether your user is using the carrier's network or local Wi-Fi. When available, the network can be reached via EDGE, GPRS, or another cell connection. That means you might want to use lightweight versions of your resources (for example, smaller versions of images) due to the connection's constricted bandwidth.
- **`kSCNetworkReachabilityFlagsConnectionOnTraffic`**—Specifies that addresses can be reached with the current network configuration but that a connection must first be established. Any actual traffic will initiate the connection.
- **`kSCNetworkReachabilityFlagsIsDirect`**—Tells you whether the network traffic goes through a gateway or arrives directly.

To confirm that connectivity code works, it is best evaluated on an iPhone. iPhones provide both cell and Wi-Fi support, allowing you to confirm that the network remains reachable when using a WWAN connection. Test out this code by toggling Wi-Fi and Airplane mode off and on in the iPhone's Setting app. Be aware that there's sometimes a slight delay when checking for network reachability, so design your applications accordingly. Let the user know what your code is up to during the check.

Listing 15-1 **Testing a Network Connection**

```
- (BOOL) connectedToNetwork
{
    // Create zero address
    struct sockaddr_in zeroAddress;
    bzero(&zeroAddress, sizeof(zeroAddress));
    zeroAddress.sin_len = sizeof(zeroAddress);
    zeroAddress.sin_family = AF_INET;

    // Recover reachability flags
    SCNetworkReachabilityRef defaultRouteReachability =
```

From the Library of Eric Berna

```
        SCNetworkReachabilityCreateWithAddress(NULL,
        (struct sockaddr *)&zeroAddress);
    SCNetworkReachabilityFlags flags;

    BOOL didRetrieveFlags =
        SCNetworkReachabilityGetFlags(
        defaultRouteReachability, &flags);
    CFRelease(defaultRouteReachability);

    if (!didRetrieveFlags)
    {
        printf("Could not recover network flags\n");
        return NO;
    }

    BOOL isReachable = flags & kSCNetworkFlagsReachable;
    BOOL needsConnection = flags & kSCNetworkFlagsConnectionRequired;
    return (isReachable && !needsConnection) ? YES : NO;
}
```

Recipe: Extending the `UIDevice` Class for Reachability

The `UIDevice` class provides information about the current device in use, such as its battery state, model, orientation, and so forth. Adding reachability seems like a natural extension for a class whose purpose is to report device state. Recipe 15-1 defines a `UIDevice` category called Reachability. It hides calls to the System Configuration framework and provides a simple way to check on the current network state. You can ask if the network is active, and whether it is using cell service or Wi-Fi.

Most connectivity-checking solutions assume that a connected device, whose connection is not provided by WWAN cell service, has Wi-Fi connectivity. This is an assumption that may not continue to hold true should Apple open up Bluetooth services to the SDK. Recipe 15-1 uses a direct Wi-Fi-checking solution developed by Matt Brown, a software developer and a fan of the first edition of this book. It has been extended slightly in this edition due to feedback from Johannes Rudolph, who figured out how to check for the new iPhone personal hotspot. It applies low-level (but SDK-friendly) calls to retrieve the local Wi-Fi IP address feature.

Note that this class uses a slightly different network check than Listing 15-1, again one inspired by Apple sample code. Use the `ignoresAdHocWiFi` Boolean to limit network checks. When this is enabled, the recipe won't return a success on detecting an ad hoc Wi-Fi connection.

From the Library of Eric Berna

Recipe 15-1 **Extending `UIDevice` for Reachability**

```
SCNetworkConnectionFlags connectionFlags;
SCNetworkReachabilityRef reachability;

// Matt Brown's get WiFi IP address solution
// Bridge check suggested by Johannes Rudolph

 - (NSString *) localWiFiIPAddress
{
    BOOL success;
    struct ifaddrs * addrs;
    const struct ifaddrs * cursor;

    success = getifaddrs(&addrs) == 0;
    if (success) {
        cursor = addrs;
        while (cursor != NULL) {

            // The second test avoids the loopback address
            if (cursor->ifa_addr->sa_family == AF_INET &&
                (cursor->ifa_flags & IFF_LOOPBACK) == 0)
            {
                NSString *name =
                    [NSString stringWithUTF8String:
                        cursor->ifa_name];

                // Wi-Fi adapter or Personal Hotspot bridge adapter
                if ([name isEqualToString:@"en0"] ||
                    [name isEqualToString:@"bridge0"])
                    return [NSString stringWithUTF8String:
                        inet_ntoa(((struct sockaddr_in *)
                            cursor->ifa_addr)->sin_addr)];
            }
            cursor = cursor->ifa_next;
        }
        freeifaddrs(addrs);
    }
    return nil;
}

// Perform basic reachability tests
- (void) pingReachabilityInternal
{
    if (!reachability)
    {
```

From the Library of Eric Berna

```
        BOOL ignoresAdHocWiFi = NO; // thanks to Apple
        struct sockaddr_in ipAddress;
        bzero(&ipAddress, sizeof(ipAddress));
        ipAddress.sin_len = sizeof(ipAddress);
        ipAddress.sin_family = AF_INET;
        ipAddress.sin_addr.s_addr =
            htonl(ignoresAdHocWiFi ?
                INADDR_ANY : IN_LINKLOCALNETNUM);

        reachability = SCNetworkReachabilityCreateWithAddress(
            kCFAllocatorDefault, (struct sockaddr *)&ipAddress);
        CFRetain(reachability);
    }

    // Recover reachability flags
    BOOL didRetrieveFlags = SCNetworkReachabilityGetFlags(
        reachability, &connectionFlags);
    if (!didRetrieveFlags)
        printf("Could not recover network reachability flags\n");
}

// Is there a network available to connect to?
- (BOOL) networkAvailable
{
    [self pingReachabilityInternal];
    BOOL isReachable =
        ((connectionFlags & kSCNetworkFlagsReachable) != 0);
    BOOL needsConnection = ((connectionFlags &
        kSCNetworkFlagsConnectionRequired) != 0);
    return (isReachable && !needsConnection) ? YES : NO;
}

// Is there an active personal hotspot?
// Thanks to Johannes Rudolph
- (BOOL) activePersonalHotspot
{
    // Personal hotspot is fixed to 172.20.10.x
    NSString* localWifiAddress = [self localWiFiIPAddress];
    return (localWifiAddress != nil &&
        [localWifiAddress hasPrefix:@"172.20.10"]);
}

// Connected to cell data?
- (BOOL) activeWWAN
{
    if (![self networkAvailable]) return NO;
```

From the Library of Eric Berna

```
    return ((connectionFlags &
        kSCNetworkReachabilityFlagsIsWWAN) != 0);
}

// Connected to local network?
- (BOOL) activeWLAN
{
    return ([[UIDevice currentDevice]
        localWiFiIPAddress] != nil);
}
```

> **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 15 and open the project for this recipe.

Scanning for Connectivity Changes

Connectivity state may change while an application is running. Checking once at application launch usually isn't enough for an application that depends on data connections throughout its lifetime. You may want to alert the user that a network connection was lost—or could finally be established.

Listing 15-2 addresses this challenge by extending the `UIDevice` reachability category to monitor network changes. It provides a pair of methods that allow you to schedule and unschedule reachability watchers, observers who must be notified when the connectivity state changes. It builds a callback that messages a watcher object when that state changes. The monitor is scheduled on the current run loop and runs asynchronously. Upon detecting a change, the callback function triggers.

Listing 15-2's callback function redirects itself to a specific delegate method, `reachabilityChanged`, which must be implemented by its watcher. That watcher object can then query `UIDevice` via this category for the current flags and network state.

The method that schedules the watcher assigns the delegate as its parameter. Here's a trivial case of how that might be implemented skeletally. In real-world deployment, you'll want to update your GUI and available choices to match the availability (or lack thereof) of network-only features.

```
- (void) reachabilityChanged
{
    [self showAlert:@"Reachability has changed."];
}

- (void) viewDidLoad
{
      [UIDevice scheduleReachabilityWatcher:self];
}
```

From the Library of Eric Berna

Be prepared for multiple callbacks. Your application will generally receive one callback at a time for each kind of state change (that is, when the cellular data connection is established or released) or when Wi-Fi is established or lost. Your user's connectivity settings (especially remembering and logging in to known Wi-Fi networks) will affect the kind and number of callbacks you may have to handle.

Be sure to inform your user when connectivity changes as well as update your interface to mirror the current state. You might want to disable buttons or menu items that depend on network access when that access disappears. Providing an alert of some kind lets the user know why the GUI has updated.

Listing 15-2 **Monitoring Connectivity Changes**

```
// For each callback, ping the watcher
static void ReachabilityCallback(
    SCNetworkReachabilityRef target,
    SCNetworkConnectionFlags flags, void* info)
{
    @autoreleasepool {
        id watcher = (__bridge id) info;
        SEL changed = @selector(reachabilityChanged);
        if ([watcher respondsToSelector: changed])
            [watcher performSelector:changed];
    }
}

// Schedule watcher into the run loop
- (BOOL) scheduleReachabilityWatcher: (id) watcher
{
    SEL changed = @selector(reachabilityChanged);
    if (![watcher respondsToSelector:changed])
    {
        NSLog(@"Error: Watcher must implement reachabilityChanged");
        return NO;
    }

    [self pingReachabilityInternal];

    SCNetworkReachabilityContext context =
        {0, (__bridge void *)watcher, NULL, NULL, NULL};
    if(SCNetworkReachabilitySetCallback(reachability,
        ReachabilityCallback, &context))
    {
        if(!SCNetworkReachabilityScheduleWithRunLoop(
            reachability, CFRunLoopGetCurrent(),
            kCFRunLoopCommonModes))
        {
```

From the Library of Eric Berna

```
            NSLog(@"Error: Could not schedule reachability");
            SCNetworkReachabilitySetCallback(reachability, NULL, NULL);
            return NO;
        }
    }
    else
    {
        NSLog(@"Error: Could not set reachability callback");
        return NO;
    }
    return YES;
}

// Remove the watcher
- (void) unscheduleReachabilityWatcher
{
    SCNetworkReachabilitySetCallback(reachability, NULL, NULL);
    if (SCNetworkReachabilityUnscheduleFromRunLoop(
        reachability, CFRunLoopGetCurrent(),
        kCFRunLoopCommonModes))
        NSLog(@"Success. Unscheduled reachability");
    else
        NSLog(@"Error: Could not unschedule reachability");

    CFRelease(reachability);
    reachability = nil;
}
```

Recovering IP and Host Information

In the day-to-day world of iPhone network programming, certain tasks come up over and over again, particularly those dealing with recovering the local iPhone IP address information and working with address structures. Listing 15-3 provides a handful of utilities, several based on Apple sample code, that help you manage these tasks.

As with Recipe 15-1 and Listing 15-2, these methods are wrapped into the `UIDevice` class as a category extension. They are, again, all implemented as class methods because their utility is not tied to any particular object instance. The methods in this recipe are as follows:

- A pair of methods (`stringFromAddress:` and `addressFromString:`) helps you convert address structures to and from string representations. These integrate well with the `NSNetService` class, allowing you to convert `sockaddr` structures into `NSString` instances and back.

From the Library of Eric Berna

- The `hostname` method returns the host name for the current device. This method observes a small iPhone quirk. The simulator normally appends the `.local` domain to the current host name. The iPhone does not. This routine forces the host name into Mac-style compliance.
- Use `getIPAddressForHost:` to look up an address for a given host name. These calls are blocking, and they take a certain amount of time to return (especially for nonexistent hosts). Use them judiciously, preferably on a secondary thread or via an `NSOperationQueue`.
- The `localIPAddress` method looks up the host's address and returns it as a string. Like `getIPAddressForHost:`, this method uses `gethostbyname()` to convert a host name into an IP address.
- A final method, `whatismyipdotcom`, helps move past a local LAN to determine a cable, DSL, or similar IP address. It sends out a call to the whatismyip.com website, which returns the connection IP address. This method is run synchronously, so it blocks. You should always make sure you are connected to the network before attempting to call this method and invoke it on a secondary thread.

Listing 15-3 **IP and Host Utilities**

```
// Return a string that represents the socket address
+ (NSString *) stringFromAddress:
    (const struct sockaddr *) address
{
    if (address && address->sa_family == AF_INET)
    {
        const struct sockaddr_in* sin =
            (struct sockaddr_in *) address;
        return [NSString stringWithFormat:%d",
            [NSString stringWithUTF8String:
                inet_ntoa(sin->sin_addr)],
            ntohs(sin->sin_port)];
    }

    return nil;
}

// Retrieve socket address from a string representation
+ (BOOL)addressFromString:(NSString *)IPAddress
    address:(struct sockaddr_in *)address
{
    if (!IPAddress || ![IPAddress length]) return NO;

    memset((char *) address, sizeof(struct sockaddr_in), 0);
    address->sin_family = AF_INET;
    address->sin_len = sizeof(struct sockaddr_in);
```

From the Library of Eric Berna

```
    int conversionResult = inet_aton([IPAddress UTF8String],
        &address->sin_addr);
    if (conversionResult == 0)
    {
        NSAssert1(conversionResult != 1,
           @"Failed to convert IP address: %@", IPAddress);
        return NO;
    }

    return YES;
}

// Produce the local hostname
- (NSString *) hostname
{
    char baseHostName[256]; // Thanks, Gunnar Larisch
    int success = gethostname(baseHostName, 255);
    if (success != 0) return nil;
    baseHostName[255] = '\0';

#if TARGET_IPHONE_SIMULATOR
     return [NSString stringWithFormat:@"%s",
         baseHostName];
#else
    return [NSString stringWithFormat:@"%s.local",
        baseHostName];
#endif
}

// Associate a host name with an IP address
- (NSString *) getIPAddressForHost: (NSString *) theHost
{
    struct hostent *host =
        gethostbyname([theHost UTF8String]);
    if (!host) {herror("resolv"); return NULL; }
    struct in_addr **list = (
        struct in_addr **)host->h_addr_list;
    NSString *addressString =
        [NSString stringWithCString: inet_ntoa(*list[0])
            encoding:NSUTF8StringEncoding];
    return addressString;
}

// Return the local IP address
- (NSString *) localIPAddress
{
    struct hostent *host =
```

From the Library of Eric Berna

```
        gethostbyname([[self hostname] UTF8String]);
    if (!host) {herror("resolv"); return nil;}
    struct in_addr **list =
        (struct in_addr **)host->h_addr_list;
    return [NSString stringWithCString: inet_ntoa(*list[0])
        encoding:NSUTF8StringEncoding];
}

// Call on whatismyip.com. Be aware that this URL periodically
// changes. Do not use in production code.
- (NSString *) whatismyipdotcom
{
    NSError *error;
    NSURL *ipURL = [NSURL URLWithString:
        @"http://automation.whatismyip.com/n09230945.asp"];
    NSString *ip = [NSString stringWithContentsOfURL:ipURL
        encoding:NSUTF8StringEncoding error:&error];
    return ip ? ip : [error localizedDescription];
}
```

Using Queues for Blocking Checks

Use operation queues when working with blocking network calls. Listing 15-4 offers two use-case examples. In the first, a method runs a series of basic connectivity tests. The second method looks up IP addresses for Google and Amazon.

The connectivity method checks whether the iPhone or iPad 3G is connected to a cellular network (checked by Recipe 15-1's `activeWWAN` method). If so, it launches a connection to whatismyip.com. It starts this process by initiating the application's network activity indicator. This produces active feedback in the device's status bar for an ongoing network operation.

Next, it creates a new operation queue, adding an execution block that retrieves the address using the blocking network call. Upon completion, it performs yet another block, but this time it does so on the main thread because GUI updates are not thread-safe. The final steps update the primary text view by logging the results and stopping the network activity indicator.

The second method operates on the same touch points but with slight changes. Instead of limiting itself to cellular connectivity, it only requires a general network connection (`networkAvailable`). Like the other method, it places all its blocking calls into an `NSOperationQueue` block and updates its GUI on the main thread after these calls finish.

The approach demonstrated by these methods works best for calls that can be run asynchronously but shouldn't block the main GUI. If you do need the GUI to wait during operations, be sure to put up some kind of "Please wait" dialog, as demonstrated in Chapter 13, "Alerting the User."

From the Library of Eric Berna

Listing 15-4 **Using `NSOperationQueue` with Blocking Calls**

```
// Perform a suite of basic connectivity tests
- (void) runTests
{
    UIDevice *device = [UIDevice currentDevice];
    [self log:@"\n\n"];
    [self log:@"Current host: %@", [device hostname]];
    [self log: %@", [device localIPAddress]];
    [self log: %@", [device localWiFiIPAddress]];
    [self log: %@", [device localWiFiIPAddresses]];

    [self log:@"Network available?: %@",
        [device networkAvailable] ? @"Yes" : @"No"];
    [self log:@"Active WLAN?: %@",
        [device activeWLAN] ? @"Yes" : @"No"];
    [self log:@"Active WWAN?: %@",
        [device activeWWAN] ? @"Yes" : @"No"];
    [self log:@"Active hotspot?: %@",
        [device activePersonalHotspot] ? @"Yes" : @"No"];

    if (![device activeWWAN]) return;

    [self log:@"Contacting whatismyip.com"];

    [[UIApplication sharedApplication]
        setNetworkActivityIndicatorVisible:YES];
    [[[NSOperationQueue alloc] init] addOperationWithBlock:
    ^{
        NSString *results = [device whatismyipdotcom];
        [[NSOperationQueue mainQueue] addOperationWithBlock:
        ^{
            [self log:@"IP Address: %@", results];
            [[UIApplication sharedApplication]
                 setNetworkActivityIndicatorVisible:NO];
         }];
    }];
}

// Look up addresses for Google and Amazon
- (void) checkAddresses
{
    UIDevice *device = [UIDevice currentDevice];
    if (![device networkAvailable]) return;
    [self log:@"Checking IP Addresses"];
```

From the Library of Eric Berna

```
    [[UIApplication sharedApplication]
        setNetworkActivityIndicatorVisible:YES];
    [[[NSOperationQueue alloc] init] addOperationWithBlock:
    ^{
         NSString *google = [device getIPAddressForHost:
             @"www.google.com"];
         NSString *amazon = [device getIPAddressForHost:
             @"www.amazon.com"];
         [[NSOperationQueue mainQueue] addOperationWithBlock:
         ^{
             [self log: %@", google];
             [self log: %@", amazon];
             [[UIApplication sharedApplication]
                setNetworkActivityIndicatorVisible:NO];
         }];
    }];
}
```

Checking Site Availability

After recovering a site's IP address, use the `SCNetworkReachabilityCreateWithAddress()` function to check its availability. Pass a `sockaddr` record populated with the site's IP address, and then check for the `kSCNetworkFlagsReachable` flag when the function returns. Listing 15-5 shows the site checking the `hostAvailable:` method. It returns `YES` or `NO`, indicating whether the host can be reached. Some sample checks are shown in Figure 15-1.



Figure 15-1 Use standard network calls to check whether sites are available.

From the Library of Eric Berna

This kind of check is synchronous and will block interaction until the method returns. Use operation queues like the ones shown in Listing 15-4 in real-world deployment. During testing, this recipe took slightly under 3 seconds to run all six tests, including the "notverylikely.com" test, which was included to force a lookup failure.

Listing 15-5 **Checking Site Reachability**

```
- (BOOL) hostAvailable: (NSString *) theHost
{

    NSString *addressString =
        [self getIPAddressForHost:theHost];
    if (!addressString)
    {
        NSLog(@"Error recovering IP address from host name");
        return NO;
    }

    struct sockaddr_in address;
    BOOL gotAddress =
        [UIDevice addressFromString:addressString address:&address];

    if (!gotAddress)
    {
        NSLog(@"Error recovering sockaddr address from %@",
            addressString);
        return NO;
    }

    SCNetworkReachabilityRef defaultRouteReachability =
        SCNetworkReachabilityCreateWithAddress(
            NULL, (struct sockaddr *)&address);
    SCNetworkReachabilityFlags flags;

    BOOL didRetrieveFlags = SCNetworkReachabilityGetFlags(
        defaultRouteReachability, &flags);
    CFRelease(defaultRouteReachability);

    if (!didRetrieveFlags)
    {
        NSLog(@"Error. Could not recover reachability flags");
        return NO;
    }

    BOOL isReachable = flags & kSCNetworkFlagsReachable;
    return isReachable ? YES : NO;;
}
```

From the Library of Eric Berna

```
#define CHECK(SITE) [self log:@"• %@ : %@", SITE, [device hostAvailable:SITE] ?
@"available" : @"not available"];

- (void) checkSites
{
    UIDevice *device = [UIDevice currentDevice];
    NSDate *date = [NSDate date];
    CHECK(@"www.google.com");
    CHECK(@"www.ericasadun.com");
    CHECK(@"www.notverylikely.com");
    CHECK(@"192.168.0.108");
    CHECK(@"pearson.com");
    CHECK(@"www.pearson.com");
    [self log:@"Elapsed time: %0.1f",
        [[NSDate date] timeIntervalSinceDate:date]];
}
```

Synchronous Downloads

Synchronous downloads allow you to request data from the Internet, wait until that data is received, and then move on to the next step in your application. The following snippet is both synchronous and blocking. You will not return from this method until all the data is received.

```
- (UIImage *) imageFromURLString: (NSString *) urlstring
{
    // This is a blocking call
    return [UIImage imageWithData:[NSData
        dataWithContentsOfURL:[NSURL URLWithString:urlstring]]];
}
```

The `NSURLConnection` class provides a more general download approach than class-specific URL initialization. It provides both synchronous and asynchronous downloads, the latter provided by a series of delegate callbacks. Recipe 15-2 focuses on the simpler, synchronous approach. It begins by creating an `NSMutableURLRequest` with the URL of choice. That request is sent synchronously using the `NSURLConnection` class.

```
NSMutableURLRequest *theRequest =
    [NSMutableURLRequest requestWithURL:url];
NSData* result = [NSURLConnection sendSynchronousRequest:
    theRequest returningResponse:&response error:&error];
```

This call blocks until the request fails (returning `nil`, and an error is produced) or the data finishes downloading.

Recipe 15-2 performs the synchronous request using an operation queue. This allows the main thread to keep executing without blocking. To accommodate this, the `log:` method, which provides updates through the download process, has been modified for

From the Library of Eric Berna

thread safety. Instead of updating the text view directly, the method uses the main (GUI-safe) queue.

```
// Perform GUI update on main thread
[[NSOperationQueue mainQueue] addOperationWithBlock:^(){
    textView.text = log;
}];
```

This example allows testing with three predefined URLs. There's one that downloads a short (3MB) movie, another using a larger (35MB) movie, and a final fake URL to test errors. The movies are sourced from the Internet Archive (archive.org), which provides a wealth of public domain data.

Some Internet providers produce a valid web page, even when given a completely bogus URL. The data returned in the response parameter helps you determine when this happens. This parameter points to an `NSURLResponse` object. It stores information about the data returned by the URL connection. These parameters include expected content length and a suggested filename. Should the expected content length be less than zero, that's a good clue that the provider has returned data that does not match up to your expected request.

```
[self log:@"Response expects %d bytes",
   response.expectedContentLength];
```

As you can see in Recipe 15-2, integrating large downloads into the main application GUI is messy, even with a secondary thread. It also provides no inter-download feedback. Recipe 15-3 addresses both these issues by using asynchronous downloads with delegate callbacks.

Recipe 15-2 **Synchronous Downloads**

```
// Large Movie (35 MB)
#define LARGE_URL @"http://www.archive.org/download/\
    BettyBoopCartoons/Betty_Boop_More_Pep_1936_512kb.mp4"

// Short movie (3 MB)
#define SMALL_URL @"http://www.archive.org/download/\
    Drive-inSaveFreeTv/Drive-in--SaveFreeTv_512kb.mp4"

// Invalid movie
#define FAKE_URL \
    @"http://www.thisisnotavalidurlforthisexample.com"

// Where to save file
#define DEST_PATH [NSHomeDirectory() \
    stringByAppendingString:@"/Documents/Movie.mp4"]

// Remove observer and the player view
-(void)myMovieFinishedCallback: (NSNotification*)aNotification
```

From the Library of Eric Berna

```
{
    [[NSNotificationCenter defaultCenter] removeObserver:self];
    [movieController.view removeFromSuperview];
    movieController = nil;
}

- (void) downloadFinished
{
    // Restore GUI
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Go", @selector(go));
    seg.enabled = YES;
    [[UIApplication sharedApplication]
        setNetworkActivityIndicatorVisible:NO];

    if (!success)
    {
        [self log:@"Failed download"];
        return;
    }

    // Prepare a player
    movieController = [[MPMoviePlayerController alloc]
        initWithContentURL:[NSURL fileURLWithPath:DEST_PATH]];
    movieController.view.frame = self.view.bounds;
    movieController.controlStyle = MPMovieControlStyleFullscreen;
    [self.view addSubview:movieController.view];

    // Listen for the movie to finish
    [[NSNotificationCenter defaultCenter]
        addObserver:self
        selector:@selector(myMovieFinishedCallback:)
        name:MPMoviePlayerPlaybackDidFinishNotification
        object:movieController];

    // Start playback
    [movieController play];
}

// Call this on an operation queue
- (void) getData: (NSURL *) url
{
    [self log:@"Starting download"];

    NSDate *startDate = [NSDate date];
```

From the Library of Eric Berna

```
    NSMutableURLRequest *theRequest =
        [NSMutableURLRequest requestWithURL:url];
    NSURLResponse *response;
    NSError *error;
    success = NO;

    NSData* result = [NSURLConnection
        sendSynchronousRequest:theRequest
        returningResponse:&response error:&error];

    if (!result)
    {
        [self log:@"Download error: %@",
            [error localizedFailureReason]];
        return;
    }

    if (response.expectedContentLength ==
        NSURLResponseUnknownLength)
    {
        [self log:@"Download error."];
        return;
    }

    if (![response.suggestedFilename
        isEqualToString:url.path.lastPathComponent])
    {
        [self log:@"Name mismatch. Probably carrier error page"];
        return;
    }

    if (response.expectedContentLength != result.length)
    {
        [self log:@"Got %d bytes, expected %d",
            result.length, response.expectedContentLength];
        return;
    }

    success = YES;
    [self log:@"Read %d bytes", result.length];
    [result writeToFile:DEST_PATH atomically:YES];
    [self log:@"Data written to file: %@.", DEST_PATH];
    [self log:@"Response suggested file name: %@",
        response.suggestedFilename];
    [self log:@"Elapsed time: %0.2f seconds.",
        [[NSDate date] timeIntervalSinceDate:startDate]];
}
```

From the Library of Eric Berna

```
- (void) go
{
    // Disable GUI
    self.navigationItem.rightBarButtonItem = nil;
    seg.enabled = NO;

    // Choose which item to download
    NSArray *items = [NSArray arrayWithObjects:
        SMALL_URL, LARGE_URL, FAKE_URL, nil];
    NSString *whichItem =
        [items objectAtIndex:seg.selectedSegmentIndex];
    NSURL *sourceURL = [NSURL URLWithString:whichItem];

    // Remove any existing data
    if ([[NSFileManager defaultManager] fileExistsAtPath:DEST_PATH])
        [[NSFileManager defaultManager]
            removeItemAtPath:DEST_PATH error:nil];

    // Start the network activity indicator
    [[UIApplication sharedApplication]
        setNetworkActivityIndicatorVisible:YES];

    // Perform the download on a new queue
    [[[NSOperationQueue alloc] init] addOperationWithBlock:
     ^{
         [self getData:sourceURL];
         [[NSOperationQueue mainQueue] addOperationWithBlock:^{
             // Finish up on main thread
             [self downloadFinished];
         }];
     }];
}
```

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 15 and open the project for this recipe.

Asynchronous Downloads in Theory

iOS 5 will introduce a beautiful and elegant way to download data. Unfortunately, at the time this book was being written—and about to go to press—it was completely broken in the beta SDK.

The following code snippet demonstrates how the download *will* work on release. It was tested thoroughly on various iOS 5 betas. Its only problem was that the destination

From the Library of Eric Berna

URL did not point to the actual downloaded data. Otherwise, everything worked properly both on the simulator and the device. The progress bar updated properly, and the finished downloading callback was called at the end of the download. Your client must declare the `NSURLConnectionDownloadDelegate` protocol.

Use appropriate caution in adapting these calls to your code in case Apple changes how it deploys the asynchronous elements in its release SDK.

```
- (void)connection:(NSURLConnection *)connection
    didWriteData:(long long)bytesWritten
    totalBytesWritten:(long long)totalBytesWritten
    expectedTotalBytes:(long long)expectedTotalBytes
{
    float percent = (float) totalBytesWritten /
        (float) expectedTotalBytes;

    // Perform GUI update on main thread
    [[NSOperationQueue mainQueue] addOperationWithBlock:^(){
        progress.progress = percent;
    }];
}

- (void)connectionDidFinishDownloading:(NSURLConnection *)connection
    destinationURL:(NSURL *)destinationURL
{
    if (!destinationURL)
    {
        self.title = @"Download Failed";
        return;
    }

    // This is broken in Summer 2011.
    NSLog(@"Theoretically downloaded to: %@", destinationURL);
}

- (void) go
{
    self.navigationItem.rightBarButtonItem = nil;

    progress.progress = 0.0f;

    NSURL *url = [NSURL URLWithString:DATA_URL_STRING];
    NSURLRequest *request = [NSURLRequest requestWithURL:url];
    connection = [NSURLConnection
        connectionWithRequest:request delegate:self];
}
```

From the Library of Eric Berna

Recipe: Asynchronous Downloads

This recipe introduces asynchronous downloads outside of the `connectionWithRequest:delegate:` approach. Its helper class has been tested with iOS 5 and continues to work properly. Be aware that several of the methods you'll read about in this recipe are officially deprecated in favor of the newer and cleaner asynchronous downloads shown in the previous section. Specifically, these deprecations affect `connection:didReceiveData:`, `connection:didFailWithError:`, and `connection:didReceiveResponse:`. These calls are available in iOS 5 but may be withdrawn in some future iOS update.

Asynchronous downloads allow your application to download data in the background without explicit operation queues. They keep your code from blocking while waiting for a download to finish. You might use asynchronous downloads with table views, presenting placeholder images while downloading thumbnails from a service such as YouTube. Recipe 15-3 supports using `NSURLConnections` asynchronously. It builds a helper class called `DownloadHelper` that hides the details involved in downloading data. It works in the following fashion. Instead of sending a synchronous request, it initializes the connection and assigns a delegate.

```
NSURLConnection *theConnection = [[NSURLConnection alloc]
    initWithRequest:theRequest delegate:sharedInstance];
```

When a connection is set up this way, the data starts to download asynchronously, but it does not yet allow the GUI to update without blocking. To accomplish that, you must schedule the connection on the current run loop. Make sure to unschedule the connection after the download finishes or errors out. A download may finish either by retrieving all the requested data or failing with an error.

```
[self.urlconnection scheduleInRunLoop:[NSRunLoop currentRunLoop]
    forMode:NSRunLoopCommonModes];
```

Delegate methods help you track the download life cycle. You receive updates when new data is available, when the data has finished downloading, or if the download fails. To support these callbacks, the `DownloadHelper` class defines its variables and properties as follows:

- Two `strong` string properties point to the requested resource (`urlString`) and where its data is to be saved (`targetPath`). These `urlStrings` are used to initialize the URL request that begins the download process (`requestWithURL:`).
- `outputStream` is a strong instance variable that allows the incoming data to be written to the `targetPath` as it arrives.
- A weak `delegate` property points to the client object. The delegate, which implements the custom `DownloadHelperDelegate` protocol, is updated with optional callbacks as the download progresses. This external delegate is distinct from the internal delegate used with the `NSURLConnection` object. External callbacks occur when the download succeeds (`connection:didFinishLoading:`), fails

From the Library of Eric Berna

(connection:didFailWithError:), when the filename becomes known (connection:didReceiveResponse:), and as each chunk of data arrives (connection:didReceiveData:).

- The data consumer can update a progress view to show the user how far a download has progressed by querying the read-only bytesRead and expectedLength properties. The Boolean isDownloading property indicates whether the helper object is currently downloading.
- The urlconnection property stores the current NSURLConnection object. It is kept on hand to allow the DownloadHelper class's cancel method to halt an ongoing download.

```
@protocol DownloadHelperDelegate <NSObject>
@optional
- (void) downloadFinished;
- (void) downloadReceivedData;
- (void) dataDownloadFailed: (NSString *) reason;
@end

@interface DownloadHelper : NSObject
{
    NSOutputStream *outputStream;
    NSURLConnection *urlconnection;

    BOOL isDownloading;
    int bytesRead;
    int expectedLength;
}
@property (strong) NSString *urlString;
@property (strong) NSString *targetPath;
@property (weak) id <DownloadHelperDelegate> delegate;

@property (readonly) BOOL isDownloading;
@property (readonly) int bytesRead;
@property (readonly) int expectedLength;

+ (id) download:(NSString *) aURLString
    withTargetPath: (NSString *) aPath
    withDelegate: (id <DownloadHelperDelegate>) aDelegate;
- (void) cancel;
@end
```

The client starts the download by calling the class download:withTargetPath:withDelegate: convenience method. It presumably assigns itself as the DownloadHelper delegate. This starts a new download and returns the new

From the Library of Eric Berna

`DownloadHelper` object, which should be retained by the client until the download finishes or the client specifically asks it to `cancel`.

> **Note**
>
> Recipe 15-3 assumes that you are assured an expected content length from the data provider. When the server side returns a response using chunked data (that is, `Transfer-Encoding:chunked`), the content length is not specified in the response. Recipe 15-3 does not work with chunked data because it tests for content length and fails if the expected length is unknown (`NSURLResponseUnknownLength`).

Recipe 15-3 **Download Helper**

```
#define SAFE_PERFORM_WITH_ARG(THE_OBJECT, THE_SELECTOR, THE_ARG) \
    (([THE_OBJECT respondsToSelector:THE_SELECTOR]) ? \
    [THE_OBJECT performSelector:THE_SELECTOR withObject:THE_ARG] : nil)

@implementation DownloadHelper
@synthesize delegate, urlString, targetPath;
@synthesize isDownloading, bytesRead, expectedLength;

// Begin the download
- (void) start
{
    isDownloading = NO;
    if (!urlString)
    {
        NSLog(@"URL string required but not set");
        return;
    }

    NSURL *url = [NSURL URLWithString:urlString];
    if (!url)
    {
        NSString *reason = [NSString stringWithFormat:
            @"Could not create URL from string %@", urlString];
        SAFE_PERFORM_WITH_ARG(delegate,
            @selector(dataDownloadFailed:), reason);
        return;
    }

    NSMutableURLRequest *theRequest =
        [NSMutableURLRequest requestWithURL:url];
    if (!theRequest)
    {
        NSString *reason = [NSString stringWithFormat:
            @"Could not create URL request from string %@",
            urlString];
```

From the Library of Eric Berna

```
            SAFE_PERFORM_WITH_ARG(delegate,
                @selector(dataDownloadFailed:), reason);
            return;
        }

        urlconnection = [[NSURLConnection alloc]
            initWithRequest:theRequest delegate:self];
        if (!urlconnection)
        {
            NSString *reason = [NSString stringWithFormat:
                @"URL connection failed for string %@", urlString];
            SAFE_PERFORM_WITH_ARG(delegate,
                @selector(dataDownloadFailed:), reason);
            return;
        }

        outputStream = [[NSOutputStream alloc]
            initToFileAtPath:targetPath append:YES];
        if (!outputStream)
        {
            NSString *reason = [NSString stringWithFormat:
                @"Could not create output stream at path %@",
                targetPath];
            SAFE_PERFORM_WITH_ARG(delegate,
                @selector(dataDownloadFailed:), reason);
            return;
        }
        [outputStream open];

        isDownloading = YES;
        bytesRead = 0;

        NSLog(@"Beginning download");
        [urlconnection scheduleInRunLoop:[NSRunLoop currentRunLoop]
            forMode:NSRunLoopCommonModes];
    }

    - (void) cleanup
    {
        isDownloading = NO;
        if (urlconnection)
        {
            [urlconnection cancel];
            urlconnection = nil;
        }
```

From the Library of Eric Berna

```
    if (outputStream)
    {
        [outputStream close];
        outputStream = nil;
    }

    self.urlString = nil;
    self.targetPath = nil;
}

- (void) dealloc
{
    [self cleanup];
}

- (void) cancel
{
    [self cleanup];
}

- (void)connection:(NSURLConnection *)connection
    didReceiveResponse:(NSURLResponse *)aResponse
{
    // Check for bad connection
    expectedLength = [aResponse expectedContentLength];
    if (expectedLength == NSURLResponseUnknownLength)
    {
        NSString *reason = [NSString stringWithFormat:
            @"Invalid URL [%@]", urlString];
        SAFE_PERFORM_WITH_ARG(delegate,
            @selector(dataDownloadFailed:), reason);
        [connection cancel];
        [self cleanup];
        return;
    }
}

- (void)connection:(NSURLConnection *)connection
    didReceiveData:(NSData *)theData
{
    bytesRead += theData.length;
    NSUInteger bytesLeft = theData.length;
    NSUInteger bytesWritten = 0;
    do {
        bytesWritten = [outputStream write:theData.bytes
            maxLength:bytesLeft];
```

From the Library of Eric Berna

```
        if (-1 == bytesWritten) break;
        bytesLeft -= bytesWritten;
    } while (bytesLeft > 0);
    if (bytesLeft) {
        NSLog(@"stream error: %@", [outputStream streamError]);
    }

    SAFE_PERFORM_WITH_ARG(delegate,
        @selector(downloadReceivedData), nil);
}

- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
    // finished downloading the data, cleaning up
    [outputStream close];
    [urlconnection
        unscheduleFromRunLoop:[NSRunLoop currentRunLoop]
        forMode:NSRunLoopCommonModes];
    [self cleanup];

    SAFE_PERFORM_WITH_ARG(delegate,
        @selector(downloadFinished), nil);
}

- (void)connection:(NSURLConnection *)connection
    didFailWithError:(NSError *)error
{
    isDownloading = NO;
    NSLog(@"Error: Failed connection, %@",
        [error localizedFailureReason]);
    SAFE_PERFORM_WITH_ARG(delegate,
        @selector(dataDownloadFailed:), @"Failed Connection");
    [self cleanup];
}

+ (id) download:(NSString *) aURLString
    withTargetPath: (NSString *) aPath
    withDelegate: (id <DownloadHelperDelegate>) aDelegate
{
    if (!aURLString)
    {
        NSLog(@"Error: No URL string");
        return nil;
    }

    if (!aPath)
    {
        NSLog(@"Error: No target path");
```

From the Library of Eric Berna

```
        return nil;
    }

    DownloadHelper *helper = [[self alloc] init];
    helper.urlString = aURLString;
    helper.targetPath = aPath;
    helper.delegate = aDelegate;
    [helper start];

    return helper;
}
@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 15 and open the project for this recipe.

Handling Authentication Challenges

Some websites are protected with usernames and passwords. `NSURLConnection` lets you access these sites by responding to authentication challenges. Prior to iOS 5, you may have handled credentials using delegate callbacks like this:

```
- (void)connection:(NSURLConnection *)connection
    didReceiveAuthenticationChallenge:
    (NSURLAuthenticationChallenge *)challenge
{
    NSURLProtectionSpace *protectionSpace =
        [[NSURLProtectionSpace alloc]
            initWithHost:primaryHost port:0
            protocol:@"http" realm:nil
            authenticationMethod:nil];
    NSURLCredential *credential =
        [[NSURLCredentialStorage sharedCredentialStorage]
            defaultCredentialForProtectionSpace:protectionSpace];

    [[challenge sender] useCredential:credential
        forAuthenticationChallenge:challenge];
}

- (void)connection:(NSURLConnection *)connection
    didCancelAuthenticationChallenge:
    (NSURLAuthenticationChallenge *)challenge
{
    NSLog(@"Challenge cancelled");
}
```

From the Library of Eric Berna

That approach has changed under iOS 5, as these callbacks are now deprecated. Instead, you use a new connection delegate callback in anticipation of that challenge.

```
- (void)connection:(NSURLConnection *)connection
    willSendRequestForAuthenticationChallenge:
        (NSURLAuthenticationChallenge *)challenge
{
    NSURLProtectionSpace *protectionSpace =
        [[NSURLProtectionSpace alloc]
           initWithHost:primaryHost port:0
           protocol:@"http" realm:nil
           authenticationMethod:nil];
    NSURLCredential *credential =
        [[NSURLCredentialStorage sharedCredentialStorage]
            defaultCredentialForProtectionSpace:protectionSpace];

    [[challenge sender] useCredential:credential
        forAuthenticationChallenge:challenge];
}
```

Storing Credentials

The examples you just saw use the `NSURLCredential` class along with `NSURLProtectionSpace` to store their credentials. You can set these credentials along the lines of the following snippet:

```
NSURLCredential = [NSURLCredential
    credentialWithUser:username password:password
    persistence: NSURLCredentialPersistenceForSession];
NSURLProtectionSpace *protectionSpace =
    [[NSURLProtectionSpace alloc] initWithHost:@"ericasadun.com"
    port:0 protocol:@"http" realm:nil authenticationMethod:nil];
[[NSURLCredentialStorage sharedCredentialStorage]
    setDefaultCredential:credential
    forProtectionSpace:protectionSpace];
```

This series of calls creates a new credential, defines a protection space, and saves the credential to the shared credential storage. As the persistence is set to session, the credentials will not last past the immediate application execution. You can use three kinds of persistence with credential storage:

- `NSURLCredentialPersistenceNone` means the credential must be used immediately and cannot be stored for future use.
- `NSURLCredentialPersistenceForSession` allows the credential to be stored for just this application session.
- `NSURLCredentialPersistencePermanent` permits the credential to be stored to the user's keychain and be shared with other applications as well.

From the Library of Eric Berna

The protection space defines a scope for which the credential is valid. Here, it can be used for the ericasadun.com website and is only valid within that scope.

Recipe 15-4 demonstrates some of this process in action, in an extremely unrealistic way. That is, it creates nonpersistent credentials forcing constant challenges. This is tutorial in nature only. Do not use this approach in real deployment. This approach allows you to switch between authentic and inauthentic credentials, to demonstrate how the site to challenge callback is handled.

In real-world use, you will more likely use the keychain or set credential persistence to last across an entire session. Use the default credential approach shown earlier in this section to recover stored credentials for those challenges or, better yet, use the new `NSURLConnection` delegate callback method, `connectionShouldUseCredentialStorage:`, and have it always return `YES`.

To test authentication, Recipe 15-4 connects to http://ericasadun.com/Private, which was set up for use with this authentication discussion. This test folder uses the username `PrivateAccess` and password `tuR7!mZ#eh`.

To test an unauthorized connection—that is, you will be refused—set the password to `nil` or to a nonsense string. When the password is set to `nil`, the challenge will be sent a `nil` credential, producing an immediate failure. With nonsense strings, the challenge will fail after the sender rejects the credentials.

Recipe 15-4 **Authentication with `NSURLCredential` Instances**

```
@implementation TestBedViewController
- (void)connection:(NSURLConnection *)connection
    didReceiveData:(NSData *)data
{
    // Display the HTML returned
    NSString *htmlString = [[NSString alloc]
        initWithData:data encoding:NSUTF8StringEncoding];
    [webView loadHTMLString:htmlString
        baseURL:[NSURL URLWithString://ericasadun.com"]];
}

- (void)connection:(NSURLConnection *)connection
    didFailWithError:(NSError *)error
{
    NSLog(@"Failed. Error: %@", [error localizedFailureReason]);
    [webView loadHTMLString:@"<h1>Failed</h1>" baseURL:nil];
}

// This is called at most once, before a resource loads
- (BOOL)connectionShouldUseCredentialStorage:(NSURLConnection *)connection
{
    NSLog(@"Being queried about credential storage. Saying no.");
    return NO;
}
```

From the Library of Eric Berna

```
- (void)connection:(NSURLConnection *)connection
    willSendRequestForAuthenticationChallenge:
        (NSURLAuthenticationChallenge *)challenge
{
    // This allows toggling between success and fail states
    if (hasBeenTested)
    {
        NSLog(@"Second time around for authentication challenge. Fail.");
        [webView loadHTMLString:@"<h1>Fail</h1>" baseURL:nil];
        return;
    }
    else
        hasBeenTested = shouldFail;

    // Here the credentials are both hardwired and not stored
    // Without persistence, the credential is free to change
    NSString *username = @"PrivateAccess";
    NSString *password = @"tuR7!mZ#eh";

    NSURLCredential *credential =
       [NSURLCredential credentialWithUser:username
           password:shouldFail ? nil : password
           persistence:NSURLCredentialPersistenceNone];
    [[challenge sender] useCredential:credential
        forAuthenticationChallenge:challenge];
}

- (void) go
{
    NSURL *url =
        [NSURL URLWithString://ericasadun.com/Private"];

    NSURLRequest *request = [NSURLRequest requestWithURL:url
        cachePolicy:NSURLRequestReloadIgnoringLocalAndRemoteCacheData
        timeoutInterval:10];

    hasBeenTested = NO;
    NSURLConnection *connection =
        [NSURLConnection connectionWithRequest:request delegate:self];
    [connection start];
}

- (void) toggle
{
    shouldFail = !shouldFail;
    self.title = shouldFail ? @"Should fail" : @"Should succeed";
}
```

From the Library of Eric Berna

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 15 and open the project for this recipe.

Recipe: Storing and Retrieving Keychain Credentials

Recipe 15-4 demonstrated authentication challenges, and how not to store and retrieve keychain credentials. Recipe 15-5 shows how you should manage those credentials, storing them persistently in the iOS keychain.

This is quite a complicated little recipe because there's a lot going on here. In addition to storing general user credentials, the keychain can mark a default pair. This pair is retrieved with `defaultCredentialForProtectionSpace:`. That's what you see happening in the `viewWillAppear:` method.

As the view appears, the recipe loads any default credentials if available. It then sets those in the username and password fields. Upon being dismissed with Done, the recipe stores any updates. If the user taps Cancel rather than Done, the dialog is dismissed without saving. When the `storeCredentials` method sets the default credential, the username/password pair is saved (or resaved) to the keychain *and* marked as the default.

The recipe implements several text field delegate methods as well. If the user taps Done on the keyboard, it's treated as if he or she had tapped Done in the navigation bar. The text field is resigned, the data stored, and the `PasswordController` dismissed.

Another text field delegate method looks for the start of editing. Whenever the user name field is edited, the password is invalidated and cleared. At the same time the should-change-characters method is busy examining any text in the username field. Should it find a credential that matches the new username, it automatically updates the password field. This allows users to switch between several usernames for a given service without having to retype in the password each time they do so.

Recipe 15-5 **Persistent Credentials**

```
@implementation PasswordController
- (void) viewWillAppear:(BOOL)animated
{
    // Initially disable Cancel until there's something to cancel
    self.navigationItem.leftBarButtonItem.enabled = NO;

    // Load up the default credential
    NSURLProtectionSpace *protectionSpace =
        [[NSURLProtectionSpace alloc] initWithHost:HOST port:0
            protocol:@"http" realm:nil authenticationMethod:nil];
    NSURLCredential *credential =
```

From the Library of Eric Berna

```
        [[NSURLCredentialStorage sharedCredentialStorage]
            defaultCredentialForProtectionSpace:protectionSpace];

    // Set the text fields if credentials are found
    if (credential)
    {
        username.text = credential.user;
        password.text = credential.password;
    }

    // Never log credentials in real world deployment!
    // This is for testing feedback only!
    NSLog(@"Loading [%@, %@]", username.text, password.text);
}

- (void) storeCredentials
{
    NSURLCredential *credential = [NSURLCredential
        credentialWithUser:username.text
        password:password.text
        persistence: NSURLCredentialPersistencePermanent];
    NSURLProtectionSpace *protectionSpace =
        [[NSURLProtectionSpace alloc] initWithHost:HOST port:0
            protocol:@"http" realm:nil authenticationMethod:nil];

    // Most recent is always default credential
    [[NSURLCredentialStorage sharedCredentialStorage]
        setDefaultCredential:credential
        forProtectionSpace:protectionSpace];
}

// Dismiss and store credentials
- (IBAction) done:(id)sender
{
    [self dismissModalViewControllerAnimated:YES];
    [self storeCredentials];

    // Never log credentials in real world deployment!
    // This is for testing feedback only!
    NSLog(@"Storing [%@, %@]", username.text, password.text);
}

// Cancel dismisses without storing current credentials
- (IBAction) cancel:(id)sender
{
    [self dismissModalViewControllerAnimated:YES];
}
```

From the Library of Eric Berna

```
// User tapping "done" means done
- (BOOL)textFieldShouldReturn:(UITextField *)textField
{
    [textField resignFirstResponder];
    [self done:nil];
    return YES;
}

// Respond to field edits as they start
- (void)textFieldDidBeginEditing:(UITextField *)textField
{
    // Empty password when the user name changes
    if (textField == username)
        password.text = @"";

    // Only enable cancel on edits
    self.navigationItem.leftBarButtonItem.enabled = YES;
}

// Watch for known usernames during text edits
- (BOOL)textField:(UITextField *)textField
    shouldChangeCharactersInRange:(NSRange)range
    replacementString:(NSString *)string
{
    if (textField != username) return YES;

    // Calculate the target string that will occupy the field
    NSString *targetString = [textField.text
        stringByReplacingCharactersInRange:range
        withString:string];
    if (!targetString) return YES;
    if (!targetString.length) return YES;

    // Always check if there's a matching password on file
    NSURLProtectionSpace *protectionSpace =
        [[NSURLProtectionSpace alloc] initWithHost:HOST port:0
            protocol:@"http" realm:nil authenticationMethod:nil];
    NSDictionary *credentialDictionary =
        [[NSURLCredentialStorage sharedCredentialStorage]
             credentialsForProtectionSpace:protectionSpace];
    NSURLCredential *pwCredential = [
        credentialDictionary objectForKey:targetString];
    if (!pwCredential) return YES;

    // Match! If so, update the password field
    password.text = pwCredential.password;
    return YES;
```

From the Library of Eric Berna

```
}
// Modal view controllers should rotate with their parents
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation
{
    return YES;
}
@end
```

> **Get This Recipe's Code**
>
> To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from the book, go to the folder for Chapter 15 and open the project for this recipe.

Recipe: Uploading Data

Recipe 15-6 uses `POST` to create a full multipart form data submission. This recipe allows you to upload images to the TwitPic.com service using your user's Twitter credentials. The twitpic API is accessed at http://twitpic.com/api/uploadAndPost. It requires a username, password, and binary image data.

The challenge for Recipe 15-5 is to create a properly formatted body that can be used by the TwitPic service. It implements a method that generates form data from a dictionary of keys and values. For the purposes of this example, the objects in that dictionary are limited to strings and images. You can extend this approach for other data types by changing the content type string with different MIME types.

`NSOperationQueue`

This recipe uses a synchronous request to perform the upload, which can take up to a minute or so to process. To avoid blocking GUI updates, the entire submission process is embedded into an `NSOperation` subclass, `TwitterOperation`. Operations encapsulate code and data for a single task, allowing you to run that task asynchronously.

Using `NSOperation` objects lets you submit them to an asynchronous `NSOperationQueue`, a class you saw used earlier in this chapter. Operation queues manage the execution of individual operations. Each operation is prioritized and placed into the queue, where it is executed in that priority order.

```
TwitPicOperation *op = [TwitPicOperation
    operationWithDelegate:self andPath:path];
[op start];
```

Whenever you subclass `NSOperation`, make sure to implement a `main` method. This method is called when the operation executes. When `main` returns, the operation finishes.

From the Library of Eric Berna

Recipe 15-6 **Uploading Images to TwitPic**

```
#define NOTIFY_AND_LEAVE(MESSAGE) \
    {[self cleanup:MESSAGE]; return;}
#define DATA(STRING) \
    [STRING dataUsingEncoding:NSUTF8StringEncoding]
#define SAFE_PERFORM_WITH_ARG(THE_OBJECT, THE_SELECTOR, THE_ARG)\
    (([THE_OBJECT respondsToSelector:THE_SELECTOR]) ? \
    [THE_OBJECT performSelector:THE_SELECTOR \
    withObject:THE_ARG] : nil)
#define HOST    @"twitpic.com"

// Posting constants
#define IMAGE_CONTENT @"Content-Disposition: form-data; \
    name=\"%@\"; filename=\"image.jpg\"\r\n\
    Content-Type: image/jpeg\r\n\r\n"
#define STRING_CONTENT @"Content-Disposition: form-data;\
    name=\"%@\"\r\n\r\n"
#define MULTIPART @"multipart/form-data; \
    boundary=------------0x0x0x0x0x0x0x0x"

@implementation TwitPicOperation
@synthesize imageData, delegate;

- (void) cleanup: (NSString *) output
{
    self.imageData = nil;
    SAFE_PERFORM_WITH_ARG(delegate,
        @selector(doneTweeting:), output);
}

- (NSData*)generateFormDataFromPostDictionary:
    (NSDictionary*)dict
{
    id boundary = @"------------0x0x0x0x0x0x0x0x";
    NSArray* keys = [dict allKeys];
    NSMutableData* result = [NSMutableData data];

    for (int i = 0; i < [keys count]; i++)
    {
        id value = [dict valueForKey:[keys objectAtIndex:i]];
        [result appendData:[[NSString stringWithFormat:
            @"--%@\r\n", boundary]
            dataUsingEncoding:NSUTF8StringEncoding]];

        if ([value isKindOfClass:[NSData class]])
        {
```

From the Library of Eric Berna

```
            // Handle image data
            NSString *formstring =
                [NSString stringWithFormat:IMAGE_CONTENT,
                [keys objectAtIndex:i]];
            [result appendData:DATA(formstring)];
            [result appendData:value];
        }
        else
        {
            // All non-image fields assumed to be strings
            NSString *formstring =
                [NSString stringWithFormat:STRING_CONTENT,
                    [keys objectAtIndex:i]];
            [result appendData:DATA(formstring)];
            [result appendData:DATA(value)];
        }

        NSString *formstring = @"\r\n";
        [result appendData:DATA(formstring)];
    }

    NSString *formstring =[NSString stringWithFormat:
        @"--%@--\r\n", boundary];
    [result appendData:DATA(formstring)];
    return result;
}

- (void) main
{
    if (!self.imageData)
        NOTIFY_AND_LEAVE(@"ERROR: Set image before uploading.");

    NSURLProtectionSpace *protectionSpace =
        [[NSURLProtectionSpace alloc] initWithHost:HOST
            port:0 protocol:@"http" realm:nil
            authenticationMethod:nil];

    NSURLCredential *credential =
        [[NSURLCredentialStorage sharedCredentialStorage]
            defaultCredentialForProtectionSpace:protectionSpace];
    if (!credential)
        NOTIFY_AND_LEAVE(@"ERROR: Credentials not set.")

    NSString *uname = credential.user;
    NSString *pword = credential.password;
```

From the Library of Eric Berna

```
if (!uname || !pword || (!uname.length) || (!pword.length))
    NOTIFY_AND_LEAVE(@"ERROR: Credentials not set.")

NSMutableDictionary* post_dict =
    [[NSMutableDictionary alloc] init];
[post_dict setObject:uname forKey:@"username"];
[post_dict setObject:pword forKey:@"password"];
[post_dict setObject:@"Posted from iTweet" forKey:@"message"];
[post_dict setObject:self.imageData forKey:@"media"];

// Create the post data from the post dictionary
NSData *postData =
    [self generateFormDataFromPostDictionary:post_dict];

// Establish the API request.
// Use upload vs uploadAndPost to skip tweet
NSString *baseurl = @"http://twitpic.com/api/upload";
NSURL *url = [NSURL URLWithString:baseurl];
NSMutableURLRequest *urlRequest =
    [NSMutableURLRequest requestWithURL:url];
if (!urlRequest)
    NOTIFY_AND_LEAVE(@"ERROR: Error creating the URL Request");

[urlRequest setHTTPMethod: @"POST"];
[urlRequest setValue:MULTIPART
    forHTTPHeaderField: @"Content-Type"];
[urlRequest setHTTPBody:postData];

// Submit & retrieve results
NSError *error;
NSURLResponse *response;
NSLog(@"Contacting TwitPic....");
NSData* result = [NSURLConnection
    sendSynchronousRequest:urlRequest
    returningResponse:&response error:&error];

if (!result)
{
    [self cleanup:[NSString stringWithFormat:
        @"Submission error: %@",
            [error localizedFailureReason]]];
    return;
}

// Return results
NSString *outstring = [[NSString alloc]
```

From the Library of Eric Berna

```
        initWithData:result encoding:NSUTF8StringEncoding];
    [self cleanup: outstring];
}

+ (id) operationWithDelegate: (id) delegate andPath: (NSString *) path
{
    NSData *data = [NSData dataWithContentsOfFile:path];
    if (!data) return nil;

    TwitPicOperation *op = [[TwitPicOperation alloc] init];
    op.delegate = delegate;
    op.imageData = data;

    return op;
}
@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 15 and open the project for this recipe.

Twitter

Apple's new Twitter framework offers a prebuilt tweet composition view controller as well as a simple request API that you can easily integrate into your applications. It makes tweeting a simple option that you can add to your code with very little programming.

Here's basically everything you need to do in order to tweet. This method establishes a new tweet view controller, adds an image to it, sets its completion handler to dismiss, and presents it:

```
- (void) tweet: (id) sender
{
    TWTweetComposeViewController *tweeter =
       [[TWTweetComposeViewController alloc] init];
    [tweeter addImage:imageView.image];

    TWTweetComposeViewController __weak *twee = tweeter;
    tweeter.completionHandler =
    ^(TWTweetComposeViewControllerResult result){
        [[NSOperationQueue mainQueue] addOperationWithBlock:^(){
            TWTweetComposeViewController __strong *twong = twee;
            [twong dismissModalViewControllerAnimated:YES];
        }];
    };
```

From the Library of Eric Berna

```
    if (IS_IPAD)
        tweeter.modalPresentationStyle = UIModalPresentationFormSheet;
    [self presentModalViewController:tweeter animated:YES];
}
```

The tweet is sent using the default Twitter credentials established in Settings > Twitter. A simple check allows you to test whether these settings have been initialized and are usable.

```
if (TWTweetComposeViewController.canSendTweet)
        self.navigationItem.rightBarButtonItem =
           BARBUTTON(@"Tweet", @selector(tweet:));
```

If desired, you can use a custom interface without the pre-cooked controller by directly creating a `TWRequest`, which encapsulates a HTTP request with direct control of the operation method (`GET`, `POST`, `DELETE`) and its parameters.

Recipe: Converting XML into Trees

The `NSXMLParser` class provided in the iPhone SDK scans through XML, creating callbacks as new elements are processed and finished (that is, using the typical logic of a SAX parser). The class is terrific for when you're downloading simple data feeds and want to scrape just a bit or two of relevant information. It may not be so great when you're doing production-type work that relies on error checking, status information, and back-and-forth handshaking.

Trees

Tree data structures offer an excellent way to represent XML data. They allow you to create search paths through the data, so you can find just the data you're looking for. You can retrieve all "entries," search for a success value, and so forth. Trees convert text-based XML back into a multidimensional structure.

To bridge the gap between `NSXMLParser` and tree-based parse results, you can use an `NSXMLParser`-based helper class to return more standard tree-based data. This requires a simple tree node like the kind shown here. This node uses double linking to access its parent and its children, allowing two-way traversal in a tree. Only parent-to-child values are retained, allowing the tree to deallocate without being explicitly torn down.

```
@interface TreeNode : NSObject
@property (nonatomic, assign)     TreeNode        *parent;
@property (nonatomic, strong)     NSMutableArray  *children;
@property (nonatomic, strong)     NSString        *key;
@property (nonatomic, strong)     NSString        *leafvalue;
@end
```

From the Library of Eric Berna

Building a Parse Tree

Recipe 15-7 introduces the `XMLParser` class. Its job is to build a parse tree as the `NSXMLParser` class works its way through the XML source. The three standard `NSXML` routines (start element, finish element, and found characters) read the XML stream and perform a recursive depth-first descent through the tree.

The class adds new nodes when reaching new elements (`parser:didStartElement:qualifiedName:attributes:`) and adds leaf values when encountering text (`parser:foundCharacters:`). Because XML allows siblings at the same tree depth, this code uses a stack to keep track of the current path to the tree root. Siblings always pop back to the same parent in `parser:didEndElement:`, so they are added at the proper level.

After finishing the XML scan, the `parseXMLFile:` method returns the root node.

Recipe 15-7 **The `XMLParser` Helper Class**

```
@implementation XMLParser
// Parser returns the tree root. You have to go down
// one node to the real results
- (TreeNode *) parse: (NSXMLParser *) parser
{
    stack = [NSMutableArray array];
    TreeNode *root = [TreeNode treeNode];
    [stack addObject:root];

    [parser setDelegate:self];
    [parser parse];

    // Pop down to real root
    TreeNode *realroot = [[root children] lastObject];

    // Remove any connections
    root.children = nil;
    root.leafvalue = nil;
    root.key = nil;
    realroot.parent = nil;

    // Return the true root
    return realroot;
}

- (TreeNode *)parseXMLFromURL: (NSURL *) url
{
    TreeNode *results = nil;
    @autoreleasepool {
        NSXMLParser *parser =
            [[NSXMLParser alloc] initWithContentsOfURL:url];
        results = [self parse:parser];
```

From the Library of Eric Berna

```
    }
    return results;
}

- (TreeNode *)parseXMLFromData: (NSData *) data
{
    TreeNode *results = nil;
    @autoreleasepool {
        NSXMLParser *parser =
            [[NSXMLParser alloc] initWithData:data];
        results = [self parse:parser];
    }
    return results;
}

// Descend to a new element
- (void)parser:(NSXMLParser *)parser
    didStartElement:(NSString *)elementName
    namespaceURI:(NSString *)namespaceURI
    qualifiedName:(NSString *)qName
    attributes:(NSDictionary *)attributeDict
{
    if (qName) elementName = qName;

    TreeNode *leaf = [TreeNode treeNode];
    leaf.parent = [stack lastObject];
    [(NSMutableArray *)[[stack lastObject] children] addObject:leaf];

    leaf.key = [NSString stringWithString:elementName];
    leaf.leafvalue = nil;
    leaf.children = [NSMutableArray array];

    [stack addObject:leaf];
}

// Pop after finishing element
- (void)parser:(NSXMLParser *)parser
    didEndElement:(NSString *)elementName
    namespaceURI:(NSString *)namespaceURI
    qualifiedName:(NSString *)qName
{
    [stack removeLastObject];
}

// Reached a leaf
- (void)parser:(NSXMLParser *)parser
    foundCharacters:(NSString *)string
```

From the Library of Eric Berna

```
{
    if (![[stack lastObject] leafvalue])
    {
        [[stack lastObject]
            setLeafvalue:[NSString stringWithString:string]];
        return;
    }
    [[stack lastObject] setLeafvalue:
        [NSString stringWithFormat:@"%@%
        [[stack lastObject] leafvalue], string]];
}
@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 15 and open the project for this recipe.

Using the Tree Results

Listing 15-6 demonstrates an XML parse-tree consumer. This example presents a series of table view controllers that drill down from the root of the tree until reaching the leaves. Whenever leaves are encountered, their values are displayed in an alert. Subtrees lead to additional view controller screens.

This example uses the `TreeNode` class trivially. The only items of interest are the leaf values and the child nodes. The class can do far more, including returning leaves and objects that match a given key. This functionality lets you retrieve information without knowing the exact path to a child node as long as you know what the node is called, such as "entry" or "published." These two names are in fact used by Twitter's API.

Listing 15-6 **Browsing the Parse Tree**

```
#pragma mark Table-based XML Browser
@interface XMLTreeViewController : UITableViewController
@property (strong) TreeNode *root;
@end

@implementation XMLTreeViewController
@synthesize root;

- (id) initWithRoot:(TreeNode *) newRoot
{
    if (self = [super init])
    {
        self.root = newRoot;
        if (newRoot.key) self.title = newRoot.key;
    }
```

From the Library of Eric Berna

```
    return self;
}

+ (id) controllerWithRoot: (TreeNode *) root
{
    XMLTreeViewController *tvc =
        [[XMLTreeViewController alloc] initWithRoot:root];
    return tvc;
}

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

- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section
{
    return [self.root.children count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:@"generic"];
    if (!cell) cell = [[UITableViewCell alloc]
        initWithStyle:UITableViewCellStyleDefault
        reuseIdentifier:@"generic"];
    TreeNode *child = [[self.root children]
        objectAtIndex:[indexPath row]];
    cell.textLabel.text = child.key;
    cell.textLabel.font = [UIFont fontWithName:@"Futura"
        size:IS_IPAD ? 36.0f : 18.0f];

    cell.accessoryType =
        UITableViewCellAccessoryDisclosureIndicator;
    if (child.isLeaf && IS_IPAD)
        cell.accessoryType = UITableViewCellAccessoryNone;

    return cell;
}

- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    TreeNode *child = [self.root.children objectAtIndex:[indexPath row]];
```

From the Library of Eric Berna

```
    if (child.isLeaf && IS_IPAD)
    {
        SplitDetailViewController *dvc =
           (SplitDetailViewController *)
               self.splitViewController.delegate;
        dvc.textView.text = child.leafvalue;
        return;
    }
    else if (child.isLeaf) // iPhone/iPod
    {
        TextViewController *dvc = [TextViewController
            controllerWithText:child.leafvalue];
        [self.navigationController
            pushViewController:dvc animated:YES];
        return;
    }
    XMLTreeViewController *tbc =
        [XMLTreeViewController controllerWithRoot:child];
    [self.navigationController
         pushViewController:tbc animated:YES];
}
@end
```

> **Note**
>
> The tree code in this edition of the Cookbook does not require you to tear down the tree after use. A few adjustments to the autorelease pool in this update allowed me to use a tree that's strongly linked from parent to child, and weakly linked the other way.

Recipe: Building a Simple Web-based Server

A web server provides one of the cleanest ways to serve data off your phone to another computer. You don't need special client software. Any browser can list and access web-based files. Best of all, a web server requires just a few key routines. You must establish the service, creating a loop that listens for a request (`startServer`), and then pass those requests onto a handler (`handleWebRequest:`) that responds with the requested data. Recipe 15-8 shows a `WebHelper` class that handles establishing and controlling a basic web service that serves the same image currently shown on the iOS device screen.

The loop routine uses low-level socket programming to establish a listening port and catch client requests. When the client issues a `GET` command, the server intercepts that request and passes it to the web request handler. The handler could decompose it, typically to find the name of the desired data file, but in this example, it serves back an image, regardless of the request specifics.

From the Library of Eric Berna

Recipe 15-8 **Serving iPhone Files Through a Web Service**

```
#define SAFE_PERFORM_WITH_ARG(THE_OBJECT, THE_SELECTOR, THE_ARG)\
    (([THE_OBJECT respondsToSelector:THE_SELECTOR]) ? \
    [THE_OBJECT performSelector:THE_SELECTOR withObject:THE_ARG]\
        : nil)

@implementation WebHelper
@synthesize isServing, chosenPort, delegate;

- (NSString *) getRequest: (int) fd
{
    static char buffer[BUFSIZE+1];
    int len = read(fd, buffer, BUFSIZE);
    buffer[len] = '\0';
    return [NSString stringWithCString:buffer
        encoding:NSUTF8StringEncoding];
}

- (void) handleWebRequest: (int) fd
{
    // Request an image from the delegate
    UIImage *image = SAFE_PERFORM_WITH_ARG(delegate,
        @selector(image), nil);
    if (!image) return;

    // Produce a jpeg header
    NSString *outcontent = [NSString stringWithFormat:
        @"HTTP/1.0 200 OK\r\nContent-Type: image/jpeg\r\n\r\n"];
    write (fd, [outcontent UTF8String], outcontent.length);

    // Send the data and close
    NSData *data = UIImageJPEGRepresentation(image, 0.75f);
    write (fd, data.bytes, data.length);
    close(fd);
}

// Listen for external requests
- (void) listenForRequests
{
    @autoreleasepool {
        static struct sockaddr_in cli_addr;
        socklen_t length = sizeof(cli_addr);

        while (1) {
            if (!isServing) return;
```

From the Library of Eric Berna

```
            if ((socketfd = accept(listenfd,
                (struct sockaddr *)&cli_addr, &length)) < 0)
            {
                isServing = NO;
                [[NSOperationQueue mainQueue]
                    addOperationWithBlock:^(){
                       SAFE_PERFORM_WITH_ARG(delegate,
                       @selector(serviceWasLost), nil);
                }];
                return;
            }
            [self handleWebRequest:socketfd];
        }
    }
}

// Begin serving data
- (void) startServer
{
    static struct sockaddr_in serv_addr;

    // Set up socket
    if((listenfd = socket(AF_INET, SOCK_STREAM,0)) < 0)
    {
        isServing = NO;
        SAFE_PERFORM_WITH_ARG(delegate,
            @selector(serviceCouldNotBeEstablished), nil);
        return;
    }

    // Serve to a random port
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = 0;

    // Bind
    if (bind(listenfd, (struct sockaddr *)&serv_addr,
        sizeof(serv_addr)) <0)
    {
        isServing = NO;
        SAFE_PERFORM_WITH_ARG(delegate,
            @selector(serviceCouldNotBeEstablished), nil);
        return;
    }

    // Find out what port number was chosen.
    int namelen = sizeof(serv_addr);
```

From the Library of Eric Berna

```
    if (getsockname(listenfd, (struct sockaddr *)&serv_addr,
        (void *) &namelen) < 0) {
        close(listenfd);
        isServing = NO;
        SAFE_PERFORM_WITH_ARG(delegate,
            @selector(serviceCouldNotBeEstablished), nil);
        return;
    }

    chosenPort = ntohs(serv_addr.sin_port);

    // Listen
    if(listen(listenfd, 64) < 0)
    {
        isServing = NO;
        SAFE_PERFORM_WITH_ARG(delegate,
            @selector(serviceCouldNotBeEstablished), nil);
        return;
    }

    isServing = YES;
    [NSThread
        detachNewThreadSelector:@selector(listenForRequests)
        toTarget:self withObject:NULL];
    SAFE_PERFORM_WITH_ARG(delegate,
        @selector(serviceWasEstablished:), self);
}

+ (id) serviceWithDelegate:(id)delegate
{
    if (![[UIDevice currentDevice] networkAvailable])
    {
        NSLog(@"Not connected to network");
        return nil;
    }

    WebHelper *helper = [[WebHelper alloc] init];
    helper.delegate = delegate ;
    [helper startServer];
    return helper;
}
@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 15 and open the project for this recipe.

From the Library of Eric Berna

One More Thing: Using JSON Serialization

iOS 5 introduced the `NSJSONSerialization` class, which is tremendously handy when you're working with web services. All you need is a valid JSON object (namely an array or dictionary) whose components are also valid objects, including strings, numbers, arrays, dictionaries, and `NSNull`. Test an object's validity with `isValidJSONObject`, which returns `YES` if the object can be safely converted to JSON format.

```
// Build a basic JSON object
NSArray *valueArray =
    [@"val1 val2 val3" componentsSeparatedByString:@" "];

NSMutableDictionary *dict = [NSMutableDictionary dictionary];
for (NSString *each in
    [@"KeyA KeyB KeyC" componentsSeparatedByString:@" "])
    dict setObject:valueArray forKey:each];

// Convert it to JSON
if ([NSJSONSerialization isValidJSONObject:dict])
{
    NSData *data = [NSJSONSerialization
        dataWithJSONObject:dict options:0 error:nil];
    NSString *result = [[NSString alloc]
        initWithData:data encoding:NSUTF8StringEncoding];
    [self log:@"Result: %@", result];
}
```

The code from this method produces the following JSON. Notice that dictionary output is not guaranteed to be in alphabetical order.

```
Result: {"KeyA":["val1","val2","val3"],"KeyC":
    ["val1","val2","val3"],"KeyB":["val1","val2","val3"]}
```

Going the other way is just as easy. Use `JSONObjectWithData:options:error:` to convert `NSData` representing a JSON object into an Objective-C representation.

Summary

This chapter introduced a wide range of network-supporting technologies. You saw how to check for network connectivity, work with the keychain for secure authentication challenges, upload and download data via `NSURLConnection`, and more. Here are a few thoughts to take away with you before leaving this chapter:

- Most of Apple's networking support is provided through very low-level C-based routines. If you can find a friendly Objective-C wrapper to simplify your programming work, consider using it. The only drawback occurs when you specifically need tight networking control at the most basic level of your application, which is rare. There are superb resources out there. Just Google for them.

From the Library of Eric Berna

- The new Twitter API support is very easy to integrate into your apps. Try to see where your app lends itself to this kind of support. It's not just about being social. Twitter provides a great mechanism for reminders, for information tracking, and more.
- There was not space in this chapter to discuss more detailed authentication schemes for data APIs. If you need access to OAuth or XAuth, for example, search for existing Cocoa implementations. A number are available in open-source repositories, and they are easily ported to Cocoa Touch. If you need to work with simpler data checksum, digest, and encoding routines, point your browser to http://www.cocoadev.com/index.pl?NSDataCategory. This extremely handy `NSData` category offers MD5, SHA-1, and Base32 solutions, among others.
- Although some of Apple's key networking technologies simply weren't finished in time for this chapter to cover them thoroughly, there should be enough information in this chapter's sections to get you started on the right path. The deprecated workarounds I did include should continue to work until at least iOS 6.*x*, but always port from deprecated routines as soon as practical.
- When working with networking, always think "threaded." Blocks and queues are your new best friends when it comes to creating positive user experiences in networked applications.
- Even when Apple provides Objective-C wrappers, as they do with `NSXMLParser`, it's not always the class you wanted or hoped for. Adapting classes is a big part of the iPhone programming experience.

From the Library of Eric Berna

This page intentionally left blank

From the Library of Eric Berna

16

Documents and Data Sharing

Under iOS, applications can share information and data as well as move control from one application to another using a variety of system features. Each application has access to a common system pasteboard that allows copying and pasting across apps. They can open documents from one app in another app that supports that format. They can declare custom URL scheme that can be embedded in text and web pages, and custom preferences that are centralized to the settings app. This chapter introduces the ways you can integrate document, defaults, and data sharing between applications. You'll see how to integrate these features into your applications and use them smartly to make your app a cooperative citizen of the iOS ecosystem.

Working with Uniform Type Identifiers

Uniform Type Identifiers (UTIs) form a central component of iOS information sharing. UTIs are a kind of string that helps identify resource types such as images and text. UTIs specify what kind of information is being used for common data objects. They do this without relying on older indicators, such as file extensions, MIME types, and file type metadata like `OSTypes`. UTIs replace these items with a newer and more flexible technology.

UTIs use a reverse-domain–style naming convention. Common Apple-derived identifiers look like this: e.g. `public.html` and `public.jpeg`. These refer, respectively, to HTML source text and JPEG images, which are specialized types of file information.

Each UTI can assume a more general or more specific role, as needed. Take the JPEG UTI, for example. A JPEG image (`public.jpeg`) is an image (`public.image`), which is in turn a kind of data (`public.data`), which is a kind of user-viewable (or listenable) content (`public.content`), which is a kind of item (`public.item`), the generic base type for UTIs. This hierarchy is called conformance. A more specific UTI is said to conform to a more general UTI through inheritance.

Figure 16-1 shows part of Apple's basic conformance tree. Any item lower down on the tree must conform to all of its parent data attributes. Declaring a parent UTI implies

From the Library of Eric Berna

that you support all of its children. So, an application that can open `public.data`, must be able to service text, movies, image files, and more.

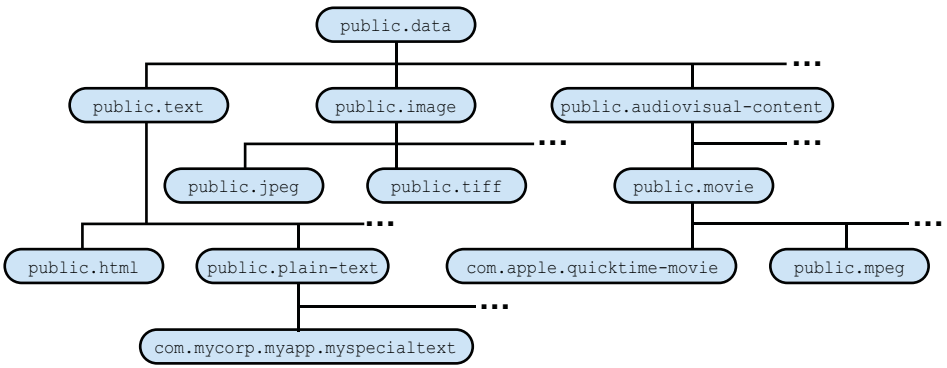


Figure 16-1 Apple's public UTI conformance tree.

UTIs allow multiple inheritance. An item can conform to more than one UTI parent. So you might imagine a data type that offers both text and image containers, which declares conformance to both.

There is no central registry for UTI items although each UTI should adhere to certain conventions. The `public` domain is reserved for iOS-specific types, common to most applications. Apple has generated a complete family hierarchy of public items. Add any third-party company-specific names by using standard reverse domain naming (e.g., `com.sadun.myCustomType` and `com.apple.quicktime-movie`).

Determining UTIs from File Extensions

The Mobile Core Services framework offers utilities that allow you to retrieve UTI information based on file extensions. Be sure to include the header files and link your apps to the framework when using these functions. The following code returns a preferred UTI when passed a path extension string. The preferred identifier is a single UTI string.

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

NSString *preferredUTIForExtension(NSString *ext)
{
    // Request the UTI via the file extension
    NSString *theUTI = (__bridge_transfer NSString *)
        UTTypeCreatePreferredIdentifierForTag(
            kUTTagClassFilenameExtension,
           (__bridge CFStringRef) ext, NULL);
    return theUTI;
}
```

From the Library of Eric Berna

You can also pass a MIME type instead of a file extension to `UTTypeCreatePreferredIdentifierForTag()` by using `kUTTagClassMIMEType` as the first argument. This function returns a preferred UTI for a given MIME type:

```
NSString *preferredUTIForMIMEType(NSString *mime)
{
    // Request the UTI via the file extension
    NSString *theUTI = (__bridge_transfer NSString *)
        UTTypeCreatePreferredIdentifierForTag(
            kUTTagClassMIMEType,
            (__bridge CFStringRef) mime, NULL);
    return theUTI;
}
```

Moving From UTI to Extension or MIME Type

To go the other way, producing a preferred extension or MIME types from a UTI, use `UTTypeCopyPreferredTagWithClass()`. The following functions return `jpeg` and `image/jpeg`, respectively, when passed `public.jpeg`:

```
NSString *extensionForUTI(NSString *aUTI)
{
    CFStringRef theUTI = (__bridge CFStringRef) aUTI;
    CFStringRef results =
        UTTypeCopyPreferredTagWithClass(
            theUTI, kUTTagClassFilenameExtension);
    return (__bridge_transfer NSString *)results;
}

NSString *mimeTypeForUTI(NSString *aUTI)
{
    CFStringRef theUTI = (__bridge CFStringRef) aUTI;
    CFStringRef results =
        UTTypeCopyPreferredTagWithClass(
            theUTI, kUTTagClassMIMEType);
    return (__bridge_transfer NSString *)results;
}
```

You must work at the leaf level with these functions, meaning at the level that declare the type extensions directly. Extensions are declared in property lists, where features like file extensions and default icons are described. (This is discussed later in this chapter.) So, for example, passing `public.text` or `public.movie` to the extension function return `nil`, while `public.plain-text` and `public.mpeg` return extensions of `txt` and `mpg`, respectively.

The former items live too high up the conformance tree, providing an abstract type than a specific implementation. There's no current API function to look down to find

From the Library of Eric Berna

items that descend from a given class that are currently defined for the application. You may want to file an enhancement request at bugreporter.apple.com. Surely, all the extensions and MIME types are registered somewhere (otherwise, how would the `UTTypeCopyPreferredTagWithClass()` look-up work in the first place?), so the ability to map extensions to more general UTIs should be possible.

Testing Conformance

You can test conformance using the `UTTypeConformsTo()` function. This function takes two arguments: a source UTI and a UTI to compare to, returning true if the first UTI conforms to the second. You can use this to test whether a more specific item is likely to confirm to a more general one. You can also test equality using `UTTypeEqual()`. Here's an example of how you might use conformance testing in real life, testing if a file path likely points to an image resource:

```
BOOL pathPointsToLikelyUTIMatch(NSString *path, CFStringRef theUTI)
{
    NSString *extension = path.pathExtension;
    NSString *preferredUTI = preferredUTIForExtension(extension);
    return (UTTypeConformsTo(
        (__bridge CFStringRef) preferredUTI, theUTI));
}

BOOL pathPointsToLikelyImage(NSString *path)
{
    return pathPointsToLikelyUTIMatch(path, CFSTR("public.image"));
}

BOOL pathPointsToLikelyAudio(NSString *path)
{
    return pathPointsToLikelyUTIMatch(path, CFSTR("public.audio"));
}
```

Retrieving Conformance Lists

`UTTypeCopyDeclaration()` offers the most general (and I think most useful) of all UTI functions in the iOS API. It returns a dictionary that includes the following keys:

- `kUTTypeIdentifierKey`—The UTI name itself, which you passed to the function (e.g., `public.mpeg`).
- `kUTTypeConformsToKey`—Any parents that the type conforms to (e.g., `public.mpeg` conforms to `public.movie`).
- `kUTTypeDescriptionKey`—A real-world description of the type in question if one exists (e.g., "MPEG movie").

From the Library of Eric Berna

- `kUTTypeTagSpecificationKey`—A dictionary of equivalent OSTypes (e.g., MPG, MPEG), file extensions (mpg, mpeg, mpe, m75, m15), and MIME types (video/mpeg, video/mpg, video/x-mpeg, video/x-mpg) for the given UTI.

In addition to these common keys, you may encounter more keys that specify imported and exported UTI declarations (`kUTImportedTypeDeclarationsKey` and `kUTExportedTypeDeclarationsKey`), icon resources to associate with the UTI (`kUTTypeIconFileKey`), a URL that points to a page describing the type (`kUTTypeReferenceURLKey`), and a version key that offers a version string for the UTI (`kUTTypeVersionKey`).

You can use the returned dictionary to ascend through the conformance tree to build an array that represents all the items that a given UTI conforms to. For example, the `public.mpeg` type conforms to `public.movie`, `public.audiovisual-content`, `public.data`, `public.item`, and `public.content`. These items are returned as an array from the `conformanceArray` function that follows:

```
// Build a declaration dictionary for the given type
NSDictionary *utiDictionary(NSString *aUTI)
{
    NSDictionary *dictionary =
        (__bridge_transfer NSDictionary *)
            UTTypeCopyDeclaration((__bridge CFStringRef) aUTI);
    return dictionary;
}

// Return an array where each member is guaranteed unique
// but that preserves the original ordering wherever possible
NSArray *uniqueArray(NSArray *anArray)
{
    NSMutableArray *copiedArray =
        [NSMutableArray arrayWithArray:anArray];

    for (id object in anArray)
    {
        [copiedArray removeObjectIdenticalTo:object];
        [copiedArray addObject:object];
    }

    return copiedArray;
}

// Return an array representing all UTIs that a given UTI conforms to
NSArray *conformanceArray(NSString *aUTI)
{
    NSMutableArray *results =
        [NSMutableArray arrayWithObject:aUTI];
```

From the Library of Eric Berna

```
    NSDictionary *dictionary = utiDictionary(aUTI);
    id conforms = [dictionary objectForKey:
        (__bridge NSString *)kUTTypeConformsToKey];

    // No conformance
    if (!conforms) return results;

    // Single conformance
    if ([conforms isKindOfClass:[NSString class]])
    {
        [results addObjectsFromArray:conformanceArray(conforms)];
        return uniqueArray(results);
    }

    // Iterate through multiple conformance
    if ([conforms isKindOfClass:[NSArray class]])
    {
        for (NSString *eachUTI in (NSArray *) conforms)
            [results addObjectsFromArray:conformanceArray(eachUTI)];
        return uniqueArray(results);
    }

    // Just return the one-item array
    return results;
}
```

Accessing the System Pasteboard

Pasteboards, also known as clipboards on some systems, provide a central OS feature for sharing data across applications. Users can copy data to the pasteboard in one application, switch tasks, and then paste that data into another application. Cut/copy/paste features are similar to those found in most operating systems. Users can also copy and paste within a single application, when switching between text fields or views.

The `UIPasteboard` class offers access to a shared device pasteboard and its contents. This snippet returns the general system pasteboard, which is appropriate for most general copy/paste use:

```
UIPasteboard *pb = [UIPasteboard generalPasteboard];
```

In addition to the general shared system pasteboard, the iPhone offers both application-specific pasteboards to better ensure data privacy, which do not extend beyond the application itself, and custom-named pasteboards that can be used across applications, but only with applications that know and utilize the pasteboard name key. Create app-specific pasteboards using `pasteboardWithUniqueName`, which returns an application pasteboard object that persists until the application quits.

From the Library of Eric Berna

Create custom pasteboards using `pasteboardWithName:create:`, which returns a pasteboard with the specified name. Use reverse-DNS naming for the pasteboard (e.g., `com.sadun.shared-application-pasteboard`). The create parameter specifies whether the system should create the pasteboard if it does not yet exist. This kind of pasteboard can persist beyond a single application run; set the persistent property to `YES` after creation. Use `removePasteboardWithName:` to destroy a pasteboard and free up the resources used by it.

Storing Data

Pasteboards can store one or more entries at a time. Each has an associated type, using the UTI to specify what kind of data is stored. For example, you might find `public.text` (and more specifically `public.utf8-plain-text`) to store text data, `public.url` for URL address, and `public.jpeg` for image data. These are among many other common data types used on iOS. The dictionary that stores the type and the data is called an `item`, and you can retrieve an array of all available items via the pasteboard's `items` property.

Query a pasteboard for its available types by sending it the `pasteboardTypes` message. This returns an array of types currently stored on the pasteboard.

```
NSArray *types = [pb pasteboardTypes];
```

You can set data on the pasteboard and associate a type by passing an `NSData` object and a UTI that describes a type the data conforms to. Alternatively, for property list objects (i.e., string, date, array, dictionary, number, or URL), set an `NSValue` via `setValue:forPasteboardType:`. These property list objects are stored internally somewhat differently than their raw-data cousins, giving rise to the method differentiation.

```
[[UIPasteboard generalPasteboard]
    setData;theData forPasteboardType:theUTI];
```

Storing Common Types

Pasteboards are further specialized for several data types, which represent the most commonly used pasteboard items. These are colors (not a property list "value" object), images (also not a property list "value" object), strings, and URLs. The `UIPasteboard` class provides specialized getters and setters to make it easier to handle these items. You can treat each of these as properties of the pasteboard, so you can set and retrieve them using dot notation. What's more each property has a plural form, allowing you to access those items as arrays of objects.

Pasteboard properties greatly simplify using the system pasteboard for the most common use-cases. The property accessors include

- `string`—Set or retrieve the first string on the pasteboard
- `strings`—Set or retrieve an array of all strings on the pasteboard
- `image`—Set or retrieve the first image on the pasteboard

From the Library of Eric Berna

- `images`—Set or retrieve an array of all images on the pasteboard
- `URL`—Set or retrieve the first URL on the pasteboard
- `URLs`—Set or retrieve an array of all URLs on the pasteboard
- `color`—Set or retrieve the first color on the pasteboard
- `colors`—Set or retrieve an array of all colors on the pasteboard

Retrieving Data

When using one of the four special classes, simply use the associated property to retrieve data from the pasteboard. Otherwise you can fetch data using the `dataForPasteboardType:` method. This method returns the data from the first item whose type matches the UTI sent as a parameter. Any other matching items in the pasteboard are ignored.

Should you need to retrieve all matching data, recover an `itemSetWithPasteboardTypes:` and then iterate through the set to retrieve each dictionary. Recover the data type for each item from the single dictionary key and the data from its value.

As mentioned, `UIPasteboard` offers two approaches for pasting to the pasteboard depending on whether the information being pasted is a property list object or raw data. Use `setValueForPasteboardType:` for property list objects, which include strings, dates, numbers, dictionaries, arrays, and URLs. For general data, use `setData:forPasteboardType:`.

When pasteboards are changed, they issue a `UIPasteboardChangedNotification`, which you can listen into via a default `NSNotificationCenter` observer. You can also watch custom pasteboards and listen for their removal via `UIPasteboardRemovedNotification`.

> **Note**
>
> If you want to successfully paste text data to Notes or Mail, make sure you use `public.utf8-plain-text` as your UTI of choice when storing information to the pasteboard. Using the `string` or `strings` properties automatically enforces this UTI.

Recipe: Passively Updating the Pasteboard

iOS selection and copy interface are not, frankly, the most streamlined elements of the operating system. There are times when you want to simplify matters for your user while preparing content that's meant to be shared to other applications. Consider Recipe 16-1. It allows the user to use a local editor to enter and edit text, while automating the process of updating the pasteboard. When the watcher is active (toggled by a simple button tap), the text is sent to the pasteboard on each edit. This is accomplished by implementing a text view delegate method (`textViewDidChange:`) that responds to edits by automatically assigning those changes to the pasteboard (`updatePasteboard`).

From the Library of Eric Berna

Recipe 16-1 **Creating an Automatic Text-Entry to Pasteboard Solution**

```
@implementation TestBedViewController
- (void) updatePasteboard
{
   // Copy the text to the pasteboard when the watcher is enabled
    if (enableWatcher)
        [UIPasteboard generalPasteboard].string = textView.text;
}

- (void)textViewDidChange:(UITextView *)textView
{
    // Delegate method calls for an update
    [self updatePasteboard];
}

- (void) toggle: (UIBarButtonItem *) bbi
{
    // switch between standard and auto-copy modes
    enableWatcher = !enableWatcher;
    bbi.title = enableWatcher ? @"Stop Watching" : @"Watch";
}

- (void) loadView
{
    [super loadView];

    // Build a text view and set the delegate to self
    textView = [[UITextView alloc] initWithFrame:CGRectZero];
    textView.font = [UIFont fontWithName:@"Futura"
        size: IS_IPAD ? 32.0f : 16.0f];
    textView.delegate = self;
    [self.view addSubview:textView];

    // Add a toggle button
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Watch", @selector(toggle:));
}

- (void) viewDidAppear:(BOOL)animated
{
    // Always perform an update whenever the view appears
    textView.frame = self.view.bounds;
    [self updatePasteboard];
}

- (void) didRotateFromInterfaceOrientation:
```

From the Library of Eric Berna

```
    (UIInterfaceOrientation)fromInterfaceOrientation
{
    // Update presentation
    [self viewDidAppear:NO];
}
@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 this book, go to the folder for Chapter 16 and open the project for this recipe.

Recipe: Enabling Document File Sharing

iOS documents aren't trapped in their sandbox. You can and should share them with your users. Offer users direct control over their documents and access to any material they may have created on-device. A simple Info.plist setting allows iTunes to display the contents of a user's Documents folder and lets those users add and remove material on demand.

To enable file sharing, add a `UIFileSharingEnabled` key to the application's Info.plist and set its value to `YES`, as shown in Figure 16-2. When working with non-raw keys and values, this item is called "Application supports iTunes file sharing." iTunes lists all applications that declare file sharing support in each device's Apps tab, as shown in Figure 16-3.

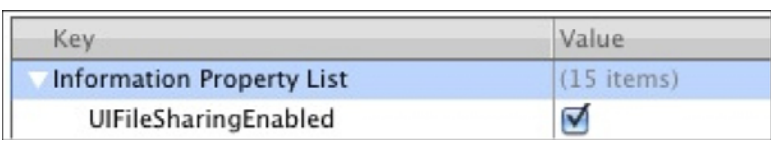


Figure 16-2 Enable `UIFileSharingEnabled` to allow user-access to the Documents folder via iTunes.

You cannot specify which kinds of items are allowed to be in the Documents folder. Users can add any materials they like, and remove any items they want to. What they cannot do, however, is navigate through subfolders using the iTunes interface. Notice the Inbox folder in Figure 16-3. This is an artifact left over from application-to-application document sharing, and it should not be there. Users cannot manage that data directly, and you should not leave the subfolder there to confuse them.

From the Library of Eric Berna

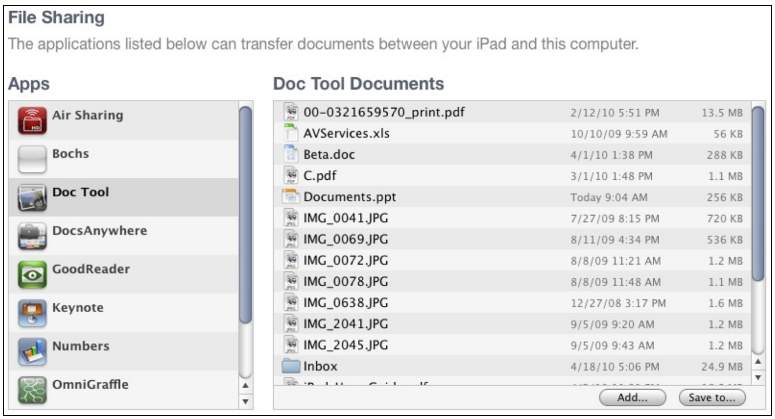


Figure 16-3 Each installed application that declares `UIFileSharingEnabled` is listed in iTunes.

Users can, however, copy that data out in its entirety to the desktop by selecting the folder in iTunes and choosing *Save to*….This allows the user to copy folders to local storage and inspect their contents. Users cannot, at this time, upload folders to the device.

This option includes the Inbox folder. Users may copy and inspect the Inbox, but they cannot delete it in iTunes, the way they can delete other files and folders. Nor should your application write files directly to the Inbox. Respect the Inbox's role, which is to capture any incoming data from other applications. When you implement file-sharing support, always check for an Inbox on resuming active status and process that data to clear out the inbox and remove it.

As a developer, you have access not just to the Documents folder, but also to the entire application sandbox. Use the Xcode Organizer (Command-2) > Devices tab > *Device* > Applications > *Application Name* to browse, upload, and download files to and from the sandbox.

You can test basic file sharing by adding a `UIFileSharingEnabled` property to an application and loading data to your Documents folder. Recipe 16-2 builds two files, creating one in the main Documents folder and adding a second in a custom test folder. Once those files are created, use iTunes to inspect, download, and delete them.

Recipe 16-2 T**esting UIFileSharingEnabled by Building Files and Folders**

```
- (void) buildItems
{
    // Define the paths
    NSString *documentsPath = [NSHomeDirectory()
        stringByAppendingPathComponent:@"Documents"];
    NSString *testFolderPath = [documentsPath
        stringByAppendingPathComponent:@"TestFolder"];
```

From the Library of Eric Berna

```
    NSString *filePath1 = [testFolderPath
        stringByAppendingPathComponent:@"Hello.txt"];
    NSString *filePath2 = [documentsPath
        stringByAppendingPathComponent:@"Hello.txt"];

    NSError *error;
    BOOL success;

    // Create a folder
    if (![[NSFileManager defaultManager]
        fileExistsAtPath:testFolderPath])
    {
        success = [[NSFileManager defaultManager]
            createDirectoryAtPath:testFolderPath
            withIntermediateDirectories:NO
            attributes:nil error:&error];
        if (!success)
        {
            NSLog(@"Error creating test folder: %@",
                error.localizedFailureReason);
            return;
        }
    }

    // Now put a file there
    success = [@"Hello world\n" writeToFile:filePath1
        atomically:YES encoding:NSUTF8StringEncoding
        error:&error];
    if (!success)
    {
        NSLog(@"Error writing file 1: %@",
            error.localizedFailureReason);
        return;
    }

    // Put a file into the main Documents folder too
    success = [@"Hello world\n" writeToFile:filePath2
        atomically:YES encoding:NSUTF8StringEncoding
        error:&error];
    if (!success)
    {
        NSLog(@"Error writing file 2: %@",
            error.localizedFailureReason);
        return;
    }

    NSLog(@"Success. Files and folder created.");
}
```

From the Library of Eric Berna

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 this book, go to the folder for Chapter 16 and open the project for this recipe.

Recipe: Monitoring the Documents Folder

At some point in the future, you will be able to use a simple `NSMetadataQuery` monitor to watch your Documents folder and report updates. At the time this book was written, that metadata surveillance was not yet extended beyond iCloud for use with other folders. Code ported from OS X fails to work as expected on iOS. As I write, there are precisely two available search domains for iOS: the ubiquitous data scope, and the ubiquitous documents scope (i.e., iCloud and iCloud). See Chapter 18, "iCloud Basics," for examples of using metadata queries.

Until that functionality arrives in iOS, you can still fall back to kqueue. This older technology provides scalable event notification (and, apparently, what powers Apple's Grand Central Dispatch). With kqueue, you can monitor add and clear events in the Documents folder. This roughly equates to looking for files being added and deleted, which are the primary kinds of updates you want to react to. Recipe 16-3 presents a kqueue implementation for watching the Documents folder.

The following snippet demonstrates how you might use this watcher to update a file list in your GUI. Whenever the contents change, an update notification allows the app to refresh those directory contents listings.

```
- (void) contentsChanged: (NSNotification *) notification
{
    NSString *docPath =
        [NSSearchPathForDirectoriesInDomains(
            NSDocumentDirectory, NSUserDomainMask, YES)
            objectAtIndex:0];
    NSArray *array = [[NSFileManager defaultManager]
        contentsOfDirectoryAtPath:docPath error:nil];
    textView.text = [array componentsJoinedByString:@"\n"];
}

- (void) loadView
{
    [super loadView];
    self.view.backgroundColor = [UIColor whiteColor];

    // Establish a text view
    textView = [[UITextView alloc] initWithFrame:CGRectZero];
    textView.font = [UIFont fontWithName:@"Futura"
        size: IS_IPAD ? 32.0f : 16.0f];
```

From the Library of Eric Berna

```
    textView.delegate = self;
    [self.view addSubview:textView];

    // Start monitoring
    [[NSNotificationCenter defaultCenter]
        addObserver:self selector:@selector(contentsChanged:)
        name:kDocumentChanged object:nil];
    NSString *docPath =
        [NSSearchPathForDirectoriesInDomains(
            NSDocumentDirectory, NSUserDomainMask, YES)
            objectAtIndex:0];
    helper = [DocWatchHelper watcherForPath:docPath];

    [self contentsChanged:nil];
}
```

This recipe is best tested by connecting a device to iTunes and adding and removing items. The device's onboard file list should update to reflect those changes in real time.

Recipe 16-3 **Using a kqueue File Monitor**

```
#import <fcntl.h>
#import <sys/event.h>

#define kDocumentChanged \
    @"DocumentsFolderContentsDidChangeNotification"

@interface DocWatchHelper : NSObject
{
    CFFileDescriptorRef kqref;
    CFRunLoopSourceRef  rls;
}
@property (strong) NSString *path;
+ (id) watcherForPath: (NSString *) aPath;
@end

@implementation DocWatchHelper
@synthesize path;

- (void)kqueueFired
{
    int             kq;
    struct kevent   event;
    struct timespec timeout = { 0, 0 };
    int             eventCount;

    kq = CFFileDescriptorGetNativeDescriptor(self->kqref);
    assert(kq >= 0);
```

From the Library of Eric Berna

```
    eventCount = kevent(kq, NULL, 0, &event, 1, &timeout);
    assert( (eventCount >= 0) && (eventCount < 2) );

    if (eventCount == 1)
        [[NSNotificationCenter defaultCenter]
            postNotificationName:kDocumentChanged
            object:self];

    CFFileDescriptorEnableCallBacks(self->kqref,
        kCFFileDescriptorReadCallBack);
}

static void KQCallback(CFFileDescriptorRef kqRef,
    CFOptionFlags callBackTypes, void *info)
{
    DocWatchHelper *helper =
        (DocWatchHelper *)(__bridge id)(CFTypeRef) info;
    [helper kqueueFired];
}

- (void) beginGeneratingDocumentNotificationsInPath:
    (NSString *) docPath
{
    int                     dirFD;
    int                     kq;
    int                     retVal;
    struct kevent           eventToAdd;
    CFFileDescriptorContext context =
        { 0, (void *)(__bridge CFTypeRef) self,
            NULL, NULL, NULL };

    dirFD = open([docPath fileSystemRepresentation], O_EVTONLY);
    assert(dirFD >= 0);

    kq = kqueue();
    assert(kq >= 0);

    eventToAdd.ident  = dirFD;
    eventToAdd.filter = EVFILT_VNODE;
    eventToAdd.flags  = EV_ADD | EV_CLEAR;
    eventToAdd.fflags = NOTE_WRITE;
    eventToAdd.data   = 0;
    eventToAdd.udata  = NULL;

    retVal = kevent(kq, &eventToAdd, 1, NULL, 0, NULL);
    assert(retVal == 0);
```

From the Library of Eric Berna

```
    self->kqref = CFFileDescriptorCreate(NULL, kq,
        true, KQCallback, &context);
    rls = CFFileDescriptorCreateRunLoopSource(
        NULL, self->kqref, 0);
    assert(rls != NULL);

    CFRunLoopAddSource(CFRunLoopGetCurrent(), rls,
        kCFRunLoopDefaultMode);
    CFRelease(rls);

    CFFileDescriptorEnableCallBacks(self->kqref,
        kCFFileDescriptorReadCallBack);
}

- (void) dealloc
{
    self.path = nil;
    CFRunLoopRemoveSource(CFRunLoopGetCurrent(), rls,
        kCFRunLoopDefaultMode);
    CFFileDescriptorDisableCallBacks(self->kqref,
        kCFFileDescriptorReadCallBack);
}

+ (id) watcherForPath: (NSString *) aPath
{
    DocWatchHelper *watcher = [[self alloc] init];
    watcher.path = aPath;
    [watcher beginGeneratingDocumentNotificationsInPath:aPath];
    return watcher;
}
@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 this book, go to the folder for Chapter 16 and open the project for this recipe.

The Document Interaction Controller

The `UIDocumentInteractionController` class allows applications to present a menu of options to users, allowing them to use document files in a variety of ways. With this class, users can take advantage of

From the Library of Eric Berna

- iOS application-to-application document sharing (i.e., "open this document in…*some app*")
- Document preview using iOS's built-in QuickLook technology
- Copying documents to the pasteboard or printing them

The controller is presented as a small menu, which allows users to specify how they want to interact with the given document. Figure 16-4 shows presentations on the iPad and iPhone.



Figure 16-4 The `UIDocumentInteractionController` shown in its options style for iPad (left) and iPhone (right).

The controller offers two basic menu styles. The "open" style offers only "open in" choices, using the menu space to provide as many destination choices as possible. The "options" style (refer to Figure 16-4) provides a list of all interaction options including open-in, quick-look, and any supported actions. Figure 16-5 shows the basic options available via these controllers.

Creating Document Interaction Controller Instances

Each document interaction controller is created to present options for a file, typically in the user's Documents folder. You supply a URL pointing to the file and present the controller, typically from a bar button. The bar button presentation style works across platforms and simplifies coding, even though the iPhone implementation scrolls the controller view up from the bottom of the screen as an action sheet.

As a rule, I use a local instance variable to hold onto any current controllers and popovers, dismissing them before introducing any new choice items. Keep that in mind as you read through this section and the ones that follow it.

From the Library of Eric Berna

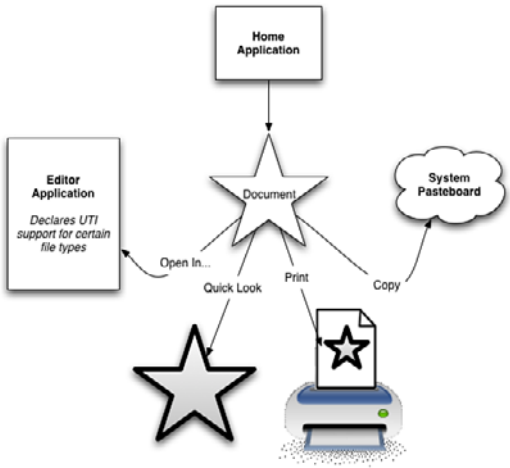


Figure 16-5 The document interaction controller provides access from your application to a variety of system services, as well as to other applications that support editing and viewing a document.

Thank you, Omni Group, for providing a copy of OmniGraffle to help build this diagram.

The following snippets show how to create and present both styles of controllers. In each case, any previous controller is dismissed, a new controller created, and the delegate assigned to the primary view controller. The presentation method determine whether the controller is shown as an options menu (`presentOptionsMenuFromBar-ButtonItem:animated:`) or as an open-in menu (`presentOpenInMenuFromBar-ButtonItem:animated:`).

```
- (void) options: (UIBarButtonItem *) bbi
{
    if (dic)
        [dic dismissMenuAnimated:NO];

    dic = [UIDocumentInteractionController
        interactionControllerWithURL:
            [NSURL fileURLWithPath:path]];
    dic.delegate = self;

    [dic presentOptionsMenuFromBarButtonItem:bbi animated:YES];
}

- (void) open: (UIBarButtonItem *) bbi
{
    if (dic)
        [dic dismissMenuAnimated:NO];
```

From the Library of Eric Berna

```
    dic = [UIDocumentInteractionController
        interactionControllerWithURL:
            [NSURL fileURLWithPath:path]];
    dic.delegate = self;

    [dic presentOpenInMenuFromBarButtonItem:bbi animated:YES];
}
```

I always implement dismissal callbacks in order to round out my local instance variable approach.

```
- (void) documentInteractionControllerDidDismissOpenInMenu:
     (UIDocumentInteractionController *) controller
    { dic = nil; }

- (void) documentInteractionControllerDidDismissOptionsMenu:
    (UIDocumentInteractionController *) controller
    { dic = nil; }
```

Document Interaction Controller Properties

Each document interaction controller offers a number of properties, which can be used in your delegate callbacks:

- The `URL` property allows you to query the controller for the file it is servicing. This is the same URL you pass when creating the controller.
- The `UTI` property is used to determine which apps can open the document. It uses the automatic routines discussed earlier in the chapter to find the most preferred UTI match based on the file name and metadata, although you can override this in code to set the property manually.
- The `name` property provides the last path component of the URL, offering a quick way to provide a user-interpretable name without having to manually strip the URL yourself.
- Use the `icons` property to retrieve an icon for the file type. Applications that declare support for certain file types provide image links in their declaration (as you'll see shortly, in the discussion about declaring file support). These images correspond to the values stored for the `kUTTypeIconFileKey` key, as discussed earlier in this chapter.
- The `annotation` property provides a way to pass custom data along with a file to any application that will open the file. There are no standards for using this property, although the item must be set to some top-level property list object, namely dictionaries, arrays, data, strings, numbers, and dates. Because there are no community standards yet, use of this property tends to be minimal except where developers use the information across their own suite of published apps.

From the Library of Eric Berna

Providing Document Quick Look Support

Add Quick Look support to your apps by providing a trio of delegate callbacks. In these methods, you declare which view controller will be used to present the preview, which view will host it, and the frame for the preview size. You may have occasional compelling reasons to use a child view controller with limited screen presence on the iPad (such as in a split view, with the preview in just one portion), but for the iPhone, there's almost never any reason not to allow the preview to take over the entire screen.

```
#pragma mark QuickLook
- (UIViewController *)
    documentInteractionControllerViewControllerForPreview:
       (UIDocumentInteractionController *)controller
{
    return self;
}

- (UIView *)
    documentInteractionControllerViewForPreview:
    (UIDocumentInteractionController *)controller
{
    return self.view;
}

- (CGRect)
    documentInteractionControllerRectForPreview:
    (UIDocumentInteractionController *)controller
{
    return self.view.frame;
}
```

Providing Printer and Copy Support

The interaction controller class currently offers two possible actions: namely, printing and copying. These actions are sent as selector parameters, namely `copy:` and `paste:`, to a pair of delegate methods. The first method specifies whether each action (and any other future action that Apple might support) can be performed. If so, it can be added to the actions menu. If not, that item is skipped. This method is called by the controller as it builds the menu and before it's presented.

In the following snippet, each action request is handled separately. In the case of copying, the file's size is checked, limiting copies to under 5MB. This approach assumes the application will be pasting actual data, rather than a URL pointing to the document. For the later case, you might simply return `YES` after checking for the file's existence rather than calculating its size.

The print action here is set to be contingent on whether printing is available. You can create a pseudo printer for your system in the Simulator by choosing File > Open Printer

From the Library of Eric Berna

Simulator. I personally use Ecamm's $20 Printopia product (www.ecamm.com/mac/printopia/) to enable AirPrinting to my normal Brother printer. It acts as an AirPrint server and then redirects any print requests to my normally non-AirPrint–enabled hardware. Printopia also allows me to "print" to DropBox, which is also handy, especially for saving paper.

```
- (BOOL) documentInteractionController:
    (UIDocumentInteractionController *)controller
    canPerformAction:(SEL)action
{
    if (action == @selector(copy:))
    {
        // Copy only items under 5MB
        long long fileSize =
            [[NSFileManager defaultManager]
                attributesOfItemAtPath:path error:nil].fileSize;
        return (fileSize < 5 * 1024 * 1024);
    }

    if (action == @selector(print:))
    {
        return [UIPrintInteractionController isPrintingAvailable];
    }

    return YES;
}
```

The actual copying and printing is performed in a second delegate callback. In this method, once again, the requested action is checked before performing any task. This application sample manages images, using the pasteboard's `image` property to perform the copy action. As with the previous method, you'll need to adapt this to whatever data you're actually working with.

On the printing end of things, the printer is first checked to see if the item can be printed. If not, it logs an error and returns. Otherwise, it chooses the appropriate print controller presentation on a per-platform basis and presents it. All further printing control is handled through the `UIPrintInteractionController` class. If you want to act as the print controller's delegate, there is a full suite of callbacks that you can implement. The callbacks let you know when the print job has started and stopped. For the most part, I just let it run in its default and let these things be as they may.

```
- (BOOL) documentInteractionController:
    (UIDocumentInteractionController *)controller
    performAction:(SEL)action
{
    if (action == @selector(copy:))
    {
```

From the Library of Eric Berna

```
        // Copy the image to the pasteboard
        UIImage *image = [UIImage imageWithContentsOfFile:path];
        [UIPasteboard generalPasteboard].image = image;
        return YES;
    }

    if (action == @selector(print:))
    {
        // Check if the item can be printed
        if (![UIPrintInteractionController
            canPrintURL:controller.URL])
        {
            NSLog(@"Item is not printable: %@", controller.URL);
            return NO;
        }

        [UIPrintInteractionController
            sharedPrintController].printingItem = controller.URL;
        if (IS_IPAD)
            [[UIPrintInteractionController sharedPrintController]
                presentFromBarButtonItem:
                    self.navigationItem.rightBarButtonItem
                animated:YES completionHandler:nil];
        else
            [[UIPrintInteractionController sharedPrintController]
                presentAnimated:YES completionHandler:nil];
        }
        return YES;
    }

    return YES;
}
```

Recipe: Checking for the Open Menu

When using a document interaction controller, be aware of this rule of thumb: The options menu will almost always provide some kind of menu choices, especially if you implement the action and quick look callbacks. You may not, however, have any open-in options to work with.

That happens when there are no applications installed on a device that support the file type you are working with. This may be caused by an obscure file type, or more often, because the user has not yet purchased and installed any applications that offer that file type.

Because of this, you'll want to check whether to offer an "Open" menu option in the first place. Recipe 16-4 performs a rather ugly test to see if external apps will offer

From the Library of Eric Berna

themselves as presenters and editors for a given URL. What it does is this: It creates a new, temporary controller, and attempts to present it. If it succeeds, there are conforming file destinations installed on the device and the method immediately dismisses the controller. If not, there are no such apps, and any "Open In" buttons should be disabled.

This is obviously a rather dreadful implementation but it has the advantage of being able to test as you lay out your interface or when you start working with a new file. I encourage you to file an enhancement request at bugreporter.apple.com.

Recipe 16-4 **Checking for External Document Support**

```
// Determine whether a file URL can be opened in external apps
- (BOOL)canOpen: (NSURL *)fileURL
{
    UIDocumentInteractionController *tmp =
        [UIDocumentInteractionController
            interactionControllerWithURL:fileURL];
    tmp.delegate = self;
    BOOL success = [tmp presentOpenInMenuFromRect:CGRectZero
        inView:self.view animated:NO];
    [tmp dismissMenuAnimated:NO];
    return success;
}
```

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 this book, go to the folder for Chapter 16 and open the project for this recipe.

Declaring Document Support

Documents can declare their support for certain file types in their Info.plist property list. This kind of declaration is part of the Launch Services system and is similar to declarations made on OS X. Figure 16-6 shows one such document declaration.

The `CFBundleDocumentTypes` array lists each file type supported by the application. Each item in this list is a dictionary that corresponds to the declarations you read in earlier using the `UTTypeCopyDeclaration()` function. Most importantly, the dictionary needs to list its content types, namely the supported UTI or UTIs. In Figure 16-6, that type is `public.data`, which means the application will allow itself to open pretty much any iOS file, and its name, here `com.sadun.public-data`. As you can tell, this is an arbitrary string.

To associate an icon with the file data, you can add bundle images to the dictionary's `CFBundleTypeIconFiles` array. For simplicity, this declaration uses the application's icon, but you can create more document-specific icons as required.

From the Library of Eric Berna

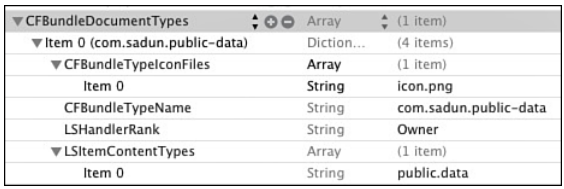


Figure 16-6 Declare supported document types in your Info.plist property list.

The handler rank key specifies how to rank the application. Choices include `Owner`, `Default`, and `Alternative`. The rules for these are as follows. An owner should be the natural creator and editor of a given file. I use owner in this example just because it's more or less to be guaranteed to be given priority for opening documents, which is handy when testing out application-to-application document sharing. The default application says to launch services that it's the most preferred application for opening a file type, even if it is not the document creator. For the most part, you'll want to use "alternative", which states that your application is one of many that can display or edit a given file type.

Recipe: Implementing Document Support

If your application provides document support, each time it becomes active, you should check to see if an Inbox has appeared in the Documents folder. If so, you should move elements out of that inbox and to where they belong, typically in the main Documents directory. Once the inbox has been cleared, delete it. This provides the best user experience, especially in terms of any file sharing through iTunes where the Inbox and its role may confuse users.

```
- (void)applicationDidBecomeActive:(UIApplication *)application
{
    // perform inbox test here
}
```

When moving items to Documents, you'll want to check for name conflicts, and use an alternative path name (typically by appending a hyphen followed by a number) to avoid overwriting any existing file. The following method helps find an alternative name for a destination path. It gives up after a thousand attempts, but seriously, none of your users should be hosting that many duplicate document names. If they do, there's something deeply wrong with your overall application design.

```
- (NSString *) findAlternativeNameForPath: (NSString *) path
{
    NSString *ext = path.pathExtension;
    NSString *base = [path stringByDeletingPathExtension];
```

From the Library of Eric Berna

```
    for (int i = 0; i < 999; i++)
    {
        NSString *dest =
            [NSString stringWithFormat:@"%@-%d.%@", base, i, ext];

        // if the file does not yet exist, use this destination path
        if (![[NSFileManager defaultManager]
            fileExistsAtPath:dest])
            return dest;
    }

    NSLog(@"Exhausted possible names for file %@. Bailing.",
        path.lastPathComponent);
    return nil;
}
```

Recipe 16-5 walks through the ugly details of scanning for the Inbox and moving files into place. It removes the Inbox once it is emptied. As you can see, any method like this is file manager-intensive. It primarily involves handling all the error combination possibilities that might pop-up throughout the task. This should run quickly for small file support. If you must handle large files, such as video or audio, make sure to perform this processing on its own operation queue.

If you plan to support `public.data` files (i.e., will open anything), as this recipe does, you may want to display those files using `UIWebView` instances. Refer to Technical Q&A QA1630 (http://developer.apple.com/library/ios/#qa/qa1630) for details about which document types iOS can and cannot display in those views. Web views can present most audio and video assets, as well as Excel, Keynote, Numbers, Pages, PDF, Powerpoint, and Word resources in addition to simple HTML.

Recipe 16-5 **Handling Incoming Documents**

```
- (void) cleanInboxIfNeeded
{
    NSString *documentsPath = [NSHomeDirectory()
        stringByAppendingPathComponent:@"Documents"];
    NSString *inboxPath = [documentsPath
        stringByAppendingPathComponent:@"Inbox"];

    // Does the Inbox exist? If not, we're done
    BOOL isDir;
    if (![[NSFileManager defaultManager]
        fileExistsAtPath:inboxPath isDirectory:&isDir])
        return;

    NSError *error;
    BOOL success;
```

From the Library of Eric Berna

```
// If the Inbox is not a folder, remove it.
if (!isDir)
{
    success = [[NSFileManager defaultManager]
        removeItemAtPath:inboxPath error:&error];
    if (!success)
    {
        NSLog(@"Error deleting Inbox file (not directory): %@",
            error.localizedFailureReason);
        return;
    }
}

// Retrieve a list of files in the Inbox
NSArray *fileArray = [[NSFileManager defaultManager]
    contentsOfDirectoryAtPath:inboxPath error:&error];
if (!fileArray)
{
    NSLog(@"Error reading contents of Inbox: %@",
        error.localizedFailureReason);
    return;
}

// Remember the number of items
NSUInteger initialCount = fileArray.count;

// Iterate through each file, moving it to Documents
for (NSString *filename in fileArray)
{
    NSString *source =
        [inboxPath stringByAppendingPathComponent:filename];
    NSString *dest =
        [documentsPath stringByAppendingPathComponent:filename];

    // Is the file already there?
    BOOL exists =
        [[NSFileManager defaultManager] fileExistsAtPath:dest];
    if (exists) dest = [self findAlternativeNameForPath:dest];
    if (!dest)
    {
        NSLog(@"Error. File name conflict not resolved");
        continue;
    }

    // Move file into place
    success = [[NSFileManager defaultManager]
```

From the Library of Eric Berna

```
            moveItemAtPath:source toPath:dest error:&error];
        if (!success)
        {
            NSLog(@"Error moving file from Inbox: %@",
                error.localizedFailureReason);
            continue;
        }
    }

    // Inbox should now be empty
    fileArray = [[NSFileManager defaultManager]
        contentsOfDirectoryAtPath:inboxPath error:&error];
    if (!fileArray)
    {
        NSLog(@"Error reading contents of Inbox: %@",
            error.localizedFailureReason);
        return;
    }

    if (fileArray.count)
    {
        NSLog(@"Error clearing Inbox. %d items remain",
            fileArray.count);
        return;
    }

    // Remove the inbox
    success = [[NSFileManager defaultManager]
        removeItemAtPath:inboxPath error:&error];
    if (!success)
    {
        NSLog(@"Error removing inbox: %@",
            error.localizedFailureReason);
        return;
    }

    NSLog(@"Moved %d items from the Inbox", initialCount);
}

// Check for the Inbox each time the app becomes active
- (void)applicationDidBecomeActive:(UIApplication *)application
{
    [self cleanInboxIfNeeded];
}
```

From the Library of Eric Berna

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 this book, go to the folder for Chapter 16 and open the project for this recipe.

Exported Type Declarations

You can declare custom UTIs by providing an exported type declaration in your Info.plist property list. These type specifications mirror the ones in Figure 16–7, which is used for declaring document type support. Figure 16–7 shows a type declaration for JPEG images.

Again, you see each of the keys that can be retrieved via the `UTTypeCopyDeclaration()` function. Here, those items are specified to the system as a preliminary declaration. You can easily adapt this template to create your own custom file types and associated UTIs as needed. The most common reason for declaring a new type is to introduce support for file formats supported on other platforms.

```
<key>UTExportedTypeDeclarations</key>
    <array>
        <dict>
            <key>UTTypeIdentifier</key>
            <string>public.jpeg</string>
            <key>UTTypeReferenceURL</key>
            <string>http://www.w3.org/Graphics/JPEG/</string>
            <key>UTTypeDescription</key>
            <string>JPEG image</string>
            <key>UTTypeIconFile</key>
            <string>public.jpeg.icns</string>
            <key>UTTypeConformsTo</key>
            <array>
                <string>public.image</string>
                <string>public.data</string>
            </array>
            <key>UTTypeTagSpecification</key>
            <dict>
                <key>com.apple.ostype</key>
                <string>JPEG</string>
                <key>public.filename-extension</key>
                <array>
                    <string>jpeg</string>
                    <string>jpg</string>
                </array>
                <key>public.mime-type</key>
                <string>image/jpeg</string>
            </dict>
        </dict>
    </array>
```

Figure 16-7 Declaring the JPEG UTI.

From the Library of Eric Berna

Creating URL-Based Services

Apple's built-in applications offer a variety of services that can be accessed via URL calls. You can ask Safari to open Web pages, Maps to show a map, or use the `mailto:` style URL to start composing a letter in Mail. A URL scheme refers to the first part of the URL that appears before the colon, such as `http:` or `ftp:`.

These services work because iOS knows how to match URL schemes to applications. A URL that starts with `http:` opens in Mobile Safari. The `mailto:` URL always links to Mail. What you may not know is that you can define your own URL schemes and implement them in your applications. Not all standard schemes are supported on the iPhone. The FTP scheme is not available for use.

Custom schemes allow applications to launch whenever Mobile Safari (or another application) opens a URL of that type. For example, should your application register xyz, any `xyz:` links go directly to your application for handling, where they're passed to the application delegate's URL opening method (after first passing through the launch-with-options method; you'll read more about these details later). You do not have to add any special coding there. If all you want to do is run an application, adding the scheme and opening the URL enables cross-application launching.

Handlers extend launching to allow applications to do something with the URL that's been passed to it. They might open a specific data file, retrieve a particular name, display a certain image, or otherwise process information included in the call.

Using URL Schemes

The advantages of scheme-based launching are many. Take the Iconfactory's Twitterrific, for example. Early in its development, developer Craig Hockenberry introduced a custom service that let users and third-party developers launch his application and open a pre-filled, ready-to-post tweet.

In days before Twitter was integrated at the system level, this let developers add Twitter support to their applications without any programming. Because Twitterrific already stored the sensitive username and password information, all that you had to supply was the body text. Control passed to Twitterrific, which took over and allowed users to finish tweeting. When done, users quit Twitterrific and might have returned to the original application if desired.

URL-based services work best when they provide some kind of performance boost or data leverage. In the case of Twitterrific, it wasn't about tweeting. It takes very little code to tweet. (Refer to Chapter 15, "Networking.") It was about freeing the developer from responsibility of securely storing user credentials or forcing users to enter those credentials on each use.

Service Downsides

It's not all good news on the services front; there's definitely a downside to third-party services. Once your application depends on a service, you basically force your users to download a second application. And that application may not always be the one they

From the Library of Eric Berna

want to use. Consider the loyal Echofon user who might not have had Twitterrific installed on his or her phone. If your app had demanded Twitterrific, you may have met resistance.

Any features that depend on third-party services must always be optional. Consider if, for example, Echofon had introduced its own tweeting URL scheme. When you add service-based features to your application, make those features robust and flexible enough to allow users to choose their preferred client.

Another downside is this: iOS applications cannot tell what schemes are available a priori. There's no way to poll for on-offer services. Apple provides no public registry that you can scan through to see what's out there. Using services is basically a matter of trust.

That being said, you can test whether a URL service is available. If the `UIApplication`'s `canOpenURL:` method returns `YES`, you are guaranteed that `openURL:` will be able to launch another application to open that URL. You are not guaranteed that the URL is valid, only that its scheme is registered properly to an existing application.

```
if ([[UIApplication sharedApplication] canOpenURL:aURL])
    [[UIApplication sharedApplication] openURL:aURL];
```

Cross-Promotion

There is another important business-oriented aspect to scheme-based launching, namely cross-promotion. Defining URL schema allows your application to test whether other applications exist from your company's lineup. If the application cannot handle the URL (i.e., `canOpenURL:` returns `NO`), you can provide links to App Store, encouraging users to download other applications from your company.

Registering Schemes: Declaring the URL

It takes two steps to add services to your application. First, declare your URL scheme in your Info.plist property list. Second, add a handler to your application delegate. Here are the steps you take to do this.

To declare your URL scheme, you need to specify information for the iOS's Launch Services. Add a `CFBundleURLTypes` entry into your Info.plist. This consists of an array of dictionaries that describe the URL types the application can open and handle. Each dictionary is quite simple. They contain two keys: a `CFBundleURLName` and an array of `CFBundleURLSchemes`.

The URL name is an abstract name (also known as its "kind"). You can use any string. On the Mac, this provides the visible description shown in Finder. On iOS, it's just a way of keeping your schemes straight. Figure 16-8 names its URL type `com.sadun.urlSchemeDemonstration`.

From the Library of Eric Berna



Figure 16-8 A custom URL scheme declaration. This scheme will open any URL prefixed by `xyz:`.

The Schemes array provides a list of prefixes that belong to the abstract name. You can add one scheme or many. The following declares just one. You may want to prefix your name with an x (e.g., `x-sadun-services`). Although the iOS family is not part of any standards organization, the x prefix indicates that this is an unregistered name. Recipe 16-6 uses the `xyz` scheme shown in Figure 16-8.

```
<key>CFBundleURLTypes</key>
<array>
    <dict>
        <key>CFBundleURLName</key>
        <string>com.sadun.urlSchemeDemonstration</string>
        <key>CFBundleURLSchemes</key>
        <array>
            <string>x-sadun-services</string>
        </array>
    </dict>
</array>
```

Former iOS developer (and current Apple employee) Emanuele Vulcano has started an informal registry over at the CocoaDev Web site (http://cocoadev.com/index.pl?ChooseYourOwniPhoneURLScheme). iOS developers can share their schemes in a central listing, so that you can discover services you want to use and promote services that you offer. The registry lists services, their URL schemes, and describes how these services can be used by other developers.

Registering Schemes: Adding the Handler Method

The second part of implementing URL handling is to provide any of a series of application delegate calls. This method chain has grown over time and is quite crusty in its complexity. The single method you'll probably want to define to handle URLs is demonstrated in Recipe 16-6. This provides the preferred entry point for iOS 5. To test this recipe, launch the app, suspend it, and open an `xyz:` URL in Mobile Safari.

Recipe 16-6 **Providing URL Scheme Support**

```
- (BOOL)application:(UIApplication *)application
    openURL:(NSURL *)url
    sourceApplication:(NSString *)sourceApplication
    annotation:(id)annotation
{
```

From the Library of Eric Berna

```
    tbvc.title = url.path;
    NSLog(@"URL: %@", url);
    NSLog(@"Opened from : %@", sourceApplication);
    NSLog(@"Annotation : %@", annotation);

    return YES;
}
```

> **Get This Recipe's Code**
>
> To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from this book, go to the folder for Chapter 16 and open the project for this recipe.

By preferred entry point, it does not mean only entry point. You can also grab items from the launch dictionary in `application:didFinishLaunchingWithOptions:`. A final, now deprecated approach is to implement `application:handleOpenURL:`, which remains as a valid entry point in iOS 5, but I strongly encourage you to avoid it.

Returning Control to a Calling Application

With careful programming, you can allow the calling application to regain control after handling a URL request. An application might send back requested material along with a status indication of whether the operation was a success. It depends on how you define and implement your protocol. By adding a reverse scheme element in the calling URL, you know how to call home once the requested service completes.

```
x-sadun-services:paste?scheme=iping&data=hello+world&\
clipboard=test1&password=foobar&expire=1500
```

Be aware that app-to-app URL launching has a fairly long latency. You can expect a return trip to last up to 5 seconds or more. What's more, should the calling client lie or provide the wrong scheme parameter, control may be transferred to a third application. If the scheme does not refer to a real application, the request will simply hang in the receiving app. On the other hand, allowing the service to return to the originating application keeps the user from having to quit and relaunch that first application and is a feature more applications should offer once they finish handling a request.

Recipe: Adding Custom Settings Bundles—Or Not

iOS uses the `NSUserDefaults` class to access and manage application preferences. With it, you can store information that your application needs to preserve between successive runs. For example, you might save a current player name, a list of high scores, or the last-used view configuration. User defaults programmatically assign values to a persistent

From the Library of Eric Berna

database associated with your application. These defaults are stored in your application sandbox's Library folder, in a property list file named with your application identifier.

Treat user defaults as a mutable dictionary. Set and retrieve objects using keys, just as you would with that dictionary. Defaults entries are limited to standard property list types—that is, `NSString`, `NSNumber`, `NSDate`, `NSData`, `NSArray`, and `NSDictionary`. When you need to store information that does not fall into one of these classes, consider using another file (such as one that resides in your sandbox's Library folder) or serialize your object into `NSData` and store that data in defaults.

The `synchronize` method forces the defaults database to update to the latest changes made in memory. Synchronizing assures you that the file-based defaults data is up to date, an important factor if your application gets interrupted for some reason. The following snippet demonstrates setting, synchronizing, and retrieving data from the user defaults system:

```
[[NSUserDefaults standardUserDefaults]
    setObject:colors forKey:@"colors"];
[[NSUserDefaults standardUserDefaults]
    setObject:locs forKey:@"locs"];

[[NSUserDefaults standardUserDefaults] synchronize];

NSLog(@"%@", [[NSUserDefaults] objectForKey@"lastViewTag"]);
```

The Settings App

iPhone applications can add custom preferences into the main Settings app (see Figure 16-9). These preferences access the same application-specific user defaults that you work with programmatically. Any changes your users make to these screens update and synchronize with standard user defaults. The difference is that Settings provides a "friendly" GUI for your users and that the chances of your users stumbling across these settings is, practically zero.

A word of caution: Avoid using Settings bundles with your apps. Build your preferences directly into your app or prepare for an onslaught of customer service requests because the discoverability of this feature is marginal. I include coverage of this topic out of a sense of courtesy. Although they are not easily discoverable for users, they are particularly helpful for kiosk-style apps. When working with apps intended for trade shows or other public use, using Settings bundles helps hide those defaults items you don't want to expose in-app.

Custom settings are listed after system settings, but otherwise look and act like the ones that Apple preloaded into your system. As the screenshots in Figure 16-9 show, custom preferences provide a variety of data interaction styles, including text fields, switches, and sliders.

From the Library of Eric Berna

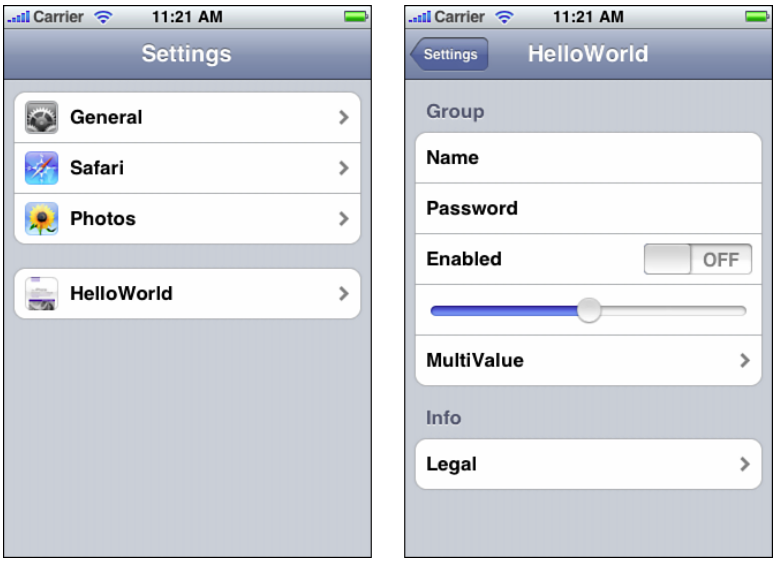


Figure 16-9 (Left) Custom settings bundles for third-party applications appear on the Settings screen after the built-in settings. On the iPhone, you may have to scroll down a bit to find them. (Right) Developer-defined preferences elements can include text fields (both regular and secure), switches, sliders, multivalue choices, group titles, and child panes.

Because these settings create standard `NSUserDefaults` entries, you can easily query and modify any of these settings from code. For example, Recipe 16-7 defines a field called "Name" (see Figure 16-9, right screenshot, first item in the top Group). This text field stores its value to the `@"name_preference"` key. You can see whether the user has entered a value for this key from your application.

```
NSLog(@"% objectForKey:@"name_preference"];
```

Avoid Sensitive Information

Use settings to store nonsensitive account preferences such as usernames and option toggles. Although passwords are visually obscured with dots, they're stored in clear text in your application sandbox. When working with sensitive information, always use your iPhone's secure keychain instead. Settings bundles do not offer keychain integration at this time. (Keychain recipes appear in Chapter 15.)

Settings Schema

A copy of the settings schema resides in your Developer folder at /Developer/Platforms/iPhoneOS.platform/Developer/Library/Xcode/Plug-ins/iPhoneSettingsPlistStructDefs.xcodeplugin. Xcode uses this file to check its property list syntax. In the file, you can see

From the Library of Eric Berna

all the definitions and the required and optional attributes used to specify custom preferences. If Apple should ever expand or change its definitions, you'll be able to find those changes in this file.

Defining a Settings Bundle

Each settings pane corresponds to one property list file. Recipe 16-7 shows the source for the pane in Figure 16-9 (right). It demonstrates each SDK settings type and provides a sample definition. Types include text fields (strings), sliders (floating-point numbers), switches (Boolean values), and multiple selection (one-of-n choices). In addition, you can group items and link to child panes.

To add new settings, build a dictionary and add it to the `PreferencesSpecifiers` array. Each individual preference dictionary needs, at a minimum, a type and a title. Some settings, like the `PSGroupSpecifier` group item, require nothing more to work. Others, such as text fields, use quite a few properties. You want to specify capitalization and autocorrection behaviors as well as the keyboard type and whether the password security feature obscures text, as you can see in this recipe.

To add a settings bundle to your program, follow these steps. Alternatively, you can create a new settings bundle by choosing File > New File > iPhone OS > Resource > Settings Bundle:

1. Create each of the property lists, one for each screen. The primary plist must be named Root.plist.
2. Create a new folder in the Finder and add your property lists to that folder.
3. Rename the folder to Settings.bundle. OS X warns you about the name; go ahead and confirm the rename. The folder transforms into a bundle. (To view the contents of your new bundle, right-click [Control-click] and choose Show Package Contents from the contextual pop up.)
4. Drag the bundle into the File Navigator of your Xcode project.
5. Optionally, create a 29x29 version of your main application icon file (typically icon.png) and add it to your project with the name Icon-Settings.png. This art is used by the Settings application to label your bundle along with the application name.

When you next run your program, the settings bundle installs and makes itself available to the Settings application. Should your source have any syntax errors, you find a blank screen rather than the settings you expect. It helps to build your settings in stages to avoid this.

Recipe 16-7 **Creating a Custom Settings Pane**

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
```

From the Library of Eric Berna

```
<dict>
    <key>Title</key>
    <string>YOUR_PROJECT_NAME</string>
    <key>StringsTable</key>
    <string>Root</string>
    <key>PreferenceSpecifiers</key>
    <array>
        <dict>
            <key>Type</key>
            <string>PSGroupSpecifier</string>
            <key>Title</key>
            <string>Group</string>
        </dict>
        <dict>
            <key>Type</key>
            <string>PSTextFieldSpecifier</string>
            <key>Title</key>
            <string>Name</string>
            <key>Key</key>
            <string>name_preference</string>
            <key>DefaultValue</key>
            <string></string>
            <key>IsSecure</key>
            <false/>
            <key>KeyboardType</key>
            <string>Alphabet</string>
            <key>AutocapitalizationType</key>
            <string>None</string>
            <key>AutocorrectionType</key>
            <string>No</string>
        </dict>
        <dict>
            <key>Type</key>
            <string>PSTextFieldSpecifier</string>
            <key>Title</key>
            <string>Password</string>
            <key>Key</key>
            <string>prefs_preference</string>
            <key>DefaultValue</key>
            <string></string>
            <key>IsSecure</key>
            <true/>
            <key>KeyboardType</key>
            <string>Alphabet</string>
            <key>AutocapitalizationType</key>
            <string>None</string>
```

From the Library of Eric Berna

```
    <key>AutocorrectionType</key>
    <string>No</string>
</dict>
<dict>
    <key>Type</key>
    <string>PSToggleSwitchSpecifier</string>
    <key>Title</key>
    <string>Enabled</string>
    <key>Key</key>
    <string>enabled_preference</string>
    <key>DefaultValue</key>
    <true/>
    <key>TrueValue</key>
    <string>YES</string>
    <key>FalseValue</key>
    <string>NO</string>
</dict>
<dict>
    <key>Type</key>
    <string>PSSliderSpecifier</string>
    <key>Key</key>
    <string>slider_preference</string>
    <key>DefaultValue</key>
    <real>0.5</real>
    <key>MinimumValue</key>
    <integer>0</integer>
    <key>MaximumValue</key>
    <integer>1</integer>
    <key>MinimumValueImage</key>
    <string></string>
    <key>MaximumValueImage</key>
    <string></string>
</dict>
<dict>
    <key>Type</key>
    <string>PSMultiValueSpecifier</string>
    <key>Key</key>
    <string>multi_preference</string>
    <key>DefaultValue</key>
    <string>One</string>
    <key>Title</key>
    <string>MultiValue</string>
    <key>Titles</key>
    <array>
        <string>one</string>
        <string>two</string>
```

From the Library of Eric Berna

```
                <string>three</string>
                <string>four</string>
            </array>
            <key>Values</key>
            <array>
                <string>one</string>
                <string>two</string>
                <string>three</string>
                <string>four</string>
            </array>
        </dict>

        <dict>
            <key>Type</key>
            <string>PSGroupSpecifier</string>
            <key>Title</key>
            <string>Info</string>
        </dict>
        <dict>
            <key>Type</key>
            <string>PSChildPaneSpecifier</string>
            <key>Title</key>
            <string>Legal</string>
            <key>File</key>
            <string>Legal</string>
        </dict>
    </array>
</dict>
</plist>
```

> **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 this book, go to the folder for Chapter 16 and open the project for this recipe.

Settings and Users

Although settings bundles offer a well-defined resource for developers to centralize their user-defined defaults, real-world experience suggests they're a feature you won't want to use. Few iOS users are aware of third-party settings outside their applications. Even fewer actually use those settings on a regular basis. Most users want to stay within the bounds of an application for all app-related tasks, including settings.

From the Library of Eric Berna

Because of this, many (if not most) App Store developers have moved away from settings bundles and brought their settings directly into the application. Adding settings views allows users to find and set preferences easily. Unfortunately, creating those screens is labor intensive and fussy.

There is, fortunately, a middle ground between relying solely on settings bundles and building your own views. The Llama Settings project at Google Code (http://code.google.com/p/llamasettings/) offers a set of classes that read property lists (including from your settings bundles), allowing you to display settings screens within your application without much extra work or overhead. The project was developed and is maintained by Scott Lawrence.

Open sourced, the Llama Settings classes provide similar kinds of display and interactive elements, including group titles, sliders, and switches. In addition, the project adds support for color selectors, URL launchers, and more. Although these items are not supported in Apple's Settings app, you can use them within your program by defining standard property lists without further programming.

Checking User Defaults

You may solicit and set defaults via settings bundles, application-based views, code-level access, or a hybrid of these approaches. When using those settings, be aware that certain items may not yet exist. If a user hasn't opened your settings bundle, the default settings you specified in the bundle's property lists may not have ever been set. For most objects, you can test for this via `objectForKey:`. This method returns `nil` for nonexistent keys.

Here's one reason a `nil` value may play a role in your programming: One default that you'll always want to set through code is a "last version" key. This key records the application version that was most recently run. You'll want to check for this default whenever your application launches.

If that default is `nil`, the application is a new install. You may want to prepare files and perform other setup tasks at first run. After that setup, set a value for the key, one that indicates the currently deployed app version. (And don't forget to `synchronize` after setting that key.)

It doesn't end with a check for `nil`, though. You'll always know when the user has just upgraded from a previous version by checking that setting. When the last run version differs from your current version, you have the opportunity to perform any updates that bring your user into data compliance for the most recent release.

Summary

Want to share data across applications? This chapter showed you how. You read about UTIs and how they are used to specify data roles across applications. You saw how the pasteboard worked, and how you could share files with iTunes. You read about monitoring folders and discovered how to implement custom URLs. You dived deep into the

From the Library of Eric Berna

document interaction controller and saw how to add support for everything from printing to copying to previews. Here are a few thoughts to take with you before leaving this chapter:

- You are never limited to the built-in UTIs that Apple provides, but you should follow its lead when you decide to add your own. Be sure to use custom reverse domain naming and add as many details as possible (public URL definition pages, typical icons, file extensions) in your exported definitions. Precision matters.
- Conformance arrays help you determine what kind of thing you're working with. Knowing that it's an image and not, say, a text file or movie, can help you better process the data associated with any file.
- The general pasteboard offers a terrific way to work with shared data, but if you have application-specific needs for cross communication, there's no reason you can't use a custom pasteboard to share information. Just be aware that pasteboard data will not persist across reboots.
- The Documents folder belongs to the user and not to you. Remember that and provide respectful management of that directory.
- The documents interaction controller supercedes a lot of the reasons many developers used to use custom URL schemes. Use the controller to provide the app-to-app interaction your users demand and don't be afraid of introducing annotation support to help ease transition between apps. Offering a "return to previous app" button using annotation details is a lovely way to support full-platform app integration.
- Don't offer an "Open In" menu option unless there's an "Open In" controller ready to back up that button. The solution you read about in this chapter is crude but it's better than dealing with angry, frustrated, or confused users through customer support.
- Settings bundles? Just say no.

From the Library of Eric Berna

17

Connecting to the Address Book

In addition to standard user interface controls and media components that you'd see on any computer, the iOS SDK provides a number of tightly focused developer solutions specific to iOS delivery. Among the most useful of these is the Address Book framework, allowing you to programmatically access and manage the contacts database. This chapter introduces the Address Book and demonstrates how to use its frameworks in your applications. You read about accessing information on a contact-by-contact basis, how to modify and update contact information, and how to use predicates to find just the contact you're interested in. This chapter also covers the GUI classes that provide interactive solutions for picking, viewing, and modifying contacts. By the time you've read this chapter, you'll have discovered the Address Book from the bottom up.

The AddressBook Frameworks

The iOS SDK provides not one, but two, AddressBook frameworks: AddressBook.framework and AddressBookUI.framework. As their names suggest, they occupy distinct niches in the iOS SDK. AddressBook provides low-level Core Foundation (C-based) objects and routines for accessing contact information from the iPhone's onboard databases. AddressBookUI offers high-level Objective-C–based `UIViewController` browser objects to present to users. Both frameworks are small. They provide just a few classes and data types.

AddressBookUI

The AddressBookUI framework provides several precooked view controllers that interact with the onboard contacts database. These interfaces include a general people picker, a contact viewer, and a contact editor. You set a delegate and then push these controllers onto your navigation stack or display them modally, as shown in the recipes in this chapter.

From the Library of Eric Berna

Like other special-purpose controllers discussed throughout this book, the AddressBookUI controllers are not very general or flexible. Apple intends you to use them as provided, with little or no customization from the developer. What's more, they require a certain degree of low-level programming prowess. As you see in this chapter, these classes interact with the underlying Address Book in circuitous ways.

AddressBook and Its Databases

The AddressBook framework offers centralized access to iOS's Contacts database. This database stores information about individual contacts, including names, addresses, phone numbers, IM details, and more. Built using a Core Foundation–style API, the AddressBook framework allows you to query, modify, and add contacts from your applications.

The iOS contact data resides in the user's home Library folder. On the Macintosh-based iPhone simulator, you can freely access these files in ~/Library/Application Support/iPhone Simulator/*iOS Version*//Library/AddressBook. The files you find there—specifically, AddressBook.sqlitedb and AddressBookImages.sqlitedb use standard SQLite to store contact information and, in the latter file, optional contact images.

On iOS, the same files live in /var/mobile/Library/AddressBook, that is, out of the application sandbox and therefore out of the reach of your direct inspection. Use the two AddressBook frameworks to query or modify the user's contact information rather than attempting to access these files through code.

Records

In the Core Foundation C-based AddressBook framework, the `ABRecordRef` type provides the core contact structure. This record stores all information for each contact, including name, e-mail, phone numbers, an optional image, and so forth. Every record corresponds to a complete Address Book contact. Create a new record by calling `ABPersonCreate()`:

```
ABRecordRef person = ABPersonCreate();
```

Each record is stored in a single central Address Book. You can access this data by creating a new Address Book instance. Despite the name, the `ABAdressbookCreate()` function does not create a new Address Book; it builds a local object that loads its data from the system Address Book. This function offers the primary entry-point for you to access Address Book data directly. What's more, until you create an Address Book or a record instance, many of the constants you need to use in your applications will be undefined. Apple has many warnings along these lines in its documentation: "The value of these constants is undefined until one of the following functions has been called: `ABAddressBookCreate()`, `ABPersonCreate()`, `ABGroupCreate()`."

From the Library of Eric Berna

The Custom ABStandin Class

Rather than create an Address Book instance each time data is accessed, the samples in this chapter use a custom `ABStandin` class. This class provides several basic Address Book access utilities and stores a static shared `ABAddressBookRef` instance. It is implemented as follows:

```
static ABAddressBookRef shared = NULL;

@implementation ABStandin

// Return the current shared address book,
// Creating if needed
+ (ABAddressBookRef) addressBook
{
    if (shared) return shared;

    shared = ABAddressBookCreate();
    return shared;
}

// Load the current address book
+ (ABAddressBookRef) currentAddressBook
{
    if (shared)
    {
        CFRelease(shared);
        shared = nil;
    }

    return [self addressBook];
}

// Save out the address book
+ (BOOL) save: (NSError **) error
{
    CFErrorRef cfError;
    if (shared)
    {
        BOOL success = ABAddressBookSave(shared, &cfError);
        if (!success)
        {
            if (error)
                *error = (__bridge_transfer NSError *)cfError;
```

From the Library of Eric Berna

```
            return NO;
        }
        return YES;
    }
    return NO;
}
@end
```

In earlier editions of this book, Address Book objects were created on the fly and autoreleased. With ARC, you can no longer make these CF objects autoreleasable. (Or, to be more precise, you can, but it involves a bit of extra-ARC compilation, and the static analyzer will give you a hard time about it.) The code in this chapter assumes that a single static instance is available at all times.

To help that, there's a second method here called `currentAddressBook`. This method releases the current instance and creates a new one. You only need to do this in the circumstance that the Address Book has been altered since you made changes and you need to refresh its data before applying new updates. There are several approaches you can take to this problem.

First, you can refresh the data each time you plan to read from or write to the Address Book. Replace any calls in this chapter's samples from `addressBook` to `currentAddress book`. You'll always be guaranteed clean data.

Or, second, you can refresh the data each time the application becomes active to ensure that any edits made when the application was suspended are reflected in the current data. That's the approach I generally use. It all comes down to how paranoid you are (and paranoia is not a bad thing in this case) and how much overhead you're willing to deal with.

A final method called `save:` updates the persistent Address Book data store with any updates you have made to the shared in-memory Address Book instance.

Querying the Address Book

You can ask the Address Book for the number of objects currently stored in its database using a single function call. Although the name refers to Persons, this method counts all entries, including businesses, and returns a simple count.

```
+ (int) contactsCount
{
    ABAddressBookRef addressBook = [ABStandin addressBook];
    return ABAddressBookGetPersonCount(addressBook);
}
```

Wrapping the AddressBook Framework

The AddressBook framework can be fussy and difficult to work with if you're more comfortable using Objective-C. To deal with this, this chapter introduces a simple wrapper

From the Library of Eric Berna

called ABContact. This wrapper class hides the CF nature of the class and replaces it with an Objective-C interface. Objective-C wrappers simplify integration between the C-based Address Book calls and normal Cocoa Touch development and memory management. The full source for this, and other associated wrapper classes, can be found in the sample code for this chapter.

The ABContact class hides an internal ABRecordRef, the CF type that corresponds to each contact record. The remaining portion of the wrapper involves nothing more than generating properties and methods that allow you to reach into the ABRecordRef to set and access its subrecords.

```
@interface ABContact : NSObject
{
      ABRecordRef record;
}
@end
```

A second helper class ABContactsHelper wraps the AddressBook framework at the global level. It allows you to modify and query the database as a whole. Take its contacts method, for example. As with the record count, the Address Book can also return an array of contacts. You recover individual records by calling the ABAddressBookCopyArrayOfAllPeople() function, which returns a CF array of ABRecordRef objects.

The following ABContactsHelper method wraps this function to retrieve a record array. It then adds each contact record to a new ABContact wrapper and returns a standard NSArray populated with Objective-C contact instances.

```
+ (NSArray *) contacts
{
    ABAddressBookRef addressBook = [ABStandin addressBook];
    NSArray *thePeople =
        (__bridge_transfer NSArray *)
            ABAddressBookCopyArrayOfAllPeople(addressBook);
    NSMutableArray *array = [NSMutableArray array];
    for (id person in thePeople)
        [array addObject:[ABContact
            contactWithRecord:(__bridge ABRecordRef)person]];
    return array;
}
```

Working with Record Functions

The majority of ABRecordRef functions use the ABPerson prefix. This prefix corresponds to the ABPerson class that is available on the Macintosh, but not on iOS. So, while the function calls are ABPerson-centric, all the data affected by these calls are actually ABRecordRef instances.

The reason for this becomes clearer when you notice that the same ABRecordRef structure is used in the AddressBook framework to represent both people (individual

From the Library of Eric Berna

contacts, whether people or businesses) and groups (collections of contacts, such as work colleagues and personal friends). The SDK provides `ABGroup` functions as well as `ABPerson` ones, allowing you to add members and access membership lists. You read more about groups later in this section.

You already encountered the `ABPersonCreate()` function earlier in this section. As its prefix indicates, it's used to create a contact record. The SDK offers almost two dozen other `ABPerson`-named functions, as well as a handful each of `ABGroup` and `ABRecord` functions. Together, these functions allow you to set and retrieve values within a contact, compare records, test for conforming data, and more.

Retrieving and Setting Strings

Each `ABRecord` stores a about a dozen simple string values that represent, among other items, a person's name, title, job, and organization. You retrieve single-valued strings by copying field values from the record. By "single-valued," I mean that a person can own just one of each of these items. A person has only one first name, one birthday, and one nickname—at least as far as the AddressBook framework is concerned. Contrast this with phone numbers, e-mail addresses, and social-network identities. These latter are called "multivalued" and refer to items that naturally occur in multiples.

Recovering Strings

As you'd expect, working with single-valued elements is simpler than multivalued ones. You have just one item to set and to retrieve. All single-valued elements can be accessed using the same `ABRecordCopyValue()` function. The `ABContact` wrapper specializes this access based on the data type returned, such as for strings or dates. The following method copies a record value, casts it to a string, and returns that content:

```
- (NSString *) getRecordString:(ABPropertyID) anID
{
    NSString *result =
        (__bridge_transfer NSString *)
            ABRecordCopyValue(record, anID);
    return result;
}
```

To determine which value to copy, it requires a property constant (`ABPropertyID`) that identifies the requested field in the record. The sample calls that follow use identifiers for the first and last name fields:

```
// Sample uses
- (NSString *) firstname
    {return [self getRecordString:kABPersonFirstNameProperty];}
- (NSString *) lastname
    {return [self getRecordString:kABPersonLastNameProperty];}
```

From the Library of Eric Berna

String Fields

The string-based fields you can recover in this fashion are as follows; their names are fairly self-explanatory. These identifiers are declared as constant integers in the `ABPerson.h` header file. Each item identifies a single string-based field in an `ABRecordRef` record:

- `kABPersonFirstNameProperty`
- `kABPersonLastNameProperty`
- `kABPersonMiddleNameProperty`
- `kABPersonPrefixProperty`
- `kABPersonSuffixProperty`
- `kABPersonNicknameProperty`
- `kABPersonFirstNamePhoneticProperty`
- `kABPersonLastNamePhoneticProperty`
- `kABPersonMiddleNamePhoneticProperty`
- `kABPersonOrganizationProperty`
- `kABPersonJobTitleProperty`
- `kABPersonDepartmentProperty`
- `kABPersonNoteProperty`

Setting String Properties

Setting string-based properties proves to be just as simple as retrieving them. Cast the string you want to set to a `CFStringRef`. You need to use Core Foundation objects with Address Book calls. Then, use `ABRecordSetValue()` to store the data into the record.

Note that these calls do not update the Address Book. They only change the data within the record. If you want to store a user's contact information, you have to write that information back to the Address Book. Use the `save:` method shown earlier in this section.

```
- (BOOL) setString: (NSString *) aString
    forProperty:(ABPropertyID) anID
{
    CFErrorRef cfError = NULL;
    BOOL success = ABRecordSetValue(record, anID,
        (__bridge CFStringRef) aString, &cfError);
    if (!success)
    {
        NSError *error = (__bridge_transfer NSError *) cfError;
        NSLog(@"Error: %@", error.localizedFailureReason);
    }
    return success;
}
```

From the Library of Eric Berna

```
// Examples of use
- (void) setFirstname: (NSString *) aString
    {[self setString: aString forProperty:
        kABPersonFirstNameProperty];}

- (void) setLastname: (NSString *) aString
    {[self setString: aString
        forProperty: kABPersonLastNameProperty];}
```

Working with Date Properties

In addition to the string properties you just saw, the Address Book stores three key dates: an optional birthday, the date the record was created, and the date the record was last modified. These items use the following property constants:

- `kABPersonBirthdayProperty`
- `kABPersonCreationDateProperty`
- `kABPersonModificationDateProperty`

Access these items exactly as you would with strings, but cast to and from `NSDate` instances instead of `NSString` instances. Don't try to modify the latter two properties; allow the Address Book to handle them for you.

```
// Return a date-time field from a record
- (NSDate *) getRecordDate:(ABPropertyID) anID
{
    return (__bridge_transfer NSDate *)
        ABRecordCopyValue(record, anID);
}

// Get the contact's birthday
- (NSDate *) birthday
    {return [self getRecordDate:kABPersonBirthdayProperty];}

// Set a date-time field in a record
- (BOOL) setDate: (NSDate *) aDate forProperty:(ABPropertyID) anID
{
    CFErrorRef cfError = NULL;
    BOOL success = ABRecordSetValue(record, anID,
        (__bridge CFDateRef) aDate, &cfError);
    if (!success)
    {
        NSError *error = (__bridge_transfer NSError *) cfError;
        NSLog(@"Error: %@", error.localizedFailureReason);
    }
    return success;
}
```

From the Library of Eric Berna

```
// Set the contact's birthday
- (void) setBirthday: (NSDate *) aDate
    {[self setDate: aDate forProperty: kABPersonBirthdayProperty];}
```

Multivalue Record Properties

Each person may have multiple e-mail addresses, phone numbers, and important dates (beyond the birthday singleton) associated with his or her contact. `ABPerson` uses a multivalue structure to store lists of these items.

Each multivalue item is actually an array of pairs. `ABContact` represents these pairs as a two-item dictionary. The *value* refers to the actual item that's being added to the Address Book. The *label* specifies the role of that item. For example, a person may have a work phone, a home phone, and a mobile phone. Each of these roles (work, home, and mobile) are labels. The actual phone numbers are the values associated with the roles.

Here's how you can create the `ABContact` dictionaries that populate each of these arrays. You pass a value and label, building up a dictionary from those elements. These elements are used when calling `ABMultiValueAddValueAndLabel()`.

```
+ (NSDictionary *) dictionaryWithValue: (id) value
    andLabel: (CFStringRef) label
{
    NSMutableDictionary *dict = [NSMutableDictionary dictionary];
    if (value) [dict setObject:value forKey:@"value"];
    if (label) [dict setObject:
        (__bridge NSString *)label forKey:@"label"];
    return dict;
}
```

Multivalue Labels

The AddressBook framework uses three generic labels, which can be used for any data class. These are `kABWorkLabel`, `kABHomeLabel`, and `kABOtherLabel`. In addition, you'll find type-specific labels for URLs, Social Media, Instant Messaging, Phone Numbers, Dates, and Relationships. Each of these items is documented in the AddressBook documentation and may change over time. The Social Media elements (Twitter, Facebook, LinkedIn, and so on) represent the latest additions to the iOS 5 SDK.

Retrieving Multivalue Arrays

Recover each array from the record via its property identifier, returning a `CFArrayRef`, which you can use directly or bridge to an `NSArray`. The multivalue property identifiers you work with are as follows:

- `kABPersonEmailProperty`
- `kABPersonPhoneProperty`
- `kABPersonURLProperty`

From the Library of Eric Berna

- kABPersonDateProperty
- kABPersonAddressProperty
- kABPersonSocialProfileProperty
- kABPersonInstantMessageProperty

Each multivalue type plays an important role in storing data back to the record. The type is used to allocate memory and determine the size for each field within the record.

The first three of these items (e-mail, phone, and URL) store multistrings—that is, arrays of strings. Their associated type is the kABMultiStringPropertyType.

The next item, the date property, stores an array of dates using kABMultiDateTimePropertyType.

The last three items, namely the address, social profile, and instant-message properties consist of arrays of dictionaries and use kABMultiDictionaryPropertyType. Each of these dictionaries has its own complexities and custom keys. (They are discussed later in this section.)

> **Note**
>
> "Related names" represents another multistring property, but one that does not actually get used on the iOS Contacts application at this time. It uses the kABPersonRelatedNamesProperty constant and stores names and their relationships (for example, Mary Ball—Washington might be stored in George Washington's contact using the kABPersonMotherLabel). See ABPerson.h for a full list of relation constants.

Retrieving Multivalue Property Values

To retrieve array of values for any of these properties, copy the property out of the record (using ABRecordCopyValue()) and then break it down into its component array. The Address Book provides a function that copies the array from the property into a standard CFArrayRef.

```
- (NSArray *) arrayForProperty: (ABPropertyID) anID
{
    CFTypeRef theProperty =
        ABRecordCopyValue(record, anID);
    if (!theProperty) return nil;
    NSArray *items = (__bridge_transfer NSArray *)
        ABMultiValueCopyArrayOfAllValues(theProperty);
    CFRelease(theProperty);
    return items;
}
```

Although you might think you've retrieved all the information with those two calls, you have not. Value retrieval alone is not sufficient for working with multivalued items. As mentioned earlier, each dictionary stored in a multivalue array uses a label and a value.

From the Library of Eric Berna

Figure 17-1 shows part of an Address Book contact page. Grouped items use labels to differentiate the role for each e-mail, phone number, and so forth. This contact has three phone numbers and three e-mail address, each of which displays a label indicating the value's role.

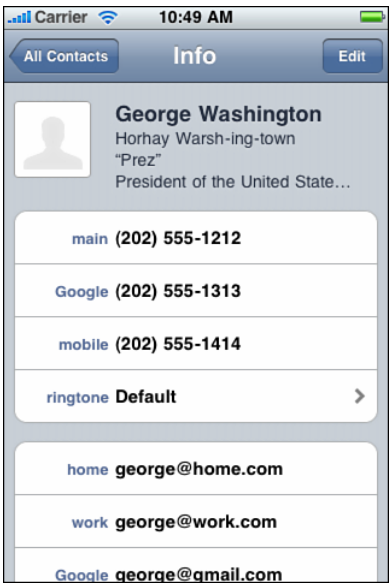


Figure 17-1 Multivalue items consist of both a label (e.g., main, Google, and mobile for these phone numbers, or home, work, and Google for these e-mail addresses) and a value.

Retrieving Multivalue Labels

You must transfer the labels from the property as well as the values to retrieve all the information stored for each multivalue property. The following method copies each label by index and adds it to a labels array. Together, the labels and the values comprise a complete multivalue collection.

```
- (NSArray *) labelsForProperty: (ABPropertyID) anID
{
    CFTypeRef theProperty = ABRecordCopyValue(record, anID);
    if (!theProperty) return nil;
    NSMutableArray *labels = [NSMutableArray array];
    for (int i = 0;
        i < ABMultiValueGetCount(theProperty); i++)
    {
```

From the Library of Eric Berna

```
        NSString *label = (__bridge_transfer NSString *)
            ABMultiValueCopyLabelAtIndex(theProperty, i);
        [labels addObject:label];
    }
    CFRelease(theProperty);
    return labels;
}
```

Storing Multivalue Data

Saving into multivalue objects works the same way as the retrieval functions you just read about, but in reverse. To store multivalued items into a record, you must transform your Cocoa Touch objects into a form the record can work with, namely the two-item value-label pair.

Custom Dictionaries

Although the value and label items are passed to the AddressBook functions as separate parameters, it's convenient to store them together in a single dictionary. This makes it easier to retrieve them in tandem when they are later needed. The following method tests a dictionary for conformance:

```
+ (BOOL) isMultivalueDictionary: (NSDictionary *) dictionary
{
    if (dictionary.allKeys.count != 2)
        return NO;
    if (![dictionary objectForKey:@"value"])
        return NO;
    if (![dictionary objectForKey:@"label"])
        return NO;

    return YES;
}
```

The following method expects an array of these custom dictionaries, whose objects correspond to the value and label retrieved from the original multivalued property. This code iterates through that array of dictionaries and adds each value and label to the mutable multivalue object, an instance of `ABMutableMultiValueRef`, which is created by calling `ABMultiValueCreateMutable()` with a property type.

```
- (ABMutableMultiValueRef) copyMultiValueFromArray:
    (NSArray *) anArray withType: (ABPropertyType) aType
{
    ABMutableMultiValueRef multi =
        ABMultiValueCreateMutable(aType);
    for (NSDictionary *dict in anArray)
    {
        if (![ABContact isMultivalueDictionary:dict])
```

From the Library of Eric Berna

```
            continue;
        ABMultiValueAddValueAndLabel(
            multi,
            (__bridge CFTypeRef) [dict objectForKey:@"value"],
            (__bridge CFTypeRef) [dict objectForKey:@"label"],
            NULL);
    }
    return multi;
}
```

The `ABPropertyType` supplied as the method parameter specifies what kind of values is being stored. For example, you can populate an e-mail property with strings and labels using `kABMultiStringPropertyType` or an Address Book using `kABMultiDictionaryPropertyType`.

Working with Multivalue Items

After you create a multivalue item using a method like the one just shown, you can then add it to a contact by setting a record value. The following method assigns a multivalue item to a property ID using the standard `ABRecordSetValue()` call. There is essentially no difference between this call and the calls that set a single date or string property. All the work is done in creating the multivalue item in the first place, not in the actual assignment.

```
- (BOOL) setMultiValue: (ABMutableMultiValueRef) multi
    forProperty: (ABPropertyID) anID
{
    CFErrorRef cfError = NULL;
    BOOL success = ABRecordSetValue(record, anID, multi, &cfError);
    if (!success)
    {
        NSError *error = (__bridge_transfer NSError *) cfError;
        NSLog(@"Error: %@", error.localizedFailureReason);
    }
    return success;
}
```

You might call this general assignment method from a specialized setter, such as the one shown here. This method assigns an array of email dictionaries by first building up a multivalued item and then setting the value.

```
- (void) setEmailDictionaries: (NSArray *) dictionaries
{
    // kABWorkLabel, kABHomeLabel, kABOtherLabel
    ABMutableMultiValueRef multi =
        [self copyMultiValueFromArray:dictionaries
            withType:kABMultiStringPropertyType];
    [self setMultiValue:multi
```

From the Library of Eric Berna

```
        forProperty:kABPersonEmailProperty];
    CFRelease(multi);
}
```

Addresses, Social Profile, and Instant-Message Properties

Three properties (addresses, social profiles, and instant message IDs) use multivalued dictionaries rather than strings or dates. This adds an extra step to the creation of a multivalue array. You must populate a set of dictionaries and then add them to an array along with their labels before building the multivalued item and adding it to a contact record. Figure 17-2 illustrates this additional layer. As the figure shows, e-mail multivalue items consist of an array of label-value pairs, where each value is a single string. In contrast, addresses use a separate dictionary, which is bundled into the value item.

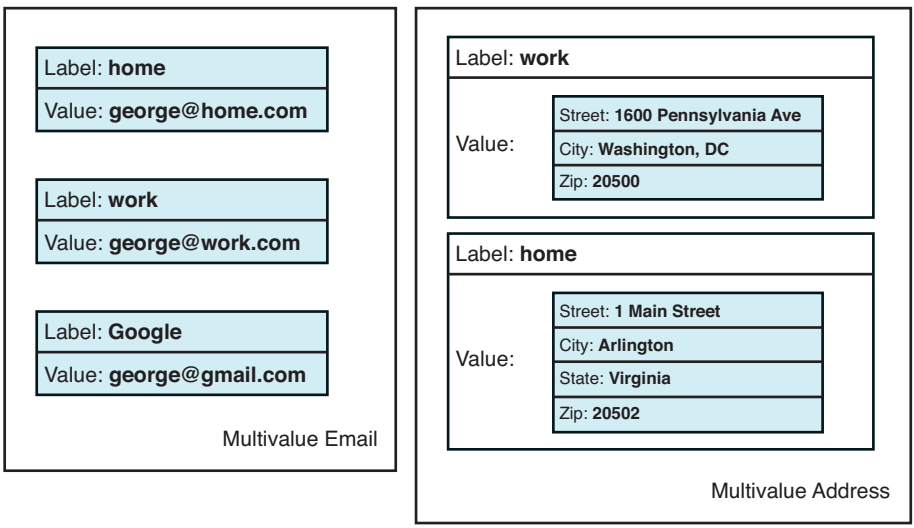


Figure 17-2 Unlike multivalue e-mail, which stores a single string for each value, multivalue addresses contain an entire address dictionary.

Here's an example that demonstrates the steps involved in creating a two-address multivalue item. This code builds the dictionaries and then adds them, along with their labels, to a base array. The array created by this code corresponds to the multivalue address object shown on the right side of Figure 17-2.

```
// Create the array that will store all the address
// value-label dictionaries
NSMutableArray *addresses = [NSMutableArray array];
```

From the Library of Eric Berna

```
// Create White House Address and add it to the array
NSDictionary *wh_addy = [ABContact
    addressWithStreet:@"1600 Pennsylvania Avenue"
    withCity:@"Washington, DC" withState:nil
    withZip:@"20500" withCountry:nil withCode:nil];
[addresses addObject:[ABContact dictionaryWithValue:wh_addy
    andLabel:kABWorkLabel]];

// Create a home address and add it to the array
NSDictionary *home_addy = [ABContact
    addressWithStreet:@"1 Main Street" withCity:@"Arlington"
    withState:@"Virginia" withZip:@"20502"
    withCountry:nil withCode:nil];
[addresses addObject:[ABContact dictionaryWithValue:home_addy
    andLabel:kABHomeLabel]];
```

This code relies on convenience methods to create both the address dictionaries and the value/label dictionaries used for the multivalue array. The following methods produce the label/value dictionaries for these three types. Notice how the keys for each dictionary is predefined, using Address Book key constants.

```
+ (NSDictionary *) addressWithStreet: (NSString *) street
    withCity: (NSString *) city
    withState:(NSString *) state
    withZip: (NSString *) zip
    withCountry: (NSString *) country
    withCode: (NSString *) code
{
    NSMutableDictionary *md = [NSMutableDictionary dictionary];
    if (street) [md setObject:street forKey:
        (__bridge NSString *) kABPersonAddressStreetKey];
    if (city) [md setObject:city forKey:
        (__bridge NSString *) kABPersonAddressCityKey];
    if (state) [md setObject:state forKey:
        (__bridge NSString *) kABPersonAddressStateKey];
    if (zip) [md setObject:zip forKey:
        (__bridge NSString *) kABPersonAddressZIPKey];
    if (country) [md setObject:country forKey:
        (__bridge NSString *) kABPersonAddressCountryKey];
    if (code) [md setObject:code forKey:
        (__bridge NSString *) kABPersonAddressCountryCodeKey];
    return md;
}

+ (NSDictionary *) socialWithURL: (NSString *) url
    withService: (NSString *) serviceName
```

From the Library of Eric Berna

```
    withUsername: (NSString *) username
    withIdentifier: (NSString *) key
{
    NSMutableDictionary *md = [NSMutableDictionary dictionary];
    if (url) [md setObject:url forKey:
        (__bridge NSString *) kABPersonSocialProfileURLKey];
    if (serviceName) [md setObject:serviceName forKey:
        (__bridge NSString *) kABPersonSocialProfileServiceKey];
    if (username) [md setObject:username forKey:
        (__bridge NSString *) kABPersonSocialProfileUsernameKey];
    if (key) [md setObject:key forKey:
        (__bridge NSString *)
            kABPersonSocialProfileUserIdentifierKey];
    return md;
}

+ (NSDictionary *) imWithService: (CFStringRef) service
    andUser: (NSString *) userName
{
    NSMutableDictionary *im = [NSMutableDictionary dictionary];
    if (service) [im
        setObject:(__bridge NSString *) service
        forKey:(__bridge NSString *)
            kABPersonInstantMessageServiceKey];
    if (userName) [im setObject:userName
        forKey:(__bridge NSString *)
            kABPersonInstantMessageUsernameKey];
    return im;
}
```

Address Book Images

Each record in the Address Book may be associated with an optional image. You can copy image data to and from each record. The `ABPersonHasImageData()` function indicates whether data is available for a given record. Use this to test whether you can retrieve image data.

Image data is stored as Core Foundation data, which can be bridged to `NSData`. As the `UIImage` class fully supports converting images into data and creating images from data, you just need to cast that data as needed. Use the `UIImagePNGRepresentation()` function to transform a `UIImage` instance into an `NSData` representation. To reverse, use `imageWithData:` to create a new image from `NSData`.

```
// Retrieve image
- (UIImage *) image
{
```

From the Library of Eric Berna

```
    if (!ABPersonHasImageData(record)) return nil;
    CFDataRef imageData = ABPersonCopyImageData(record);
    if (!imageData) return nil;

    NSData *data = (__bridge_transfer NSData *)imageData;
    UIImage *image = [UIImage imageWithData:data];
    return image;
}

// Set image
- (void) setImage: (UIImage *) image
{
    CFErrorRef cfError = NULL;
    BOOL success;

    if (image == nil) // remove
    {
        if (!ABPersonHasImageData(record)) return;
        success = ABPersonRemoveImageData(record, &cfError);
        if (!success)
        {
            NSError *error = (__bridge_transfer NSError *) cfError;
            NSLog(@"Error: %@", error.localizedFailureReason);
        }
        return;
    }

    NSData *data = UIImagePNGRepresentation(image);
    success = ABPersonSetImageData(
        record, (__bridge CFDataRef) data, &cfError);
    if (!success)
    {
        NSError *error = (__bridge_transfer NSError *) cfError;
        NSLog(@"Error: %@", error.localizedFailureReason);
    }
    return;
}
```

Creating, Adding, and Deleting Records

The `ABPersonCreate()` function returns a new `ABRecordRef` instance. This record exists outside the Address Book and represents a freestanding data structure. To date, all the examples in this chapter have modified individual records, but none so far has actually saved a record to the Address Book. Keep that in mind as you look at this convenience method that returns a newly initialized contact.

From the Library of Eric Berna

```
+ (id) contact
{
    ABRecordRef person = ABPersonCreate();
    id contact = [ABContact contactWithRecord:person];
    CFRelease(person);
    return contact;
}
```

Writing to the Address Book

To write new information to the Address Book takes two steps. First, add the record and then save the Address Book. New iOS developers often forget the second step, leading to an Address Book that appears to resist changes. This method adds a new contact to the Address Book but does not save the changes. Remember to call the custom AddressBook class's `save:` method, shown earlier, after adding the contact to make the new contact stick.

```
+ (BOOL) addContact: (ABContact *) aContact
    withError: (NSError **) error
{
    ABAddressBookRef addressBook = [ABStandin addressBook];
    BOOL success;
    CFErrorRef cfError = NULL;

    success = ABAddressBookAddRecord(
        addressBook, aContact.record, &cfError);
    if (!success)
    {
        if (error)
            *error = (__bridge_transfer NSError *)cfError;
        return NO;
    }

    return YES;
}
```

Removing Records

Removing a record from the Address Book requires a save step, just like adding a record. Once removed, the record still exists as an object, but it no longer is stored in the Address Book database. Here's how you can remove a record from the Address Book. Again, call `[ABStandin save:&error]` after.

```
- (BOOL) removeSelfFromAddressBook: (NSError **) error
{
    CFErrorRef cfError = NULL;
    BOOL success;
```

From the Library of Eric Berna

```
    ABAddressBookRef addressBook = [ABStandin addressBook];

    success = ABAddressBookRemoveRecord(
        addressBook, self.record, &cfError);
    if (!success)
    {
        if (error)
            *error = (__bridge_transfer NSError *)cfError;
        return NO;
    }

    return success;
}
```

Searching for Contacts

The default Address Book framework allows you to perform a prefix search across records. This function returns an array of records whose composite names (typically first name appended by last name, but localizable to countries where that pattern is reversed) match the supplied string.

```
NSArray *array = (__bridge_transfer NSArray *)
    ABAddressBookCopyPeopleWithName(addressBook, CFSTR("Eri"));
```

It's easier to use predicates and properties to perform searches. Use the custom `ABContact` class properties with `NSPredicate` instances to quickly filter for desired matches. The following code matches a string against a contact's first name, middle name, last name, and nickname. The predicate uses property names to define how it matches or rejects contacts. It does this using a case and diacritical insensitive match (`[cd]`) that compares against all points within each string (`contains`), not just the start (`begins with`).

```
+ (NSArray *) contactsMatchingName: (NSString *) fname
{
    NSPredicate *pred;
    NSArray *contacts = [ABContactsHelper contacts];
    pred = [NSPredicate predicateWithFormat:
        @"firstname contains[cd] %@ OR lastname contains[cd] %@ OR\
        nickname contains[cd] %@ OR middlename contains[cd] %@",
        fname, fname, fname, fname];
    return [contacts filteredArrayUsingPredicate:pred];
}
```

> **Note**
>
> Apple's Predicate Programming Guide offers a comprehensive introduction to predicate basics. It demonstrates how to create predicates and use them in your application.

From the Library of Eric Berna

Sorting Contacts

At times, you may want to sort your contacts in some way. The following methods mimic string sorting, but do so using the `ABContact` class's composite name property that combines the first, middle, and last names. You can easily adapt this approach to sort using other keys.

```
- (BOOL) isEqualToString: (ABContact *) aContact
{
    return [self.compositeName isEqualToString:aContact.compositeName];
}

- (NSComparisonResult) caseInsensitiveCompare: (ABContact *) aContact
{
    return [self.compositeName
        caseInsensitiveCompare:aContact.compositeName];
}
```

Working with Groups

Groups collect contacts into related sets, such as friends, colleagues, drinking buddies, and other natural groupings. Each group is nothing more than another `ABRecord`, but with a few special properties. Groups don't store names, addresses, and phone numbers. Instead, they store a reference to other contact records.

Counting Groups

You can count the number of groups in the current Address Book by retrieving the group records, as shown in this method. There's no direct way to query the number of groups as there is with the number of person contacts. The following methods are part of an `ABGroup` wrapper class that provides an Objective-C wrapper for Address Book groups like `ABContact` wraps Address Book contacts.

```
+ (int) numberOfGroups
{
    ABAddressBookRef addressBook = [ABStandin addressBook];
    NSArray *groups = (__bridge_transfer NSArray *)
        ABAddressBookCopyArrayOfAllGroups(addressBook);
    int ncount = groups.count;
    return ncount;
}
```

Creating Groups

Create groups using the `ABGroupCreate()` function. This function returns an `ABRecordRef` in the same way that `ABPersonCreate()` does. The difference lies in the record type. For groups, this property is set to `kABGroupType` instead of `kABPersonType`.

From the Library of Eric Berna

```
+ (id) group
{
    ABRecordRef grouprec = ABGroupCreate();
    id group = [ABGroup groupWithRecord:grouprec];
    CFRelease(grouprec);
    return group;
}
```

Adding and Removing Members

Add and remove members of a group by calling `ABGroupAddMember()` and `ABGroupRemoveMember()`. These calls affect records only and, as always, are not applied globally until you save the Address Book.

```
- (BOOL) addMember: (ABContact *) contact
    withError: (NSError **) error
{
    CFErrorRef cfError = NULL;
    BOOL success;

    success = ABGroupAddMember(self.record,
        contact.record, &cfError);
    if (!success)
    {
        if (error)
            *error = (__bridge_transfer NSError *)cfError;
        return NO;
    }

    return YES;
}

- (BOOL) removeMember: (ABContact *) contact
    withError: (NSError **) error
{
    CFErrorRef cfError = NULL;
    BOOL success;

    success = ABGroupRemoveMember(self.record,
        contact.record, &cfError);
    if (!success)
    {
        if (error)
            *error = (__bridge_transfer NSError *)cfError;
        return NO;
    }

    return YES;
}
```

From the Library of Eric Berna

Listing Members

Each member of a group is a person, or using this chapter's `ABContact` terminology, a contact. This method scans through a group's members and returns an array of `ABContact` instances, each initialized with the `ABRecordRef` for a group member.

```
- (NSArray *) members
{
    NSArray *contacts = (__bridge_transfer NSArray *)
           ABGroupCopyArrayOfAllMembers(self.record);
    NSMutableArray *array =
        [NSMutableArray arrayWithCapacity:contacts.count];
    for (id contact in contacts)
        [array addObject:[ABContact contactWithRecord:
            (__bridge ABRecordRef)contact]];
    return array;
}
```

Group Names

Every group has a name. It is the primary group property that you can set and retrieve. It uses the `kABGroupNameProperty` identifier, and it otherwise works just like the contacts properties.

```
- (NSString *) getRecordString:(ABPropertyID) anID
{
    return (__bridge_transfer NSString *) ABRecordCopyValue(record, anID);
}

- (NSString *) name
{
    return [self getRecordString:kABGroupNameProperty];
}

- (void) setName: (NSString *) aString
{
    CFErrorRef cfError = NULL;
    BOOL success;

    success = ABRecordSetValue(record, kABGroupNameProperty,
        (__bridge CFStringRef) aString, &cfError);
    if (!success)
    {
        NSError *error = (__bridge_transfer NSError *) cfError;
        NSLog(@"Error: %@", error.localizedFailureReason);
    }
}
```

From the Library of Eric Berna

ABContact, ABGroup, and ABContactsHelper

The sample code that accompanies this section includes source for three wrapper classes that have been demonstrated through these sections. The snippets shown throughout this discussion highlight the techniques used in these classes. Because of the length and overall redundancy of the classes, a single recipe listing has been omitted from this section. You can find the complete classes in the sample code for this chapter.

The custom `ABContact` class is somewhat based on the Mac-only `ABPerson` class. It provides a more Cocoa Touch interface with built-in Objective-C 2.0 properties than the C-style property queries that Apple's `ABPerson` uses. All the contact-specific methods you've seen to date in this section derive from this class.

A second class called `ABGroup` wraps all the group functionality for `ABRecordRef` instances. It offers Objective-C access to group creation and management. Use this class to build new classes, and add and remove members.

The final class, `ABContactsHelper` provides Address Book–specific methods. Use this class to search through the Address Book, retrieve arrays of records, and so forth. Although I included a few basic searches across names and phone numbers, you can easily expand this class, which is hosted at http://github.com/erica, for more complex queries.

Recipe: Searching the Address Book

Predicate-based searches are both fast and effective. Recipe 17-1 is built on `ABContact`'s predicate queries. It presents a search table that displays a scrolling table of contacts. This table responds to user search bar queries with live listing updates.

When a user taps a row, this recipe displays an `ABPersonViewController` instance. This class offers a contact overview, displaying the details for a given record, similar to the view shown in Figure 17-1. To use this view controller, you allocate and initialize it and set its `displayedPerson` property. As you'd expect, this property stores an `ABRecordRef`, which can easily be retrieved from an `ABContact` instance.

Person view controllers offer a limited delegate. By setting the `personViewDelegate` property, you can subscribe to the `personViewController:shouldPerformDefault-ActionForPerson:` method. This method triggers when users select certain items in the view, including phone numbers, e-mail addresses, URLs, and addresses. Return `YES` to perform the default action (dialing, e-mailing, etc.) or `NO` to skip.

Recipe 17-1 uses this callback to display the value for the selected item in the debug console. It's a convenient way to determine what the user touched and where his or her point of interest lies.

Although you can interact with other display elements, like the contact note and ringtone, these items do not produce callbacks.

To extend this recipe to allow editing, set the person view controller's `allowsEditing` property to `YES`. This provides the edit button that appears at the top right of the display. When tapped, the edit button triggers the same editing features in the person view that you normally see in the Contacts application.

From the Library of Eric Berna

Recipe 17-1 **Selecting and Displaying Contacts with Search**

```
// Return the number of table sections
- (NSInteger)numberOfSectionsInTableView:(UITableView *)aTableView
{
    // One section for this example
    return 1;
}

// Return the number of rows per section
- (NSInteger)tableView:(UITableView *)aTableView
    numberOfRowsInSection:(NSInteger)section
{
    if (aTableView == self.tableView)
        return matches.count;

    // On cancel, matches are restored in the search bar delegate
    matches = [ABContactsHelper
        contactsMatchingName:searchBar.text];
    matches = [matches sortedArrayUsingSelector:
        @selector(caseInsensitiveCompare:)];
    return matches.count;


    matches = [matches sortedArrayUsingSelector:
        @selector(caseInsensitiveCompare:)];
    return matches.count;
}

// Produce a cell for the given index path
- (UITableViewCell *)tableView:(UITableView *)aTableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Dequeue or create a cell
    UITableViewCellStyle style = UITableViewCellStyleDefault;
    UITableViewCell *cell =
        [aTableView dequeueReusableCellWithIdentifier:@"BaseCell"];
    if (!cell)
        cell = [[UITableViewCell alloc]
            initWithStyle:style reuseIdentifier:@"BaseCell"];

    ABContact *contact = [matches objectAtIndex:indexPath.row];
    cell.textLabel.text = contact.compositeName;
    return cell;
}
```

From the Library of Eric Berna

```
- (BOOL)personViewController:
       (ABPersonViewController *) personViewController
    shouldPerformDefaultActionForPerson:(ABRecordRef)person
    property:(ABPropertyID)property
    identifier:(ABMultiValueIdentifier)identifierForValue
{
    // Reveal the item that was selected
    if ([ABContact propertyIsMultiValue:property])
    {
        NSArray *array =
            [ABContact arrayForProperty:property inRecord:person];
        NSLog(@"%@", [array objectAtIndex:identifierForValue]);
    }
    else
    {
        id object =
            [ABContact objectForProperty:property inRecord:person];
        NSLog(@"%@", [object description]);
    }

    return NO;
}

// Respond to user taps by displaying the person viewcontroller
- (void) tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    ABContact *contact = [matches objectAtIndex:indexPath.row];
    ABPersonViewController *pvc =
        [[ABPersonViewController alloc] init];
    pvc.displayedPerson = contact.record;
    pvc.personViewDelegate = self;
    // pvc.allowsEditing = YES; // optional editing
    [self.navigationController
        pushViewController:pvc animated:YES];
}
```

Get This Recipe's Code

To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from this book, go to the folder for Chapter 17 and open the project for this recipe.

From the Library of Eric Berna

Recipe: Accessing Contact Image Data

Recipe 17-2 expands on Recipe 17-1 by adding contact image thumbnails to each table cell. It does this by creating a new thumbnail from the contact's data. When image data is available, that image is rendered onto the thumbnail. When it is not, the thumbnail is left blank. Upon being drawn, the image is assigned to the cell's `imageView`.

Figure 17-3 shows the interface for this recipe. In this screenshot, a search is in progress (matching against the letter "e"). The records that match the search each display their image thumbnail.

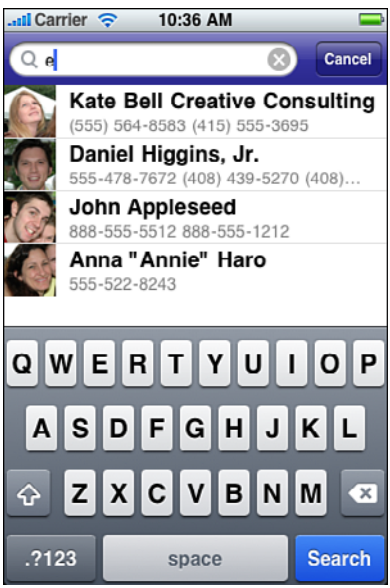


Figure 17-3 You can easily retrieve and display the image data associated with Address Book contacts.

Notice how simple it is to create and use thumbnails in this recipe. It takes just a few lines of code to build a new image context, draw into it (for contacts with images), and save it out to a `UIImage` instance.

Recipe 17-2 **Displaying Address Book Images in Table Cells**

```
- (UITableViewCell *)tableView:(UITableView *)aTableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Dequeue or create a cell
    UITableViewCellStyle style =  UITableViewCellStyleSubtitle;
    UITableViewCell *cell =
```

From the Library of Eric Berna

```
        [aTableView dequeueReusableCellWithIdentifier:@"BaseCell"];
    if (!cell)
        cell = [[UITableViewCell alloc]
            initWithStyle:style reuseIdentifier:@"BaseCell"];

    ABContact *contact = [matches objectAtIndex:indexPath.row];
    cell.textLabel.text = contact.compositeName;
    cell.detailTextLabel.text = contact.phonenumbers;

    CGSize small = CGSizeMake(48.0f, 48.0f);
    UIGraphicsBeginImageContext(small);
    UIImage *image = contact.image;
    if (image)
        [image drawInRect:(CGRect){.size = small}];
    cell.imageView.image = UIGraphicsGetImageFromCurrentImageContext();
    UIGraphicsEndImageContext();

    return cell;
}
```

> **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 this book, go to the folder for Chapter 17 and open the project for this recipe.

Recipe: Picking People

The AddressBookUI framework offers a people picker controller, which lets you browse your entire contacts list. It works similarly to the individual contact screen, but does not require an a priori contact selection. The `ABPeoplePickerNavigationController` class presents the interactive browser shown in Figure 17-4.

Allocate and display the controller before presenting it modally. Make sure to set the `peoplePickerDelegate` property, which allows you to catch user interactions with the view.

```
- (void) action: (UIBarButtonItem *) bbi
{
    ABPeoplePickerNavigationController *ppnc =
        [[ABPeoplePickerNavigationController alloc] init];
    ppnc.peoplePickerDelegate = self;
    [self presentModalViewController:ppnc animated:YES];
}
```

From the Library of Eric Berna

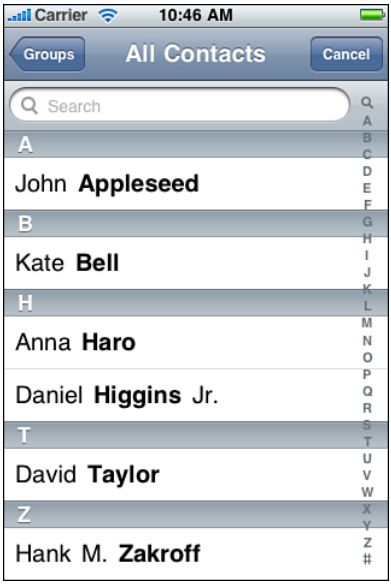


Figure 17-4 The iPhone people picker navigation control enables users to search through the contacts database and select a person or organization.

When you declare the `ABPeoplePickerNavigationControllerDelegate` protocol, your class must implement the following three methods. These methods respond to users when they tap a contact, or any of a contact's properties, or when the user taps Cancel:

- `peoplePickerNavigationController:shouldContinueAfterSelecting-Person:`—When users tap a contact, you have two choices. You can accept the person as the final selection and dismiss the modal view controller or you can navigate to the individual display. To pick just the person, this method returns `NO`. To continue to the individual screen, return `YES`. The second argument contains the selected person, in case you want to stop after selecting any `ABPerson` record.
- `peoplePickerNavigationController:shouldContinueAfterSelecting-Person:property:identifier:`—This method does not get called until the user has progressed to an individual contact display screen. Then, it's up to you whether to return control to your program (return `NO`) or to continue (return `YES`). You can determine which property has been tapped and to recover its value using the code from Recipe 17-1. Although this method should be optional, it is not at the time of writing this book.
- `peoplePickerNavigationControllerDidCancel:`—When a user taps Cancel, you still want a chance to dismiss the modal view. This method catches the cancel event, allowing you to use it to perform the dismissal.

From the Library of Eric Berna

Recipe 17-3 presents the simplest possible people picking example. It presents the picker and waits for a user to select a contact. When the user does so, it dismisses the picker and changes the view controller title (in the navigation bar) to show the composite name of the selected person. Returning `NO` from the primary callback means the property callback will never be called. You must still include it in your code as all three methods are required.

Recipe 17-3 **Picking People**

```
- (BOOL)peoplePickerNavigationController:
        (ABPeoplePickerNavigationController *)peoplePicker
    shouldContinueAfterSelectingPerson:(ABRecordRef)person
{
    self.title = [[ABContact contactWithRecord:person] compositeName];
    [self dismissModalViewControllerAnimated:YES];
    return NO;
}

- (BOOL)peoplePickerNavigationController:
        (ABPeoplePickerNavigationController *)peoplePicker
    shouldContinueAfterSelectingPerson:(ABRecordRef)person
    property:(ABPropertyID)property
    identifier:(ABMultiValueIdentifier)identifier
{
    // required method that is never called in the people-only-picking
    [self dismissModalViewControllerAnimated:YES];
    return NO;
}

- (void)peoplePickerNavigationControllerDidCancel:
    (ABPeoplePickerNavigationController *)peoplePicker
{
    [self dismissModalViewControllerAnimated:YES];
}

- (void) action: (UIBarButtonItem *) bbi
{
    ABPeoplePickerNavigationController *ppnc =
        [[ABPeoplePickerNavigationController alloc] init];
    ppnc.peoplePickerDelegate = self;
    [self presentModalViewController:ppnc animated:YES];
}
```

From the Library of Eric Berna

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 this book, go to the folder for Chapter 17 and open the project for this recipe.

Recipe: Limiting Contact Picker Properties

When you need users to pick a certain kind of property, such as an e-mail address, you won't want to present users with a person's street address or fax number. Limit the picker's displayed properties to show just those items you want the users to select from. Figure 17-5's picker has been limited to e-mail selection.

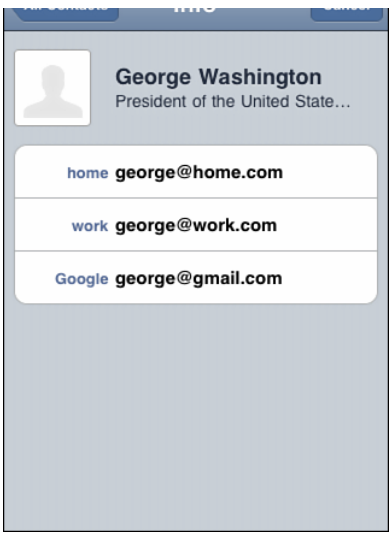


Figure 17-5 The people picker's displayed properties allows you to choose which properties to present to users (in this case, e-mail only).

To make this happen, choose the displayed properties by submitting an array of property types to the controller. Set the picker's `displayedProperties` property. Recipe 17-4 offers two picking options, one for e-mail, the other for phone numbers. Although these examples use a single property for the properties array, you can choose to display any number of properties.

From the Library of Eric Berna

Recipe 17-4 **Choosing Display Properties**

```
- (BOOL)peoplePickerNavigationController:
        (ABPeoplePickerNavigationController *)peoplePicker
    shouldContinueAfterSelectingPerson:(ABRecordRef)person
{
    // Continue onto the detail screen
    return YES;
}

- (BOOL)peoplePickerNavigationController:
        (ABPeoplePickerNavigationController *)peoplePicker
    shouldContinueAfterSelectingPerson:(ABRecordRef)person
    property:(ABPropertyID)property
    identifier:(ABMultiValueIdentifier)identifier
{
    // Guaranteed to only be working with e-mail or phone here
    [self dismissModalViewControllerAnimated:YES];
    NSArray *array =
       [ABContact arrayForProperty:property inRecord:person];
    self.title = (NSString *)[array objectAtIndex:identifier];
    return NO;
}

- (void)peoplePickerNavigationControllerDidCancel:
    (ABPeoplePickerNavigationController *)peoplePicker
{
    // Respond to cancel
    [self dismissModalViewControllerAnimated:YES];
}

#define NUMBER(X) [NSNumber numberWithInt:X]

- (void) email: (UIBarButtonItem *) bbi
{
    ABPeoplePickerNavigationController *ppnc =
        [[ABPeoplePickerNavigationController alloc] init];
    ppnc.peoplePickerDelegate = self;
    [ppnc setDisplayedProperties:
        [NSArray arrayWithObject:NUMBER(kABPersonEmailProperty)]];
    [self presentModalViewController:ppnc animated:YES];
}

- (void) phone: (UIBarButtonItem *) bbi
{
    ABPeoplePickerNavigationController *ppnc =
        [[ABPeoplePickerNavigationController alloc] init];
```

From the Library of Eric Berna

```
    ppnc.peoplePickerDelegate = self;
    [ppnc setDisplayedProperties:
        [NSArray arrayWithObject:NUMBER(kABPersonPhoneProperty)]];
    [self presentModalViewController:ppnc animated:YES];
}
```

> **Get This Recipe's Code**
>
> To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from this book, go to the folder for Chapter 17 and open the project for this recipe.

Recipe: Adding and Removing Contacts

Allow your users to create new contacts with the `ABNewPersonViewController` class. This view controller offers an editing screen that simplifies the interactive creation of a new Address Book entry. After allocating and initializing the view controller, start by creating a new contact and assigning it to the `displayedPerson` property. If you want, you can prefill the contact's record with properties first.

Next, assign the `newPersonViewDelegate`, making sure to declare the `ABNewPersonViewControllerDelegate` protocol in the delegate class. Delegates receive one new callback: `newPersonViewController:didCompleteWithNewPerson:`. This callback is sent for both selection and cancel events. Check the `person` parameter to determine which case applies, as shown in Recipe 17-5. If the person is `nil`, the user tapped *Cancel*, and the new contact should not be added to the Address Book. If the user taps *Done* after editing the new contact, add the contact data to the Address Book and save it.

To remove a contact from the Address Book, the user is first presented with standard people picker navigation controller and solicited for a selection. When the user selects a name, the callback method asks them to confirm deletion before continuing. Once confirmed, the Address Book must be saved. Removing a record from the Address Book requires a save step, just like adding a record.

Recipe 17-5 **Using the New Person View Controller**

```
// See the Alerts chapter for more details about modal queries
- (BOOL) ask: (NSString *) aQuestion
{
    UIAlertView *alertView = [[UIAlertView alloc]
        initWithTitle:aQuestion message:nil delegate:nil
        cancelButtonTitle:@"Cancel" otherButtonTitles:@"Okay",
        nil];
    ModalAlertDelegate *delegate =
        [ModalAlertDelegate delegateWithAlert:alertView];
    int response = [delegate show];
    return response;
}
```

From the Library of Eric Berna

```
#pragma mark NEW PERSON DELEGATE METHODS
- (void)newPersonViewController:
        (ABNewPersonViewController *)newPersonViewController
    didCompleteWithNewPerson:(ABRecordRef)person
{
    if (person)
    {
        ABContact *contact = [ABContact contactWithRecord:person];
        self.title = [NSString stringWithFormat:
            @"Added %@", contact.compositeName];

        NSError *error;
        BOOL success =
            [ABContactsHelper addContact:contact withError:&error];
        if (!success)
        {
            NSLog(@"Could not add contact. %@",
                error.localizedFailureReason);
            self.title = @"Error.";
        }

        [ABStandin save:nil];
    }
    else
        self.title = @"Cancelled";

    [self.navigationController popViewControllerAnimated:YES];
}

#pragma mark PEOPLE PICKER DELEGATE METHODS
- (BOOL)peoplePickerNavigationController:
        (ABPeoplePickerNavigationController *)peoplePicker
    shouldContinueAfterSelectingPerson:(ABRecordRef)person
{
    [self dismissModalViewControllerAnimated:YES];
    ABContact *contact = [ABContact contactWithRecord:person];

    NSString *query = [NSString stringWithFormat:
        @"Really delete %@?",  contact.compositeName];
    if ([self ask:query])
    {
        self.title = [NSString stringWithFormat:@"Deleted %@",
            contact.compositeName];
        [contact removeSelfFromAddressBook:nil];
        [ABStandin save:nil];
    }
```

From the Library of Eric Berna

```
    return NO;
}

- (BOOL)peoplePickerNavigationController:
        (ABPeoplePickerNavigationController *)peoplePicker
    shouldContinueAfterSelectingPerson:(ABRecordRef)person
    property:(ABPropertyID)property
    identifier:(ABMultiValueIdentifier)identifier
{
    [self dismissModalViewControllerAnimated:YES];
    return NO;
}

- (void)peoplePickerNavigationControllerDidCancel:
    (ABPeoplePickerNavigationController *)peoplePicker
{
    [self dismissModalViewControllerAnimated:YES];
}

#pragma mark Respond to User Requests
- (void) add
{
    // create a new view controller
    ABNewPersonViewController *npvc =
        [[ABNewPersonViewController alloc] init];

    // Create a new contact
    ABContact *contact = [ABContact contact];
    npvc.displayedPerson = contact.record;

    // Set delegate
    npvc.newPersonViewDelegate = self;

    [self.navigationController
        pushViewController:npvc animated:YES];
}

- (void) remove
{
    ABPeoplePickerNavigationController *ppnc =
        [[ABPeoplePickerNavigationController alloc] init];
    ppnc.peoplePickerDelegate = self;
    [self presentModalViewController:ppnc animated:YES];
}
```

From the Library of Eric Berna

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 this book, go to the folder for Chapter 17 and open the project for this recipe.

Modifying and Viewing Individual Contacts

The `ABPersonViewController`'s `allowsEditing` property enables your user to edit contacts. You create a controller, assign an Address Book record as its `displayedPerson`, set its editable property, and display it. Here's a typical example of this feature in a callback method. Upon resolving to a particular contact, the application creates a new person presentation and pushes it onto the navigation stack.

```
- (void)unknownPersonViewController:
        (ABUnknownPersonViewController *)unknownPersonView
    didResolveToPerson:(ABRecordRef)person
{
    // Handle cancel events
    if (!person) return;

    ABPersonViewController *abpvc = [[ABPersonViewController alloc] init];
    abpvc.displayedPerson = person;
    abpvc.allowsEditing = YES;
    abpvc.personViewDelegate = self;

    [self.navigationController pushViewController:abpvc animated:YES];
}
```

User Edits

When the `allowsEditing` property has been set to `YES`, an *Edit* button appears at the top-right of the displayed person presentation. Users may tap this button and modify contact details. Edits are automatically propagated to the Address Book. You do not have to explicitly add calls to save the data; this is handled for you by the CocoaTouch AddressBookUI implementation. Unlike the Contacts application, the *Delete Contact* option will not appear in red at the bottom of the controller.

If you want to allow the user to view the contact information without enabling edits, switch the `allowsEditing` property value to `NO`. Otherwise, leave the presentation code as-is. No other modifications are necessary.

Delegate Methods

The person view controller has only one delegate callback method. You specify if tapping items like e-mail addresses or URLs, should perform actions based on those taps (i.e.,

From the Library of Eric Berna

opening Mail or linking to Safari). Return YES or NO as desired. When set to YES, be aware that control may pass out of the application in response to these actions.

```
#pragma mark PERSON DELEGATE
- (BOOL)personViewController:
        (ABPersonViewController *)personViewController
    shouldPerformDefaultActionForPerson:(ABRecordRef)person
    property:(ABPropertyID)property
    identifier:(ABMultiValueIdentifier)identifierForValue
{
    return NO;
}
```

Recipe: The "Unknown" Person Controller

What happens when you have some information like an e-mail address or a phone number, but you don't have a contact to associate with it yet? The ABUnknownPersonViewController allows you to add that information to a new or existing contact. This class exists to associate known properties with unknown contacts. It works like this.

You allocate and initialize the view controller and then create and prefill a record. Recipe 17-6 creates a new Address Book record and adds a single randomly generated phone number.

You can add more items if you want, but typically, this controller is used to add a single bit of information. For example, you might provide a service that generates new disposable e-mail address (like fakemailgenerator.com) and then offers to add that new address to a contact. The unknown person controller specializes in that kind of add-one-item utility. Consider Figure 17-6 (left), which shows this recipe's controller with its random phone number.

The Create New Contact and Add to Existing Contact buttons are controlled by the allowsAddingToAddressBook property. When a user taps on the new contact button, a form appears that is prefilled with the properties from that record. If you disable this adding property, the buttons do not appear. The allowsAction property enables users to on form elements to connect to an e-mail address, to phone numbers, to Web pages, and so forth. This provides a handy way to present a list of interactive contact information and URLs with an already-defined view controller. Figure 17-6 (right) shows how you might use this to solicit a phone call from users.

The alternateName and message properties provide text that fills the name and organization fields. Although you can populate these fields with data via the record, the alternate options do not transfer to contacts. Therefore, they provide a nice way to prompt the user without side effects.

From the Library of Eric Berna

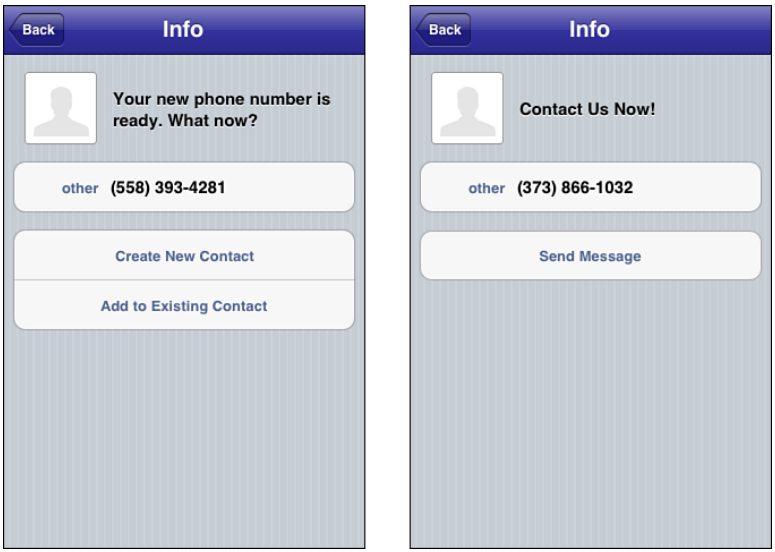


Figure 17-6 (Left) The unknown person controller allows you to display properties and add them to a new or existing contact. (Right) When you disable the Address Book features and enable actions, you create a sheet that offers one-tap access to phone numbers, e-mail addresses, URLs, and so forth.

Set the `unknownPersonViewDelegate` property and declare the `ABUnknownPersonViewControllerDelegate` protocol to receive the `unknownPersonViewController:didResolveToPerson:` method. Called when the user taps *Done*, this method allows you to recover the record that the data was saved to. It also provides a place where you can programmatically pop the view controller, knowing that the user has finished interaction with the dialog.

Recipe 17-6 **Working with the Unknown Controller**

```
#pragma mark UNKNOWN PERSON DELEGATE
- (void)unknownPersonViewController:
        (ABUnknownPersonViewController *)unknownPersonView
    didResolveToPerson:(ABRecordRef)person
{
    // Handle cancel events
    if (!person) return;

    ABPersonViewController *abpvc =
        [[ABPersonViewController alloc] init];
```

From the Library of Eric Berna

```
    abpvc.displayedPerson = person;
    abpvc.allowsEditing = YES;
    abpvc.personViewDelegate = self;

    [self.navigationController
         pushViewController:abpvc animated:YES];
}

#pragma mark PERSON DELEGATE
- (BOOL)personViewController:
        (ABPersonViewController *)personViewController
    shouldPerformDefaultActionForPerson:(ABRecordRef)person
    property:(ABPropertyID)property
    identifier:(ABMultiValueIdentifier)identifierForValue
{
    return NO;
}

- (BOOL)unknownPersonViewController:
        (ABUnknownPersonViewController *)personViewController
    shouldPerformDefaultActionForPerson:(ABRecordRef)person
    property:(ABPropertyID)property
    identifier:(ABMultiValueIdentifier)identifier
{
    return YES;
}

#pragma mark Action
- (NSString *) randomPhone
{
    NSString *phone = [NSString
        stringWithFormat:@"%d%d%d-%d%d%d-%04d",
        2 + (rand() % 8), (rand() % 10), 1 + (rand() % 9),
        2 + (rand() % 8), (rand() % 10), 1 + (rand() % 9),
        rand() % 10000];
    return phone;
}

- (void) addPhone
{
    ABUnknownPersonViewController *upvc =
        [[ABUnknownPersonViewController alloc] init];
    upvc.unknownPersonViewDelegate = self;

    ABContact *contact = [ABContact contact];
    NSString *phone = [self randomPhone];
    [contact addPhoneItem:phone withLabel:kABOtherLabel];
```

From the Library of Eric Berna

```
    upvc.allowsActions = NO;
    upvc.allowsAddingToAddressBook = YES;
    upvc.message = @"Your new phone number is ready. What now?";
    upvc.displayedPerson = contact.record;

    [self.navigationController pushViewController:upvc animated:YES];
}

- (void) call
{
    ABUnknownPersonViewController *upvc =
        [[ABUnknownPersonViewController alloc] init];
    upvc.unknownPersonViewDelegate = self;

    ABContact *contact = [ABContact contact];
    NSString *phone = [self randomPhone];
    [contact addPhoneItem:phone withLabel:kABOtherLabel];

    upvc.allowsActions = YES;
    upvc.allowsAddingToAddressBook = NO;
    upvc.message = @"Contact Us Now!";
    upvc.displayedPerson = contact.record;

    [self.navigationController pushViewController:upvc animated:YES];
}
```

Get This Recipe's Code

To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from this book, go to the folder for Chapter 17 and open the project for this recipe.

One More Thing: Adding Random Contact Art

The Monster ID project consists of a collection of body part art that can be compiled together to form random pictures. It was developed by Andreas Gohr, and was inspired by a Web post by Don Park and the combinatoric critters. Built up by adding predrawn arms, legs, a body, and so forth, the resulting composite image produces a full creature.

The `randomImage` method in Recipe 17-7 builds a monster image from its components. This art can be assigned directly to an Address Book contact or can be used as the seed for a new contact using the unknown person controller. Either way, you can use an `ABContact` instance and assign an image using its `image` property. If you're not using an AddressBookUI controller, don't forget to save the Address Book after updating a record's image.

From the Library of Eric Berna

Recipe 17-7 uses the unknown person controller approach. When the user taps an action button, it requests a Monster ID image and presents the controller, which is shown in Figure 17-7. The image appears in the Info page, and the standard create/add buttons allow users to add the generated art to a new or existing contact. Users can tap Back to return without adding the image. A new monster generates each time the sheet appears.

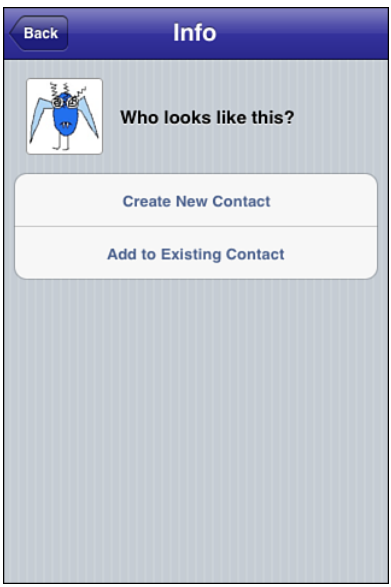


Figure 17-7 You can add images to contacts with the unknown person controller as well as e-mail addresses, phone numbers, and other text-based data.

Recipe 17-7 **Combining Random Art with the Unknown Person Controller**

```
// Graphics from: http://www.splitbrain.org/go/monsterid
- (UIImage *) randomImage
{
    // Build a random image based on the monster id art

    CGRect rect = CGRectMake(0.0f, 0.0f, 120.0f, 120.0f);
    UIGraphicsBeginImageContext(CGSizeMake(120.0f, 120.0f));

    UIImage *part;
    part = [UIImage imageNamed:IMAGEFILE(@"oldarms_%d.png", 5)];
    [part drawInRect:rect];
    part = [UIImage imageNamed:IMAGEFILE(@"oldlegs_%d.png", 5)];
    [part drawInRect:rect];
```

From the Library of Eric Berna

```
    part = [UIImage imageNamed:IMAGEFILE(@"oldbody_%d.png", 15)];
    [part drawInRect:rect];
    part = [UIImage imageNamed:IMAGEFILE(@"oldmouth_%d.png", 10)];
    [part drawInRect:rect];
    part = [UIImage imageNamed:IMAGEFILE(@"oldeyes_%d.png", 15)];
    [part drawInRect:rect];
    part = [UIImage imageNamed:IMAGEFILE(@"oldhair_%d.png", 5)];
    [part drawInRect:rect];

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

    return image;
}


- (void) addAvatar
{
    ABUnknownPersonViewController *upvc =
        [[ABUnknownPersonViewController alloc] init];
    upvc.unknownPersonViewDelegate = self;

    ABContact *contact = [ABContact contact];
    contact.image = [self randomImage];

    upvc.allowsActions = NO;
    upvc.allowsAddingToAddressBook = YES;
    upvc.message = @"Who looks like this?";
    upvc.displayedPerson = contact.record;

    [self.navigationController
        pushViewController:upvc animated:YES];
}
```

Get This Recipe's Code

To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from this book, go to the folder for Chapter 17 and open the project for this recipe.

From the Library of Eric Berna

Summary

This chapter introduced Address Book core functionality for both the AddressBook and AddressBookUI frameworks. You've read about how these work in this heavily technical chapter. After all that work, here are a few final thoughts about the frameworks you just encountered:

- Although useful, the low-level Address Book functions can prove frustrating to work with directly. The various helper classes that accompany this chapter may help make your life easier. All the wrappers have been updated for iOS 5 and automatic reference counting.
- Accessing and modifying an Address Book image works like any other field. Supply image data instead of strings, dates, or multivalue arrays. Don't hesitate to use image data in your contacts applications.
- The view controllers provided by the AddressBookUI framework work seamlessly with the underlying AddressBook routines. There's no need to roll your own GUIs for most common Address Book interaction tasks.
- Do you work with Vcards? The AddressBook framework provides two Vcard-specific functions that allow you to create Vcard data from an array of records (`ABPersonCreateVCardRepresentationWithPeople()`) and transform Vcard data into ABRecordRefs (`ABPersonCreatePeopleInSourceWithVCardRepresentation ()`).
- The unknown person controller provides a really great way to store specific information (such as a company's e-mail address, an important Web site, etc.) into a contact while allowing user discretion for where (or whether) to place that information.

From the Library of Eric Berna

18

iCloud Basics

Ready to emerge from the confines of a single device or desktop system? Welcome to iCloud. iCloud lets you share data beyond the limitations of individual hardware and into a central data-storage hub. With iCloud, your applications can access and update shared documents without explicit synchronization. Changes on one unit propagate out to all other registered systems, updating when that hardware next accesses the Internet. This completely passive coordination is yours for free via iCloud. All you need to do is set up some entitlements and incorporate its features in your applications. It's surprisingly easy to use. This chapter introduces iCloud, shows you how it works, introduces the key classes you'll use, and walks you through several recipes and code snippets that show you how to use it in your own projects.

Setting Up an iCloud-Compatible Project

To get started on your first iCloud project, you add a simple entitlement. Open the Project > TARGETS > Entitlements pane and check the Enable Entitlements box shown in Figure 18-1. This option builds a new entitlement file, adding a trio of entitlement types. These types include iCloud containers (the shared folders where you can add and read data), iCloud Key-Value Store (enabling a shared dictionary look-up system), and Keychain Access Groups (allowing secure item sharing between applications).

Although Apple briefly flirted with enabling iCloud access on an application-by-application basis during the iOS 5 beta period, it eventually decided to enable iCloud for all applications without special provisioning. Any application identifier added to the provisioning portal is automatically enabled with iCloud access. This decision means that, when you check the Entitlements box, you've done everything needed to get started with iCloud.

From the Library of Eric Berna

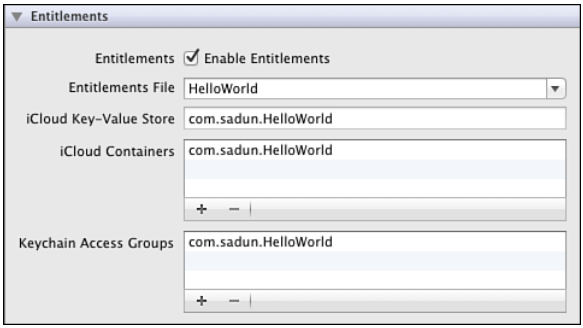


Figure 18-1 Enable iCloud functionality by establishing entitlements for your application. You can use the + button to add additional containers to your application. The first-listed container, which you can see defaults to the application identifier, acts as the default ubiquitous store.

> **Note**
>
> At the time this book was written, you cannot use iCloud on the iOS Simulator. You will want to have at least two iOS devices on-hand to work with, enabling you to test both data propagation and conflict resolution.

Determining Your Team Identifier Prefix

Every cloud application uses a special internal prefix—and it's not the one you're probably thinking of, the one associated with application identifiers at the provisioning portal. iCloud service uses your developer identifier (the "Team Identifier Prefix") to enable you to write to and access cloud-based data. This identifier is associated with your account and listed in the iOS Member Center in the Your Account page as your organization or individual ID.

You can look this identifier up by visiting developer.apple.com/ios, and clicking Member Center. Sign in with your developer credentials and view the Your Account tab. The Program Membership section lists the identifier, between the account name and account type entries.

Outside of visiting the developer site, the easiest way to locate the prefix you're looking for is to take a peek into your Team Provisioning Profile. Use the Organizer > Devices > Provisioning Profiles pane to locate the provision file. Right-click iOS Team Provisioning Profile and choose Reveal Profile in Finder. Open this file in TextEdit and search for the `TeamIdentifier` key. It should appear exactly once in the file. Directly below the key you will find a single-value, which is your team identifier prefix.

```
<key>TeamIdentifier</key>
<array>
    <string>YOURIDENTIFIER</string>
</array>
```

From the Library of Eric Berna

Containers

This team identifier prefix is used for all shared iCloud data containers. You can see this prefix in action by examining the default ubiquity folder for your application. Consider the following call. When passed `nil` as an argument, it looks up the default ubiquitous (i.e., iCloud) container URL. The default container is always the first item listed in the `com.apple.developer.ubiquity-container-identifiers` entitlement.

```
NSURL *ubiquity = [[NSFileManager defaultManager]
    URLForUbiquityContainerIdentifier:nil];
NSLog(@"Ubiquity: %@", ubiquity);
```

When executed, this code prints out the path to that container. The path includes a tilda-escaped version of the application identifier, complete with a team identifier prefix. The identifier ensures that containers are unique to each developer.

```
2011-08-29 12:17:41.094 HelloWorld[8069:707] Ubiquity:
file://localhost/private/var/mobile/Library/Mobile%20Documents/\
YOURTEAMIDENTIFIER~com~sadun~HelloWorld/
```

The default value added to the initial entitlement includes just one container, associated with the application identifier of the current project.

```
<key>com.apple.developer.ubiquity-container-identifiers</key>
<array>
    <string>$(TeamIdentifierPrefix)com.sadun.HelloWorld</string>
</array>
```

You can extend this entitlement to add more identifiers, so long as those items are associated with your team prefix, e.g.

```
<key>com.apple.developer.ubiquity-container-identifiers</key>
<array>
    <string>$(TeamIdentifierPrefix)com.sadun.HelloWorld</string>
    <string>$(TeamIdentifierPrefix)com.sadun.AnotherApp</string>
    <string>$(TeamIdentifierPrefix)com.sadun.SharedStorage</string>
</array>
```

Adding more identifiers enables you to add cross-application document sharing for any application built by the same developer account. Adding, for example, `com.sadun.AnotherApp` to these entitlements allows the `HelloWorld` application to access the `AnotherApp` application's default container.

The reverse domain-name identifiers do *not* have to be tied to a specific application. Although the entitlement domains default to the application name (`com.sadun.HelloWorld`), you can add shared items or replace the default container with a shared name (e.g., `com.sadun.SharedStorage`).

From the Library of Eric Berna

Retrieving Identifiers in Code

Your team prefix plays an important role in iCloud development. Outside of the default container (the first entitlement item), which you can access by passing `nil` to the ubiquity URL query, you cannot access other containers without your team identifier prefix. You can hard code the prefix as a string constant, or you can retrieve the prefix with a simple method call. This method extracts the prefix for you:

```
+ (NSString *) teamPrefix
{
    NSURL *ubiquity = [[NSFileManager defaultManager]
        URLForUbiquityContainerIdentifier:nil];
    if (!ubiquity) return nil;
    NSArray *elements = [[ubiquity.path lastPathComponent]
        componentsSeparatedByString:@"~"];
    if (!elements.count) return nil;
    return [elements objectAtIndex:0];
}
```

When extracted, you can create a valid container identifier by combining the prefix to a container name string (e.g., `MYPREFIX.com.sadun.SharedStorage`).

```
+ (NSString *) containerize: (NSString *) anIdentifier
{
    NSString *prefix = [self teamPrefix];
    if (!prefix) return nil;
    return [NSString stringWithFormat:@"%@.%@", prefix, anIdentifier];
}
```

Calls to `URLForUbiquityContainerIdentifier:` should not return `nil`. If they do, go through the following checklist:

- Did you add an entitlement for the container identifier?
- Did you use the proper team id prefix in the identifier?
- Did you enable iCloud on the device?

Making sure that you can retrieve a ubiquity container URL forms the first step for iCloud development.

> **Note**
>
> The two methods you saw in this section, `teamPrefix` and `containerize:`, are both part of a `CloudHelper` class, which was developed as part of this chapter. This class contains a variety of helper routines, mostly implemented at the class level, that allow you to more easily work with iCloud data in your applications.

From the Library of Eric Berna

How iCloud Works

In its simplest form, iCloud is merely a special folder. It lives on the iOS device in `/private/var/mobile/Library/Mobile Documents/`. You have partial permission to read from and write to this folder. That permission is limited to the container folder or folders declared in your application entitlements. iOS monitors and regularly updates this folder, synchronizing its contents with offsite storage. When you write to this folder, even just by dropping in an image or text file, it automatically joins the iCloud.

Your applications read and write local content from the folder. They do not need to directly access the Internet or check for connectivity. That is all handled for you by iCloud. All you have to deal with is local file URLs.

Updates arrive asynchronously at the device, where they can be handled immediately by open applications or addressed later by suspended ones. You're responsible for designing your application to handle situations where cloud access is restricted. Apps must implement conflict resolution strategies to deal with staggered updates from multiple parties to the same cloud account.

Imagine this scenario: Mom is flying to Detroit for a meeting, so she's offline for a while. Dad is at the gym, and Junior is checking in from the local library. Each of them adds new items to a shared To Do list application, and may mark some off as "done." That application must be able to handle these chronologically displaced updates in some smart manner, preferably involving users in its trickier decisions.

The `UIDocument` class, and its CoreData-enabled child `UIManagedDocument`, transform this otherwise complicated scenario into a simple notification. Applications are notified when a file's state changes, allowing you to detect and resolve conflict situations as Dad and Junior's live updates overlap with each other, and when Mom disembarks from the plane and her changes have to somehow meld with the ones created while she was offline. You might automatically merge all added items and globally apply deletions, or you might lend weight to newer versions, or you might ask the current user what to do. (If you use Core Data with the `UIManagedDocument` subclass, much of this is automated for you.)

Think about that conflict resolution scenario. Most of your coding responsibilities involve wrestling with how to resolve these conflicts in each of your applications. The rest of iCloud is actually both simple and mechanical, as you'll see in the following sections.

iCloud Container Folders

Developing and using iCloud is primarily a matter of writing to, monitoring, and reading from the right folder at the right time. Under iOS 5 and later, `NSFileManager` creates authenticated on-demand URL access to that shared storage, which you can query at will. Your application *must* be properly entitled for iCloud access to happen. Consider the following call; you would substitute an actual container identifier you added to your entitlement:

From the Library of Eric Berna

```
NSString *container =
    [CloudHelper containerize:com.sadun.HelloWorld"];
NSURL *destinationURL = [[NSFileManager defaultManager]
    URLForUbiquityContainerIdentifier: container];
```

This call either returns a URL, in the case of proper entitlements, developer prefix, and container identifier or `nil`. The container the URL points to represents the base of where your application is allowed to store data. This is part of your application's *ubiquitous data scope*.

Your data should actually live one step down from this container, in a `Documents` folder, much as it does in your Sandbox. This is part of your application's *ubiquitous documents scope*. Adding a documents folder allows your user to list and delete files individually from their Settings application. iCloud groups any items outside `Documents` into a single unit; these files must be deleted all at once. By creating a shared document home, you offer your user more nuanced access to his or her data.

You will encounter the terms *ubiquitous data scope* and *ubiquitous documents scope* throughout this chapter. Your application's ubiquitous data scope refers to all iCloud container content that lies outside `Documents` folders. The documents scope refers to iCloud content within `Documents` folders.

Building a Shared Document Home

After entitling your application, you'll be able to establish a home in which your app's iCloud documents will live. This involves creating a `Documents` subfolder in your container. The following method requests a URL that points to the container storage. If it does not find a subfolder there, it builds a new `Documents` folder. By adding it to the container, it mirrors the way your documents are stored in the application sandbox.

As you can see, this involves nothing more than using standard NSFileManager calls. The method checks to see if the folder exists. If it does not, it creates a new subfolder in the container.

```
// Return the iCloud Data URL
+ (NSURL *) ubiquityDataURLForContainer:
    (NSString *) container
{
    return [[NSFileManager defaultManager]
        URLForUbiquityContainerIdentifier:container];
}

// Return the iCloud Documents URL
+ (NSURL *) ubiquityDocumentsURLForContainer:
    (NSString *) container
{
    return [[self ubiquityDataURLForContainer:container]
        URLByAppendingPathComponent:@"Documents"];
}
```

From the Library of Eric Berna

```
// Prepare the Documents folder
+ (BOOL) setupUbiquityDocumentsFolderForContainer:
    (NSString *) container
{
    NSError *error;
    NSURL *targetURL =
        [self ubiquityDocumentsURLForContainer:container];

    // Create the ubiquity documents folder if needed
    if (![[NSFileManager defaultManager]
        fileExistsAtPath:targetURL.path])
    {
        if (![[NSFileManager defaultManager]
            createDirectoryAtPath:targetURL.path
            withIntermediateDirectories:YES
            attributes:nil error:&error])
        {
            NSLog(@"Error: %@", error.localizedFailureReason);
            return NO;
        }
    }
    return YES;
}
```

Writing To and Reading From Ubiquitous Storage

"Ubiquitous" items refer to those files that live in iCloud. Once you have a URL that points to your ubiquitous documents folder and not just the primary container, you may write to it and read from it just as you would in your sandbox. Be cautioned. This is generally a bad idea unless your files are entirely atomic, so you will not encounter any conflicts. Any file you create in that folder automatically updates its contents to iCloud.

Latency depends on network connectivity and device settings. Usually, small files update within a few seconds. You can monitor the connectivity by examining a device's console logs. Status change notifications indicate what percentage of the document has uploaded, ending with the item becoming current.

```
Jul  9 16:16:05 unknown librariand[50] <Notice>: STATUS CHANGE: url =
/private/var/mobile/Library/Mobile
Documents/XYZZYPLUGH~com~sadun~CloudLearning/Documents/Output2.jpg status =
0xd02 (HasRevisions New Current InCloud) transferred = 0
Jul  9 16:16:05 unknown librariand[50] <Notice>: STATUS CHANGE: url =
/private/var/mobile/Library/Mobile
Documents/XYZZYPLUGH~com~sadun~CloudLearning/Documents/Output2.jpg status =
0x40c02 (HasRevisions Current InCloud Uploading) transferred = 50
Jul  9 16:16:05 unknown librariand[50] <Notice>: <LBFileProvider: 0xe82c900>
itemIsNowCurrent: - url=/private/var/mobile/Library/Mobile
Documents/XYZZYPLUGH~com~sadun~CloudLearning/Documents/Output2.jpg
```

From the Library of Eric Berna

Using the Organizer's device console logs allows you to monitor iCloud updates throughout your application runs. As there are no official notifications that you can subscribe to, I recommend leaving the console open as you develop and debug your iCloud apps. This allows you to better see what is going on behind the scenes with the iCloud daemon, providing live feedback of those changes.

Editing iCloud Storage

Determine which application files exist in your iCloud storage by visiting Settings > iCloud > Storage & Backup > Manage Storage > Documents and Data. There, listed (currently) under Unknown, you'll find any application-generated items. Figure 18-2 shows a single third-party text file created by my test application. Hopefully, Apple will update this in future releases to reflect the document source rather than "Unknown."

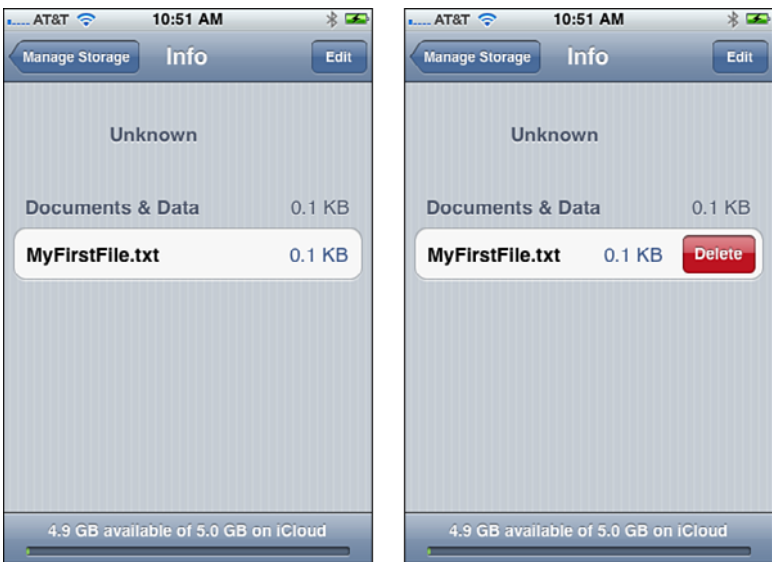


Figure 18-2 Application-generated iCloud documents appear in the Documents & Data section in Settings.

You can delete cloud files here by swiping-to-delete or tapping Edit. Removing a file here removes it from the cloud folder on your device and from the iCloud central storage. This is especially important for testing, because it allows you to reset across all devices.

Designing for iCloud Ubiquity

There are many issues to take into account when building applications for iCloud. As a rule, design your non-Core Data applications around `UIDocument` and your Core Data

From the Library of Eric Berna

applications around `UIManagedDocument`. These classes were built for iCloud from the ground up and help simplify many of the synchronization issues that arise from using ubiquitous data.

When using iCloud, let your users choose which items to save to iCloud. Documents can live just as easily in your sandbox as they can in the cloud. Never force your users to pay extra money to Apple to gain ubiquitous access for items they don't want or need.

Be clever when letting users freely move their data into and out of the cloud. Removing data from the cloud can mean two things. The user may want to delete the file entirely, or they may want to move the data from the cloud to the device. Whether deleting or transferring, moving a file from the cloud removes that data from all other devices. Design for both scenarios.

When you need a file to survive its deletion from the cloud on a device that did not initiate its removal, you can take advantage of a user's `Library/Caches` folder. Proactively caching copies of the data whenever the device is used allows you to store local versions so users don't entirely lose access to formerly ubiquitous data. The caches folder isn't backed up either by iTunes or iCloud, so you don't place as big a burden on a user's paid iCloud quota. At the same time, local caching does affect available space for the device, and you should allow the cached copy to be removed when the user no longer wants access to the data.

In an ideal world, Apple would introduce an optional iCloud *pending-deletion* state in addition to deleting cloud items immediately or transferring them from the cloud to the sandbox. Such an option would allow applications to check items out of the cloud on each registered device; the file would be removed once all registered devices were offered the opportunity to copy the cloud-stored version or when the user forced a full delete request.

You can create a rough approximation of this behavior by adding an "automatically check out copies of this document" feature to your applications. Copy the document from cloud to cache, and then, once the application detects that the document is no longer available from iCloud, move it from cache to sandbox for local use.

As a final note, remember that file names are case-sensitive on iOS, but are not on OS X. Take care when creating iCloud file names on iOS so you do not establish pairs of names that differ only by case. Add simple in-app checks to prevent users from doing so.

> **Note**
>
> Do not use the cloud to store sensitive information like passwords. Use the device's keychain, and do not attempt to make that data ubiquitous. Familiarize yourself with Apple's iCloud security policies that explain this stance in further depth.

Recipe: Trying Out iCloud

The best way to convince yourself that you have properly entitled and established iCloud is to create a file from code and assure yourself that it propagates across to all devices.

From the Library of Eric Berna

That's exactly what Recipe 18-1 does. It establishes the documents folder discussed earlier, writes a small text file ("Hello World") to the cloud, and builds a custom URL that you can use to access that new file from outside the cloud.

Requesting this URL requires net access and can take several seconds to retrieve. Please be patient during this time. The URL remains valid for a limited time, so the code also prints out the expiration date for that access period to the console.

After running this code, you'll want to ensure that cloud files reached their destination on each test device you have signed into the same account. Try downloading the file from the URL and, only after that, delete it using iCloud settings.

Recipe 18-1 **Giving iCloud a Spin**

```
+ (NSArray *) contentsOfUbiquityDocumentsFolderForContainer:
    (NSString *) container
{
    NSURL *targetURL =
        [self ubiquityDocumentsURLForContainer:container];
    if (!targetURL) return nil;

    NSArray *array = [[NSFileManager defaultManager]
        contentsOfDirectoryAtPath:targetURL.path error:nil];
    return array;
}

+ (NSArray *) contentsOfUbiquityDataFolder
{
    return [self contentsOfUbiquityDataFolderForContainer:nil];
}

+ (NSURL *) ubiquityDocumentsFileURL: (NSString *) filename
    forContainer: (NSString *) container
{
    if (!filename) return nil;
    NSURL *fileURL =
        [[self ubiquityDocumentsURLForContainer:container]
            URLByAppendingPathComponent:filename];
    return fileURL;
}

+ (NSURL *) ubiquityDocumentsFileURL: (NSString *) filename
{
    return [self ubiquityDocumentsFileURL:filename
        forContainer:nil];
}
```

From the Library of Eric Berna

```
// Log the contents of the ubiquity documents folder
- (void) list: (id) sender
{
    NSLog(@"Contents of Documents: %@",
        [CloudHelper contentsOfUbiquityDocumentsFolder]);
}

// Create a new text file
- (void) create: (id) sender
{
    // Write to default container
    NSError *error;
    NSURL *targetURL = [CloudHelper
        ubiquityDocumentsFileURL:@"MyFirstFile.txt"];

    // Write a "Hello World" Text file to the cloud
    NSLog(@"About to write to file.");
    if (![@"Hello from the cloud!" writeToURL:targetURL
        atomically:YES encoding:NSUTF8StringEncoding error:nil])
    {
        NSLog(@"Error writing to %@: %@",
            targetURL, error.localizedFailureReason);
        return;
    }

    // Retrieve a URL to share
    NSDate __autoreleasing *date;
    NSURL *url = [[NSFileManager defaultManager]
        URLForPublishingUbiquitousItemAtURL:targetURL
        expirationDate:&date error:&error];
    if (!url)
        NSLog(@"Error creating publishing URL: %@",
            error.localizedFailureReason);
    else
    {
        NSLog(@"iCloud URL: %@", url);
        NSLog(@"Expires: %@", date);
    }
}

- (void) loadView
{
    [super loadView];
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"List", @selector(list:));
```

From the Library of Eric Berna

```
    self.navigationItem.leftBarButtonItem =
        BARBUTTON(@"Create", @selector(create:));

    // Default Ubiquity Container
    NSLog(@"Data: %@", [CloudHelper ubiquityDataURL]);

    // Shared Ubiquity Container
    NSString *sharedIdentifier =
        [CloudHelper containerize:@"com.sadun.SharedStorage"];
    NSLog(@"Shared: %@", [CloudHelper
        ubiquityDataURLForContainer:sharedIdentifier]);

    // Nonexistent Ubiquity Container. Will error.
    NSString *nonexistentIdentifier =
        [CloudHelper containerize:@"com.sadun.nonexistent"];
    NSLog(@"Nonexistent: %@", [CloudHelper
        ubiquityDataURLForContainer:nonexistentIdentifier]);

    BOOL success = [CloudHelper setupUbiquityDocumentsFolder];
    if (success)
        NSLog(@"Default ubiquity Documents folder is ready");
    else
    {
        NSLog(@"Error setting up ubiquitous documents folder");
        return;
    }
}
```

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 this book, go to the folder for Chapter 18 and open the project for this recipe.

Working with UIDocument

The `UIDocument` class is an abstract class that supports cloud-ready asynchronous reading and writing of data. `UIDocument` simplifies the way you work with iCloud data, because it handles nearly all the issues you encounter about coordinating local data with cloud data. Recipe 18-1's "just write to the right folder" approach cannot be recommended for serious development. What `UIDocument` does is add a shell for managing your document, letting you send read, save, and write requests through it in a safer and more structured way.

From the Library of Eric Berna

You never work directly with the `UIDocument` class. Instead, you subclass it and implement several simple key features that link the class to the ubiquitous data.

Establishing the Document as a File Presenter

`UIDocument` implements the `NSFilePresenter` class. This means that you can register the class to receive updates whenever its cloud data gets updated by telling `NSFileCoordinator` to add the document as "presenter." This is terribly unhelpful naming, I know, but what it means is that a presenter class is one that takes a strong interest in knowing when outside changes happen to a given file. Receiving alerts about these changes allows it to update its GUI presentation in response.

The registration process works like this. You create a document and a coordinator, initializing the coordinator with the document as its performer as the following code snippet shows. Registering the new document/presenter with `NSFileCoordinator` class allows it to receive updates. You must perform both these steps, both at the instance and class level.

```
imageDocument = [[ImageDocument alloc] initWithFileURL:ubiquityURL];
imageDocument.delegate = self;
coordinator = [[NSFileCoordinator alloc]
    initWithFilePresenter:imageDocument];
[NSFileCoordinator addFilePresenter:imageDocument];
```

Although all the actual coordination implementation details are handled for you via the `UIDocument` class (for example, `UIDocument` handles all file coordination issues for you, so you do not need to manually coordinate reading and writing), this set up portion is not. Unless you establish the presenter/coordinator pair manually, your document *will not* receive updates from the cloud.

Moving to the Cloud

`NSFileManager` allows you to move local files to and from the cloud using the `setUbiquitous:itemAtURL:destinationURL:error:` method. What this method does is nothing more than move (not copy) the file in question from your sandbox into the central iCloud folder and back. However, it does so safely in an approved manner.

The method takes three arguments. The first establishes the direction of movement. `YES` brings items from the sandbox to the cloud and `NO` the reverse. The second argument must always be the source URL and the third its destination. The first argument is a little pointless in that the source URL will always be either sandbox or cloud and the destination must be the reverse. If all three arguments do not line up, the method will fail.

To move to the cloud, use `YES`, the local sandbox URL, and the destination cloud URL, as shown here. To remove an item from the cloud, use `NO`, then the cloud URL, and then the local sandbox URL.

```
if (![[NSFileManager defaultManager] setUbiquitous:YES
    itemAtURL:localURL destinationURL:ubiquityURL error:&error])
```

From the Library of Eric Berna

```
{
    NSLog(@"Error making local file ubiquitous. %@",
        [error localizedFailureReason]);
    return;
}
```

Retrieve the cloud URL by appending the file name to the results of the `ubiquityDocumentsURL` method, which is shown in Recipe 18-1.

Evicting Items from the Cloud

Removing an item from the cloud by setting its ubiquity to `NO` deletes it from both the cloud and from all of your user's devices and computers. At times, you may want to remove a local copy in order to force it to refresh from the cloud without deleting it. To do this, do not alter the file's ubiquity. Instead, call an `NSFileManager` eviction method, as follows. Evicting a file allows iCloud to retrieve a new copy, but does not remove it from central cloud storage.

```
+ (BOOL) evictFile: (NSString *) filename
    forContainer: (NSString *) container
{
    if (!filename) return NO;

    NSURL *targetURL =
        [self ubiquityDocumentsFileURL:filename
            forContainer:container];
    BOOL targetExists =
        [[NSFileManager defaultManager]
            fileExistsAtPath:targetURL.path];
    if (!targetExists) return NO;

    NSError *error;
    if (![[NSFileManager defaultManager]
        evictUbiquitousItemAtURL:targetURL error:&error])
    {
        NSLog(@"Error evicting current copy of %@.: %@",
            filename, error.localizedFailureReason);
        return NO;
    }

    return YES;
}

+ (BOOL) evictFile: (NSString *) filename
{
    return [self evictFile:filename forContainer:nil];
}
```

From the Library of Eric Berna

In addition to file eviction, you can actually force iCloud to download a new copy. This is of use only when you write your own data handlers. You pretty much never have to do this if you use Apple's recommended `UIDocument` and `UIManagedDocument` classes.

```
+ (BOOL) forceDownload: (NSString *) filename
    forContainer: (NSString*) container
{
    if (!filename) return NO;
    NSURL *targetURL =
        [self ubiquityDocumentsFileURL:filename
            forContainer:container];
    if (!targetURL) return NO;

    NSError *error;
    if (![self evictFile:filename forContainer:container])
        return NO;
    if (![[NSFileManager defaultManager]
          startDownloadingUbiquitousItemAtURL:targetURL
          error:&error])
    {
        NSLog(@"Error starting download of %@: %@", filename,
              error.localizedFailureReason);
        return NO;
    }

    return YES;
}

+ (BOOL) forceDownload: (NSString *) filename
{
    return [self forceDownload:filename forContainer:nil];
}
```

Opening and Saving Files

You open and save `UIDocument` instances to read them in from a file and to store your changes out. Opening the file is generally performed as part of creating the document. You provide it with a file URL pointing to the stored file, and ask it to open the file. For iCloud materials, this path must point to the correct place in the ubiquitous documents folder to work with iCloud and not to a file saved locally. You can also use the `UIDocument` class to access and read non-iCloud files that have been moved back to the sandbox. This flexibility unifies how you access files, letting users choose which items to send to the cloud.

From the Library of Eric Berna

```
imageDocument = [[ImageDocument alloc] initWithFileURL:theURL];
[imageDocument openWithCompletionHandler:^(BOOL success) {
    NSLog(@"Open file was: %@", success ? @"successful" : @"failure");
    if (success) imageView.image = imageDocument.image;}];
```

Each time you make changes to your document, you'll want to save those updates as transparently as possible. Saving data, especially to small files, is cheap. Making updates as soon as your user creates changes allows you to build a persistent system that synchronizes with your user's adjustments.

```
imageDocument.image = image;
imageDocument saveToURL:imageDocument.fileURL
    forSaveOperation:UIDocumentSaveForOverwriting
    completionHandler:^(BOOL success){
      NSLog(@"Attempt to save to URL %@",
          success ? @"succeeded" : @"failed");
    }];
```

`UIDocument` is fully undo-aware and provides its own `undoManager` property hook. This allows you to save without having to worry about creating permanent changes from which you cannot recover.

Subscribing to UIDocument Notifications

When working with documents, your primary class must subscribe to a single notification: `UIDocumentStateChangedNotification`. Your handler uses that notification to handle conflict issues, allowing your document to update to the latest version. At its simplest, the notification subscription looks like this. This snippet resolves all conflict issues by accepting the most recent version and updating its presentation. This example, which works with the code in Recipe 18-2, is built around a simple image presentation app. When new images arrive, the application view updates and shows them.

```
[[NSNotificationCenter defaultCenter]
    addObserverForName: UIDocumentStateChangedNotification
    object:nil queue:[NSOperationQueue mainQueue]
    usingBlock:^(NSNotification __strong *notification)
    {
        if (imageDocument.documentState == UIDocumentStateInConflict)
        {
            NSError *error;
            NSLog(@"Resolving conflict. (Newest wins.)");
            if (![NSFileVersion
                    removeOtherVersionsOfItemAtURL:
                        imageDocument.fileURL
                    error:&error])
        {
            NSLog(@"Error removing other document versions: %@",
                error.localizedFailureReason);
```

From the Library of Eric Berna

```
            return;
        }
        imageView.image = imageDocument.image;
    }
}];
```

This code, which uses "newest version wins," represents the simplest way you can handle document conflicts using `NSFileVersion`, a class built around time-based file revisions. Although it's the simplest approach, it's not always the best. You'll probably want to use a little more finesse in your applications. iOS can store more than one version of a file at a time. You can retrieve all conflicted versions of the file version class through something like the following. The snippet

```
[NSFileVersion unresolvedConflictVersionsOfItemAtURL:
    imageDocument.fileURL]
```

returns an array of file version instances. You can iterate through those instances to help resolve your conflicts, choosing which of the conflicted saves best represents the state that you want to present to the user.

When you need to examine the actual version data to better judge or merge that material, you can pull each file's versioned contents from the instance's `URL` property. From there, you can select which file version to use by removing the other versions (as in the previous example) or replace the file data entirely.

The document's `documentState` property lets you know whether your saves are currently conflicted or if you're clear to make changes. Additional states include `UIDocumentStateNormal`, which allows users to make unhindered changes; `UIDocumentStateClosed`, indicating that the document wasn't opened open properly (or has been deliberately closed) and is probably not valid for edits; `UIDocumentStateSavingError`, telling you that the file could not be saved out; and `UIDocumentStateEditingDisabled`, which means that the file is busy. A busy file is never safe for user edits, so your application should disable changes until the state changes back to something safer.

You may allow edits during both normal and conflict states but you should inform your user when conflict issues may prevent updates from propagating out to the cloud, such as the Mom-on-a-Plane scenario discussed earlier. Apple recommends offering visual feedback, such as a green light for unhindered updates and a yellow one for modifications that may need later intervention to integrate.

Recipe: Subclassing UIDocument

At a minimum, each document instance inherits two core responsibilities: to produce its actual document data on demand and to update itself to new data in response to iCloud refreshes. When you subclass `UIDocument`, you must implement two methods. Recipe 18-2 presents the barest-bones document subclass that you might create:

From the Library of Eric Berna

- The `loadFromContents:error:` method brings new content to the class, allowing it to update its internal data. When it receives its new data, it pings an informal delegate to let it know that it's a good time to reload the image.
- The `contentsForType :error:` method gets called whenever the user saves data locally. Here, you produce a data representation of your document that can be stored to the cloud, allowing you to publish that data ubiquitously.

This recipe demonstrates the simplicity of creating a cloud-based document. With just these two methods, you can store nearly any kind of data ubiquitously.

Recipe 18-2 **Simple Cloud Document Class**

```
@interface ImageDocument : UIDocument
@property (strong) UIImage *image;
@property (weak) id delegate;
@end

#define SAFE_PERFORM_WITH_ARG(THE_OBJECT, THE_SELECTOR, THE_ARG) \
    (([THE_OBJECT respondsToSelector:THE_SELECTOR]) ? \
    [THE_OBJECT performSelector:THE_SELECTOR withObject:THE_ARG] : nil)

@implementation ImageDocument
@synthesize image, delegate;

- (BOOL) loadFromContents:(id)contents ofType:(NSString *)typeName
    error:(NSError *__autoreleasing *)outError
{
    NSLog(@"Loading external content");
    self.image = nil;
    if ([contents length] > 0)
        self.image = [[UIImage alloc] initWithData:contents];
    if (delegate)
        SAFE_PERFORM_WITH_ARG(delegate, @selector(imageUpdated:), self);

    return YES;
}

- (id) contentsForType:(NSString *)typeName
    error:(NSError *__autoreleasing *)outError
{
    NSLog(@"Publishing content");
    return UIImageJPEGRepresentation(self.image, 0.75f);
}
@end
```

From the Library of Eric Berna

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 this book, go to the folder for Chapter 18 and open the project for this recipe.

Metadata Queries and the Cloud

Two new metadata scopes, `NSMetadataQueryUbiquitousDataScope` and `NSMetadataQueryUbiquitousDocumentsScope`, point to the iCloud's Data and Documents folders. You can add an ongoing query to monitor these folders, creating a callback whenever the folder contents change. This allows you to know when new items appear or disappear and update your file listings accordingly. Listing 18-1 offers a pair of methods that demonstrate how to use this technology to query and monitor the folder:

- The first method, `startMonitoringUbiquitousDocumentsFolder`, sets up a monitor for the Documents folder scope, listening for update notifications. When these occur, the method uses an informal delegate callback to update a client.
- The second method, `stopMonitoringUbiquitousDocumentsFolder`, cancels the notification observation, allowing your code to stop any ongoing monitoring.

To search the ubiquitous container outside the Documents folder, use the `NSMetadataQueryUbiquitousDataScope` search scope instead of `NSMetadataQueryUbiquitousDocumentsScope`. You can add both scopes via `setSearchScopes:` or pass an empty array to remove search scope limitations. In iOS 5, you cannot yet use the `NSMetadataQuery` class to search sandboxes or other parts of iOS outside of authorized iCloud container storage.

The `startMonitoringUbiquitousDocumentsFolder` method, shown here, automatically extracts filenames (via the `NSMetadataItemFSNameKey` key) to an array and returns those to an informal delegate. You might want to adapt the method to return the raw `NSMetadataItem` objects or URLs (`NSMetadataItemURLKey`) instead.

Use file updates to refresh data sources but never depend on them to precisely indicate an update event. You may receive some callbacks that seem untied to anything other than iCloud checking in. These callbacks are never excessive in terms of overwhelming your application, but "contents changed" doesn't *always* precisely mean that contents have actually changed. You will receive a callback when new items appear or are removed, but you may also receive callbacks when iCloud is just "being iCloud."

If you use more than one container in your application, make sure you adapt your callbacks to determine which container was updated. Listing 18-1 assumes you'll be working with a single ubiquitous Documents folder.

From the Library of Eric Berna

Listing 18-1 **Monitoring iCloud**

```
- (void) startMonitoringUbiquitousDocumentsFolder
{
    // Remove any existing query - stored in local instance variable
    if (query) [query stopQuery];

    // Search for all file names
    query = [[NSMetadataQuery alloc] init];
    query.predicate = [NSPredicate predicateWithFormat:
        @"NSMetadataItemFSNameKey == '*'"];
    query.searchScopes = [NSArray arrayWithObject:
        NSMetadataQueryUbiquitousDocumentsScope];

    // Subscribe to query updates
    [[NSNotificationCenter defaultCenter] addObserverForName:
        NSMetadataQueryDidUpdateNotification
     object:nil queue:[NSOperationQueue mainQueue]
      usingBlock:^(NSNotification __strong *notification)
      {
         NSLog(@"Contents changed in ubiquitous documents folder");

         // disable the query while iterating through its results
         [query disableUpdates];
         NSMutableArray *array = [NSMutableArray array];
         for (NSMetadataItem *item in query.results)
             [array addObject: [item
                 valueForAttribute:NSMetadataItemFSNameKey]];
         [query enableUpdates];

         // use an informal delegate callback with file names
         if (delegate) SAFE_PERFORM_WITH_ARG(delegate,
             @selector(folderContentsHaveChanged:),
             array);
     }];

    [query startQuery];
}

- (void) stopMonitoringUbiquitousDocumentsFolder
{
    [[NSNotificationCenter defaultCenter]
        removeObserver:self
        name:NSMetadataQueryDidFinishGatheringNotification
        object:nil];
    [query stopQuery];
    query = nil;
}
```

From the Library of Eric Berna

Handy Routines

This section presents a number of extremely simple routines that work together to allow you to manage files across your sandbox and cloud Documents folders. These methods assume your files live at the top level of these folders for several reasons:

- Both the container and sandbox Documents folders are made accessible to your users via Settings and iTunes. Therefore, they are directly visible.
- Neither system supports subfolders at this time.
- Your users own these folders directly, according to Apple's guidelines, and should have full access and visibility to all materials contained within.

Local URLs

Working with the sandbox Documents folder involves little more than querying for the documents directory. These methods return the documents folder and URLs that point inside that folder. This `localFileURL:` method does not test the URL it returns to see if it points to a valid object.

```
+ (NSString *) localDocumentsPath
{
    return [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
        NSUserDomainMask, YES) objectAtIndex:0];
}

+ (NSURL *) localDocumentsURL
{
    return [NSURL fileURLWithPath:[self localDocumentsPath]];
}

+ (NSURL *) localFileURL: (NSString *) filename
{
    if (!filename) return nil;
    NSURL *fileURL = [[self localDocumentsURL]
        URLByAppendingPathComponent:filename];
    return fileURL;
}
```

Locating the Ubiquitous Documents Folder

Supply a container identifier and `NSFileManager` returns URLs pointing to the root of that cloud folder. Appending Documents produces the ubiquitous documents URL.

```
+ (NSURL *) ubiquityDataURLForContainer: (NSString *) container
{
    return [[NSFileManager defaultManager]
```

From the Library of Eric Berna

```
        URLForUbiquityContainerIdentifier:container];
}

+ (NSURL *) ubiquityDocumentsURLForContainer: (NSString *) container
{
    return [[self ubiquityDataURLForContainer:container]
        URLByAppendingPathComponent:@"Documents"];
}
```

Once you have that folder's URL, you can append file names to it to point to items within the Documents subfolder. Like the `localFileURL:` method, the following methods do not test the URL it returns to see if it points to a valid object:

```
+ (NSURL *) ubiquityDataFileURL: (NSString *) filename
    forContainer: (NSString *) container
{
    if (!filename) return nil;
    NSURL *fileURL =
        [[self ubiquityDataURLForContainer:container]
            URLByAppendingPathComponent:filename];
    return fileURL;
}

+ (NSURL *) ubiquityDocumentsFileURL: (NSString *) filename
    forContainer: (NSString *) container
{
    if (!filename) return nil;
    NSURL *fileURL =
        [[self ubiquityDocumentsURLForContainer:container]
            URLByAppendingPathComponent:filename];
    return fileURL;
}
```

Retrieving Files

The first three of these methods test whether a file exists, locally or in the data or documents ubiquity. The final method uses these tests to retrieve a file URL, searching first the local folder and then the cloud. If a matching file is found, it returns that URL.

```
+ (BOOL) isLocal: (NSString *) filename
{
    if (!filename) return NO;
    NSURL *targetURL = [self localFileURL:filename];
    if (!targetURL) return NO;
    return [[NSFileManager defaultManager]
        fileExistsAtPath:targetURL.path];
}
```

From the Library of Eric Berna

```
+ (BOOL) isUbiquitousData: (NSString *) filename
    forContainer: (NSString *) container
{
    if (!filename) return NO;
    NSURL *targetURL = [self ubiquityDataFileURL:filename
        forContainer:container];
    if (!targetURL) return NO;
    return [[NSFileManager defaultManager]
        fileExistsAtPath:targetURL.path];
}

+ (BOOL) isUbiquitousDocument: (NSString *) filename
    forContainer: (NSString *) container
{
    if (!filename) return NO;
    NSURL *targetURL = [self ubiquityDocumentsFileURL:filename
        forContainer:container];
    if (!targetURL) return NO;
    return [[NSFileManager defaultManager]
       fileExistsAtPath:targetURL.path];
}

+ (NSURL *) fileURL: (NSString *) filename
    forContainer:(NSString *)container
{
    if ([self isLocal:filename])
        return [self localFileURL:filename];
    if ([self isUbiquitousDocument:filename
        forContainer:container])
        return [self ubiquityDocumentsFileURL:filename
            forContainer:container];
    if ([self isUbiquitousData:filename
        forContainer:container])
        return [self ubiquityDataFileURL:filename
            forContainer:container];
    return nil;
}
```

Setting Ubiquity

The following method applies ubiquity to a file based on its name, not its current location. This routine checks for each possible file condition. If the file is already found where it's supposed to end up, the method returns with success. If the file cannot be found, it fails. Otherwise, the code moves the file to or from the cloud as requested, wrapping away the otherwise fussy implementation details of the native `NSFileManager` method.

From the Library of Eric Berna

```
+ (BOOL) setUbiquitous:(BOOL)yorn for:(NSString *)filename
    forContainer:(NSString *)container
{
    if (!filename) return NO;

    NSError *error;
    NSURL *localURL = [self localFileURL:filename];
    NSURL *ubiquityURL =
        [self ubiquityDocumentsFileURL:filename
            forContainer:container];

    BOOL localFound = [self isLocal:filename];
    BOOL ubiquityFound =
        [self isUbiquitousDocument:filename
            forContainer:container];

    // Check file not found
    if (!localFound && !ubiquityFound) return NO;

    // Check the two "nothing to be done" cases
    if (!yorn && localFound) return YES;
    if (yorn && ubiquityFound) return YES;

    // ubiquitous to local
    if (!yorn)
    {
        // Move file away from cloud
        if (![[NSFileManager defaultManager]
              setUbiquitous:NO
              itemAtURL:ubiquityURL
              destinationURL:localURL
              error:&error])
        {
            NSLog(@"Error removing %@ from %@ storage: %@",
                filename, container,
                error.localizedFailureReason);
            return NO;
        }

        return YES;
    }

    // local to ubiquitous
    if (![[NSFileManager defaultManager]
          setUbiquitous:YES
          itemAtURL:localURL
```

From the Library of Eric Berna

```
          destinationURL:ubiquityURL
          error:&error])
    {
        NSLog(@"Error moving %@ to %@ storage: %@",
            filename, container,
            error.localizedFailureReason);
        return NO;
    }

    return YES;
}
```

Deleting Files

These methods take three approaches to deleting files. The first deletes a local copy at the top of the Documents sandbox folder. The second moves items from the cloud (data and documents) and then deletes it locally. The third finds the file first and then invokes one of the previous solutions.

```
+ (BOOL) deleteLocal: (NSString *) filename
{
    NSURL *targetURL = [self localFileURL:filename];
    if (![[NSFileManager defaultManager]
        fileExistsAtPath:targetURL.path])
    {
        NSLog(@"Local file not found: %@", filename);
        return NO;
    }

    NSError *error;
    BOOL success = [[NSFileManager defaultManager]
        removeItemAtURL:targetURL error:&error];
    if (!success)
        NSLog(@"Error removing file %@: %@",
            filename, error.localizedFailureReason);

    return success;
}

+ (BOOL) deleteUbiquitousDocument:(NSString *)filename
    forContainer:(NSString *)container
{
    NSURL *targetURL =
        [self ubiquityDocumentsFileURL:filename
            forContainer:container];
    if (![[NSFileManager defaultManager]
```

From the Library of Eric Berna

```
        fileExistsAtPath:targetURL.path])
    {
        NSLog(@"Ubiquitous file not found: %@", filename);
        return NO;
    }

    // Remove from ubiquity and then delete
    BOOL success = [self setUbiquitous:NO
        for:filename forContainer:container];
    if (success)
        return [self deleteLocal:filename];
    return NO;
}

+ (BOOL) deleteUbiquitousData:(NSString *)filename
    forContainer:(NSString *)container
{
    NSError *error;
    BOOL success;

    NSURL *targetURL =
        [self ubiquityDataFileURL:filename
            forContainer:container];
    success = [[NSFileManager defaultManager]
        fileExistsAtPath:targetURL.path];
    if (!success)
    {
        NSLog(@"Ubiquitous file not found: %@", filename);
        return NO;
    }

    success = [[NSFileManager defaultManager]
        removeItemAtURL:targetURL error:&error];
    if (!success)
    {
        NSLog(@"Could not remove item at path: %@",
            error.localizedFailureReason);
        return NO;
    }

    return YES;
}

+ (BOOL) deleteDocument: (NSString *) filename
    forContainer:(NSString *)container
{
```

From the Library of Eric Berna

```
    // If local, delete it.
    if ([self isLocal:filename])
        return [self deleteLocal:filename];
    return [self deleteUbiquitousDocument:filename
        forContainer:container];
}
```

Retrieving Modification Dates

These methods retrieve the modification date for a file, in the top level of either Documents folder (sandbox or ubiquitous), depending on where the file is found.

```
+ (NSDate *) modificationDateForURL:(NSURL *) targetURL
{
    if (!targetURL) return nil;

    NSDictionary *attributes =
        [[NSFileManager defaultManager]
            attributesOfItemAtPath:targetURL.path error:nil];
    if (!attributes) return nil;

    return [attributes fileModificationDate];
}

+ (NSDate *) modificationDateForFile: (NSString *) filename
{
    if (!filename) return nil;
    return [self modificationDateForURL:
        [self fileURL:filename]];
}

+ (NSTimeInterval) timeIntervalSinceModification:
    (NSString *) filename
{
    return [[NSDate date] timeIntervalSinceDate:
        [self modificationDateForFile:filename]];
}
```

Recipe: Accessing the Ubiquitous Key-Value Store

iCloud's key-value store lets you share small amounts of state information between devices using a ubiquitous property list. "Small" means you are limited to a total of 256Kb per key-value store and 64Kb per individual value.

Those limits mean you won't be using this utility for sharing images and other large data items. They're meant to save state such as the last page read, the default account

From the Library of Eric Berna

name, or other such tiny bits of information that can propagate between devices to enhance the seamless movement of the user from one device to the next.

> **Note**
>
> Apple's policy is that key-value stores are not charged against the user's 5GB iCloud quota. At the time this book was written, Apple was not actively enforcing the 64K quota mentioned for limited key-value stores. That quota will likely come on line by the time this book is published.

How Key-Value Stores Work

Your application's ubiquitous key-value store is nothing more than a distributed defaults dictionary that can be reached from devices registered to the same iCloud account. This store isn't meant to replace `NSUserDefaults`; it augments it by allowing you to develop applications that enable users to resume work regardless of the device used.

Imagine your user begins editing a ubiquitous document on a certain device and later launches your app on a second one. The key value store would let you load the right document (by storing the file name) and scroll it to the current position (by storing the current offset), so the user resumes interaction at the point that he or she left off.

If you want to use more extensive shared defaults, you can create those defaults in a property list file *outside* the ubiquitous Documents folder. This has two advantages. First, it lets you store whatever data you need without regard to the key-value store's quota limitations. Second, it hides those defaults from the user. Data stored outside Documents cannot be directly seen or edited by the user but they remain visible to your application.

Accessing the Key-Value Store

You can retrieve the shared store from its class method, namely `[NSUbiquitousKeyValueStore defaultStore]`. Once retrieved, query it like you would user defaults, e.g.:

```
kv = [NSUbiquitousKeyValueStore defaultStore];
switchView.on = [kv boolForKey:@"switchIsOn"];
```

Similarly, set its value using the same dictionary-style method calls. As with user defaults, you must synchronize your changes to publish them from memory to the persistent iCloud store. Although the system *may* call synchronize automatically, make a habit of manually performing the synchronization. This ensures that your changes are pushed immediately rather than with a possible several-second delay.

```
- (void) toggleSwitch: (UISwitch *) aSwitch
{
    // Send switch update out to cloud
    [kv setBool:aSwitch.isOn forKey:@"switchIsOn"];
    [kv synchronize];
}
```

From the Library of Eric Berna

The previous sample code snippets implement a ubiquitous switch. When the user interacts with the switch on one device, the change propagates to all devices. In order for that change to register, your application must subscribe to a notification that informs it about key-value store updates.

Subscribing to Key-Value Notifications

Your application will want to know when the ubiquitous store has updated item values. Subscribe it to `NSUbiquitousKeyValueStoreDidChangeExternallyNotification`. This notification lets you determine when the store has updated and adjust your in-app settings to match.

The notification's user info dictionary provides a list of affected keys as well as a reason why the update occurred. `NSUbiquitousKeyValueStoreServerChange` means the values simply updated. In addition, your application may be notified of a quota violation (`NSUbiquitousKeyValueStoreQuotaViolationChange`) or that local changes were discarded because they could not be sent to the server for an initial sync (`NSUbiquitousKeyValueStoreInitialSyncChange`).

Recipe 18-3 builds callbacks based on these various scenarios, allowing a client class to add informal callbacks. If you don't care which key was updated here, just implement the `kvStoreUpdated:` callback, which is sent on any value change. The `kvStoreUpdatedForKeys:` callback sends an array of affected keys for more nuanced responses.

Recipe 18-3 **Subscribing to Key-Value Update Notifications**

```
[[NSNotificationCenter defaultCenter]
    addObserverForName:
        NSUbiquitousKeyValueStoreDidChangeExternallyNotification
    object:nil
    queue:[NSOperationQueue mainQueue]
    usingBlock:^(NSNotification __strong *notification) {
        NSDictionary *userInfo = [notification userInfo];

        NSUInteger reason = [[userInfo objectForKey:
            NSUbiquitousKeyValueStoreChangeReasonKey] intValue];
        NSArray *keys = [userInfo objectForKey:
        NSUbiquitousKeyValueStoreChangedKeysKey];

        // Perform updates only if there is a delegate to listen
        if (!delegate) return;

        if (reason == NSUbiquitousKeyValueStoreServerChange)
        {
             if (keys.count == 1)
                 SAFE_PERFORM_WITH_ARG(delegate,
                     @selector(kvStoreUpdatedForKey:),
                     [keys lastObject]);
```

From the Library of Eric Berna

```
            else if (keys.count)
                SAFE_PERFORM_WITH_ARG(delegate,
                    @selector(kvStoreUpdatedForKeys:),
                    keys);

            SAFE_PERFORM_WITH_ARG(delegate,
                @selector(kvStoreUpdated), nil);
        }
        else if (reason ==
            NSUbiquitousKeyValueStoreInitialSyncChange)
            SAFE_PERFORM_WITH_ARG(delegate,
               @selector(kvStorePerformedInitialSync), nil);
        else if (reason ==
            NSUbiquitousKeyValueStoreQuotaViolationChange)
            SAFE_PERFORM_WITH_ARG(delegate,
               @selector(kvStoreViolatedQuota), nil);
}];
```

> **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 this book, go to the folder for Chapter 18 and open the project for this recipe.

Recipe: UIManagedDocument and Core Data

The `UIManagedDocument` class allows you to work with ubiquitous Core Data stores with iCloud. This class integrates data with the cloud. It offers all the built-in services you need for a Core Data–compliant managed model object and extends those services to ubiquitous files. At the time this chapter was written, support for this new class was just coming online. Keep that preliminary nature in mind as you read this section as details have likely changed.

Although the class is derived from `UIDocument`, there are basic differences between the way you use `UIManagedDocument` and `UIDocument`:

- **You may want to use the data container, and not Documents**—That's because the default Core Data store created by `UIManagedDocument` (i.e., not a standard SQLite store) contains a rather complex file structure. Moving out of the Documents folder allows you to handle those items as a single unit rather than expose the individual files to the end-user in his or her iCloud preferences.
- **Listen for persistent store content changes**—In addition to document state changes, your application needs to listen for persistent store updates. `NSPersistentStoreDidImportUbiquitousContentChangesNotification` indicates your application should merge iCloud changes into your managed context.

From the Library of Eric Berna

- **Merge external changes**—Use a method similar to the one discussed in this recipe to integrate inserted, modified, or deleted objects into your data store.
- **Do not save your changes**—Unlike `UIDocument`, where you can save both early and often, you only save a `UIManagedDocument` on creation—and you do so there in a rather tricky fashion as you'll see in the steps that follow.

Keep these differences in mind as you move forward to the following how-to write-up.

Remove All Context Saves

The first iCloud refactoring step involves removing all context saves from your application. Any time you see code like this,

```
NSError *error = nil;
if (![context save:&error])
    NSLog(@"Error: %@", [error localizedDescription]);
```

go ahead and comment it out. `UIManagedDocument` handles all context saves for you.

> **Note**
>
> Context saves are a different matter from saving the managed document. With documents, you create or overwrite files using `saveToURL:forSaveOperation:completionHandler:`. The edits in this step only affect context saves.

Establish Your Identifiers and URLs

Your application should establish two key identifiers that you will use throughout your managed document implementation. They include

- A real-world ready name that your new file will use when it is saved to the sandbox (e.g., "ToDo").
- A unique name to use when saving the managed store to the cloud. Use a standard reverse-domain style for this private identifier, according to Apple standards.

Here's how those identifiers might look in your source file:

```
#define SharedFileName   @"ToDo"
#define PrivateName      @"com.sadun.CloudLearning.storage"
```

Next, create two key URLs that build on these identifiers. The first is a local URL. It points to the sandbox and uses the real-world file name. The second is a cloud URL. It points to the data storage area of your ubiquitous container and uses the private reverse domain identifier.

```
NSURL *localURL = [CloudHelper localFileURL:SharedFileName];
NSURL *cloudURL = [CloudHelper ubiquityDataFileURL:PrivateName];
```

From the Library of Eric Berna

The local and cloud URLs work as a pair. The local item points to a sandbox shell. The managed document links that shell to the persistent store, which is located in the ubiquitous container at the cloud URL.

Establish the Document

Create a new managed document object by allocating it and pointing it to the sandbox URL. Your `UIManagedDocument` instance should always point to the sandbox item and not to the ubiquitous persistent store. Here's how you can set up the object with the local URL:

```
// Establish the document by pointing to the local sandbox
document = [[UIManagedDocument alloc] initWithFileURL:localURL];
```

The document's persistent store options declare the content name and include a pointer to the cloud URL. They also should establish that the store migrates its data automatically and use an inferred mapping model. Set the options like this:

```
// Set the persistent store options to point to the cloud
NSDictionary *options = [NSDictionary dictionaryWithObjectsAndKeys:
    PrivateName, NSPersistentStoreUbiquitousContentNameKey,
    cloudURL, NSPersistentStoreUbiquitousContentURLKey,
    [NSNumber numberWithBool:YES],
        NSMigratePersistentStoresAutomaticallyOption,
    [NSNumber numberWithBool:YES],
        NSInferMappingModelAutomaticallyOption,
    nil];
document.persistentStoreOptions = options;
```

Once you're done setting the store options, you're ready to read in the file if it exists or create it. Allocating a document and setting its URL does not open the file; it merely creates an instance of the class. You must use that instance to save or open the file.

Opening the Document

Here's a basic rule of thumb: When the sandbox item exists, you can assume that the ubiquitous store it links to exists as well. The converse is not true. A ubiquitous store created on another device might not have a local shell component. In that case, you can create one.

Test for the sandbox file using `NSFileManager` and the local URL. If the file exists, open it and perform a first fetch on its data. You're then ready to start operating as you normally do.

```
if ([helper isLocal:SharedFileName])
{
    NSLog(@"Attempting to open existing file");
    [document openWithCompletionHandler:^(BOOL success){
```

From the Library of Eric Berna

```
        if (!success) {NSLog(@"Error opening file"); return;}
        NSLog(@"File opened");
        [self performFetch];;
    }];
}
```

When the file is not available, create it in the sandbox. Do this regardless of whether the ubiquitous file exists or not. If it exists, the sandbox shell links to it. If it does not, both "files" are created at the same time.

This snippet creates the file, closes it, and then opens it up again. This is probably overkill, but I found that it works consistently in my testing and, when skipping these steps, it does not. Your mileage is sure to vary, especially after the beta period for this new release is over and this book goes live. When the ubiquitous portion does not yet exist, `UIManagedDocument` automatically creates the persistent store component in the location you specified in its options. In the end, the document exists in two places on each device: the store in the cloud and the wrapper in the sandbox.

```
{
    NSLog(@"Creating file.");
    // 1. save it out, 2. close it, 3. read it back in.
    // You probably can get away with doing less
    [document saveToURL:localURL
        forSaveOperation:UIDocumentSaveForCreating
        completionHandler:^(BOOL success){
            if (!success) { NSLog(@"Error creating file"); return; }
            NSLog(@"File created");
            [document closeWithCompletionHandler:^(BOOL success){
                NSLog(@"Closed new file: %@", success ?
                    @"Success" : @"Failure");
                [document openWithCompletionHandler:^(BOOL success){
                      if (!success) {
                          NSLog(@"Error opening file for reading.");
                          return;}
                      NSLog(@"File opened for reading.");
                      [self performFetch];;
             }];
         }];
    }];
}
```

Start Observing

Edits on each device are propagated out via iCloud updates. You need to listen for persistent store updates to respond to these changes, so subscribe in your setup to the following notification:

From the Library of Eric Berna

```
[[NSNotificationCenter defaultCenter]addObserver:self
    selector:@selector(documentContentsDidUpdate:)
    name:NSPersistentStoreDidImportUbiquitousContentChangesNotification
    object:nil];
```

Implement a cloud-updated method (the name is arbitrary, although `document-ContentsDidUpdate:` is typical) to match the notification selector. The following method relays the notification data to an asynchronous method that merges the update with the current store:

```
 - (void) documentContentsDidUpdate: (NSNotification *) notification
{
    NSDictionary* userInfo = [notification userInfo];
    [context performBlock:^{
        [self mergeiCloudChanges:userInfo forContext:context];}];
}
```

From here, it's up to the `mergeiCloudChanges:forContext:` method, which you can read through in Recipe 18-4, to manage those merges. This code is again adapted from Apple sample code. It iterates through the updated, refreshed, and invalidated objects and applies those changes to the local managed context.

The merging rounds out the updates made to the standard Core Data application. All other elements of your app can remain as they were prior to iCloud integration.

Recipe 18-4 **Ubiquitous Core Data**

```
#pragma mark Initialize the Core Data Stores
- (void) initCoreData
{
    NSURL *localURL = [CloudHelper localFileURL:SharedFileName];
    NSURL *cloudURL = [CloudHelper ubiquityDataFileURL:PrivateName];

    // Create the document pointing to the local sandbox
    document = [[UIManagedDocument alloc] initWithFileURL:localURL];

    // Set the persistent store options to point to the cloud
    NSDictionary *options = [NSDictionary dictionaryWithObjectsAndKeys:
        PrivateName,
            NSPersistentStoreUbiquitousContentNameKey,
        cloudURL,
            NSPersistentStoreUbiquitousContentURLKey,
        [NSNumber numberWithBool:YES],
            NSMigratePersistentStoresAutomaticallyOption,
        [NSNumber numberWithBool:YES],
            NSInferMappingModelAutomaticallyOption,
    nil];
    document.persistentStoreOptions = options;
    context = document.managedObjectContext;
```

From the Library of Eric Berna

```
    // Register as presenter
    coordinator = [[NSFileCoordinator alloc]
        initWithFilePresenter:document];
    [NSFileCoordinator addFilePresenter:document];

    // Check at the local sandbox
    if ([helper isLocal:SharedFileName])
    {
        NSLog(@"Attempting to open existing file");
        [document openWithCompletionHandler:^(BOOL success){
            if (!success) {NSLog(@"Error opening file"); return;}
            NSLog(@"File opened");

            [self performFetch];;
        }];
    }
    else
    {
        NSLog(@"Creating file.");
        // 1. save it out, 2. close it, 3. read it back in.
        // You probably can get away with doing less
        [document saveToURL:localURL
           forSaveOperation:UIDocumentSaveForCreating
          completionHandler:^(BOOL success){
              if (!success) { NSLog(@"Error creating file"); return; }
              NSLog(@"File created");
              [document closeWithCompletionHandler:^(BOOL success){
                  NSLog(@"Closed new file: %@", success ?
                      @"Success" : @"Failure");
              [document openWithCompletionHandler:^(BOOL success){
                      if (!success) {
                          NSLog(@"Error opening file for reading.");
                          return;}
                      NSLog(@"File opened for reading.");
                      [self performFetch];;
                  }];
              }];
          }];
    }

    // Register to be notified of changes to the persistent store
    [[NSNotificationCenter defaultCenter]addObserver:self
        selector:@selector(documentStateChanged:)
        name: NSPersistentStoreDidImportUbiquitousContentChangesNotification
        object:nil];
}
```

From the Library of Eric Berna

```
#pragma mark Courtesy of Apple. Thank you Apple
// Merge the iCloud changes into the managed context
- (void)mergeiCloudChanges: (NSDictionary*)userInfo
    forContext: (NSManagedObjectContext*)managedObjectContext
{
    @autoreleasepool
    {
        NSMutableDictionary *localUserInfo =
           [NSMutableDictionary dictionary];

        // Handle the invalidations
        NSSet* allInvalidations =
            [userInfo objectForKey:NSInvalidatedAllObjectsKey];
        NSString* materializeKeys[] = { NSDeletedObjectsKey,
                NSInsertedObjectsKey };

        if (nil == allInvalidations)
        {
            int c = (sizeof(materializeKeys) / sizeof(NSString*));
            for (int i = 0; i < c; i++)
            {
                NSSet* set = [userInfo objectForKey:materializeKeys[i]];
                if ([set count] > 0)
                {
                    NSMutableSet* objectSet = [NSMutableSet set];
                    for (NSManagedObjectID* moid in set)
                        [objectSet addObject:[managedObjectContext
                            objectWithID:moid]];
                   [localUserInfo setObject:objectSet
                        forKey:materializeKeys[i]];
                }
            }

            // Handle the updated and refreshed Items
            NSString* noMaterializeKeys[] = { NSUpdatedObjectsKey,
                NSRefreshedObjectsKey, NSInvalidatedObjectsKey };
            c = (sizeof(noMaterializeKeys) / sizeof(NSString*));

            for (int i = 0; i < 2; i++)
            {
                NSSet* set = [userInfo objectForKey:noMaterializeKeys[i]];
                if ([set count] > 0)
                {
                    NSMutableSet* objectSet = [NSMutableSet set];
                    for (NSManagedObjectID* moid in set)
                    {
```

From the Library of Eric Berna

```
                    NSManagedObject* realObj =
                        [managedObjectContext
                            objectRegisteredForID:moid];
                    if (realObj)
                        [objectSet addObject:realObj];
                }
                [localUserInfo setObject:objectSet
                    forKey:noMaterializeKeys[i]];
            }
        }

        // Fake a save to merge the changes
        NSNotification *fakeSave = [NSNotification
            notificationWithName:
                NSManagedObjectContextDidSaveNotification
            object:self  userInfo:localUserInfo];
        [managedObjectContext
            mergeChangesFromContextDidSaveNotification:fakeSave];
    }
    else
        [localUserInfo setObject:allInvalidations
            forKey:NSInvalidatedAllObjectsKey];

    [managedObjectContext processPendingChanges];
    [self performSelectorOnMainThread:@selector(performFetch)
        withObject:nil waitUntilDone:NO];
  }
}

// When notified about a cloud update, start merging changes
- (void) documentContentsDidUpdate: (NSNotification *) notification
{
    NSLog(@"Cloud has been updated.");
    NSDictionary* userInfo = [notification userInfo];
    [context performBlock:^{
        [self mergeiCloudChanges:userInfo forContext:context];}];
}
```

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 this book, go to the folder for Chapter 18 and open the project for this recipe.

From the Library of Eric Berna

Summary

This chapter introduced iCloud and its ability to manage ubiquitous data. In it, you discovered the basics of how to integrate its features in your own applications. You saw how to create new provisions and entitlements, learned what's going on under the hood, and discovered how to integrate iCloud access into your applications. Here are a few final thoughts to take with you:

- Once you hand over control to a `UIDocument` subclass, there's little you need to do other than to save your changes and update your undo manager on a regular basis.
- While accepting the most recently saved item is a really easy way to go with conflict resolution, it's not necessarily the best way. Role-play how your application might be used in the real world, with all kinds of possible conflicts, and design your resolution strategy around that.
- If you are working with Core Data, use `UIManagedDocument` and its built-in Core Data integration rather than subclassing `UIDocument`.
- `UIManagedDocument` takes a simple "last-changed wins" philosophy to life. Once you update your code with the elements shown in Recipe 18-4, you can keep reusing those elements across all your projects. Unlike `UIDocument`-based apps, which require more nuanced thinking, the managed document's conflict resolution strategy can be used over and over again. That makes Core Data and `UIManagedDocument` a powerfully simple solution for iCloud integration.
- Details matter. When it comes to iCloud, you must punch every point in the process. You can use a simple Recipe 18-1 "Hello World"–style test to ensure that you're correctly accessing the cloud with your entitlements before moving on to more complicated matters. Don't forget to register your document as a file presenter.
- The reason you use your developer ID with your entitlements is to allow access between all your applications. You can create a suite of applications that all work together, allowing file transfer using a shared iCloud storage container.
- Use key-value stores to provide a continuous user experience, so your user can pick up on a new device where he or she left off on another. KV stores are meant to enhance, not replace, user defaults.
- iCloud shares data between devices for a single account, and for a single developer's applications. Multiple user accounts cannot share data into a central source and separate developers cannot create shared folders. If you need this functionality, look to providers outside iCloud or build your own Web service.

From the Library of Eric Berna

- Unlike iOS, Mac OS X HFS+ file names are generally case insensitive. Creating container IDs and file names that differ only by case will produce unhappy development outcomes. Design your iOS apps to be Mac-safe from the start and vice versa.
- Users pay for cloud storage. So, use it wisely or not at all. A little state data can go a long way, and a lot of media storage will get you nowhere fast.

From the Library of Eric Berna

This page intentionally left blank

From the Library of Eric Berna

19

Core Location and MapKit

Core Location infuses iOS with on-demand geopositioning based on a variety of technologies and sources. MapKit adds interactive in-application mapping allowing users to view and manipulate annotated maps. With Core Location and MapKit, you can develop applications that help users meet up with friends, search for local resources, or provide location-based streams of personal information. This chapter introduces these location-aware frameworks and shows you how you can integrate them into your iOS applications.

How Core Location Works

Location is meaningful. Cocoa Touch understands that. Where we compute is fast becoming just as important as how we compute and what we compute. iOS is constantly on the go, traveling along with its users throughout the course of the day, at home and on the road. Core Location integrates the device's mobility into application development.

Core Location addresses location-based programming. It enables applications to hook into location-aware Web APIs. It helps you provide geotagged content to your user and lets your user search for local resources such as restaurant and event listings. With on-demand geolocation, mobile computing opens itself up to a wide range of Web 2.0 API libraries.

All these features depend on one thing: location. And it's up to Core Location to tell your application where your users are. iOS uses several methods to locate you. Together, this approach is called assisted GPS (or A-GPS). The following sections provide a rundown of the ways devices can detect and report position with A-GPS.

GPS Positioning

On current-model iPhones and iPad 3Gs, the onboard GPS system tracks movement courtesy of a series of medium Earth orbit satellites provided by the U.S. Department of Defense. These satellites emit microwave signals, which the device picks up and uses to triangulate position to a high level of accuracy. Like any GPS system, the device requires a

From the Library of Eric Berna

clear path between the user and the satellites, so it works best outdoors and away from trees.

GPS positioning is not currently available for the iPod touch line and Wi-Fi–only iPads. These units must fall back to other ways of tracking location, just as a 3G-enabled device does when it cannot lock to a satellite signal.

Wi-Fi Positioning

In the United States, Core Location's preferred pseudo-GPS geopositioning method calls on a database of routers. Hotspot positioning offers surprisingly accurate placement. When an iOS device detects nearby Wi-Fi and WiMax routers, it uses their MAC addresses to search Apple's databases, positioning the device from that data. All iOS models, including the touch line, are Wi-Fi enabled, allowing them to scan for those units.

Wi-Fi data collection works like this. The collection provider sends drivers and pedestrians down city streets throughout its covered territories, which includes most U.S. metropolitan areas. These agents scan for Wi-Fi hotspots (called *access points*) and when found, they record the location using traditional GPS positioning matched to the Wi-Fi MAC address.

This works great when Wi-Fi routers stay still. This works terribly when people pack up their Wi-Fi routers and move with them to, say, Kentucky. That said, positioning data does get regularly updated. It provides pretty accurate positioning and can usually locate you within a few hundred feet of your actual location, even though people and their routers will continue to move to Kentucky and other places.

Cell Tower Positioning

A less accurate location approach involves cell tower positioning. Here, the iPhone or iPad uses its antenna to find the nearest four or five cell towers and then triangulates your position based on the cell tower signal strength. You've probably seen cell tower location in action; it's the kind that shows you about a half mile away from where you are standing—assuming you're not standing right next to an actual cell tower.

iPod touch units and iPad Wi-Fi units cannot use cell tower positioning, lacking the GPRS cell tower antennas that are standard issue on 3G-enabled devices. Cell tower-based location usually acts as a fallback method due to its low accuracy.

Cell Tower assignment is also used for "significant location change" monitoring. When the iPhone or iPad detects that the currently-associated cell tower for the device has changed, it generates an event that applications can subscribe to. This is a low-accuracy power-saving technique using approximate location to look up nearby attractions and other features that do not require precise positioning.

Internet Provider Positioning

There's actually a third assisted positioning approach, but it is one I've never seen iOS use with Core Location. Then again, I live in a major metropolitan area; I haven't given it a

From the Library of Eric Berna

good try. This last-ditch approach uses an Internet provider location to find the nearest mapped Internet provider's central office. This is a solution of last resort. The returned data is typically up to several miles off your actual location—unless you happen to be visiting your Internet provider. This is most often used by Web pages, in an attempt to roughly find you for localized ad services.

Hybridizing the Approaches

iOS's A-GPS approaches location in stages. Based on the accuracy level you request, it uses a fallback method. If it cannot accurately locate you with GPS or Wi-Fi technology, it falls back to cell tower location for cell-enabled devices. If that doesn't work, it presumably falls back further to Internet provider location. And if that doesn't work, it finally fails.

The latest releases of the SDK provide multiple asynchronous success callbacks for each of these fallback methods. You may receive three or four results at any time. What's more, those methods keep working over time, as the device's location changes. Each callback includes an accuracy measure, indicating the method used.

Knowing how iOS does this is important. That's because any ten attempts to grab your location may result in maybe three or four Wi-Fi successes, the remainder falling back to cell tower hits. Although you can set your desired location accuracy to the highest possible settings, unless you listen for multiple callbacks, you might miss out on catching the best possible location.

The cost to this is time. A location request may take 10 or 15 seconds to establish itself. Working with multiple requests, averaging, and best-results repetition is best done in the background away from the GUI. When possible, avoid making your user wait for your program to finish its location queries.

> **Note**
>
> Apple requires that users authorize all location requests when Core Location is first launched. Once authorized, you may use location for the duration of the application session. Users can reset their authorization via Settings > General > Reset > Reset Location Warnings. (See Apple's HT1975 knowledge base article for more details.) You can increase GPS accuracy by ensuring the unit's date, time, and timezone are set correctly in Settings > General > Date & Time. Choose Set Automatically for the best possible settings.

Testing Core Location Features

The `CLLocationManager` class offers several class methods that test whether Core Location features are enabled or not. These methods include the following:

- `authorizationStatus` specifies whether users have authorized location services for the application. If the user has agreed to allow the app to use location, this value returns `YES`.

From the Library of Eric Berna

- `locationServicesEnabled` indicates whether location services are enabled on the device at all. Users can enable and disable services entirely from Settings > Location Services or can enable/disable services on an application-by-application basis.
- `significantLocationChangeMonitoringAvailable` lets your app know whether the device can provide low-power/low-accuracy updates that occur when the primary cell tower association changes on the device. Significant location change monitoring offers a great way to roughly track vehicular travel and spot nearby attractions using very little juice.
- `headingAvailable` establishes whether the location manager can provide heading events using an onboard compass.
- `regionMonitoringAvailable` specifies whether region monitoring is available on the current device; `regionMonitoringEnabled` indicates whether the user has enabled or overridden the feature. Region monitoring lets applications define geographical regions to track movement into and out from set areas. Applications can use this to associate actions with real-world location. For example, if you enter a region near your grocery store, applications can notify you to pick up an extra quart of milk.

Always test whether Core Location features are available before attempting to use them. Some of these limitations are device-specific (e.g., you cannot use GPS on first generation iPhones) while others are set by the user. If your user denies access to some of these, your application will be limited to whatever features (if any) remain authorized.

> **Note**
>
> Use the `CLLocationCoordinate2DIsValid()` function to test whether a coordinate provides a valid latitude/longitude pair. The latitude will need to fall between -90 and 90 degrees and its longitude between -180 and 180 degrees.

Recipe: Core Location in a Nutshell

Core Location is easy to use, as demonstrated by the following steps. They walk you through a process of setting up your program to request location data that's representative of normal use. These steps and Recipe 19-1 provide just one example of using Core Location's services, showing how you might pinpoint a user's location:

1. Add the Core Location framework to your project via Project > TARGETS > Build Phases > Link Binaries with Libraries.
2. Check whether the user has enabled Core Location by testing the `CLLocationManager` class's `locationServicesEnabled` class value. Users have the option to switch off Core Location from General > Location Services in the Settings application.

From the Library of Eric Berna

3. Allocate a location manager. Set the manager's delegate to your primary view controller or application delegate. Optionally, set its desired distance filter and accuracy. The distance filter specifies a minimum distance in meters. The device must move at least this distance before it can register a new update. If you set the distance for 5 meters, for example, you will not receive new events until the device has moved that far. If you plan to test by walking, you probably want to reduce that number.

   The accuracy property specifies the degree of precision that you're requesting. To be clear, the location manager does not guarantee any actual accuracy. Setting the requested accuracy asks the manager to (attempt to) retrieve at least that level. When you do not need precision, the manager will deliver its results using whatever technology is most available.

   When you do need precision, the `desiredAccuracy` property informs the manager of that need. You'll find a high level of accuracy especially important for walking and running applications. A lower accuracy level may work for driving in a car or for locating users within large geographical boundaries like cities, states, and countries.

4. Start locating. Tell the location manager to start updating the location. Delegate callbacks let you know when the location has been found. This can take many seconds or up to a minute to occur.

5. Handle the location event delegate callbacks. You'll deal with two types of callbacks: successes that return `CLLocation` data (`locationManager:didUpdateToLocation:fromLocation:`) and failures that do not (`locationManager:didFailWithError:`). Add these delegate methods to your code to catch location updates. In Recipe 19-1, the successful location logs an information overview (`description`) that includes the current latitude and longitude results.

   Depending on your requested accuracy, you may receive three or four location callbacks based on the various location methods used and the requested accuracy, so take this nonlinearity into account.

6. Start moving and wait. Callbacks arrive asynchronously, as location data becomes available. The location information returned to your application includes positioning information along with accuracy measures that you can use to evaluate precision.

When possible, test your Core Location applications on the device and not in the simulator. Although the simulator can now support a variety of location scenarios, on-device testing will provide you with the best results. Deploying Recipe 19-1 to the device allows you to test results as you walk or drive around with your iOS unit.

From the Library of Eric Berna

Recipe 19-1 **Using Core Location to Retrieve Latitude and Longitude**

```
// Logging  utility
- (void) doLog: (NSString *) formatstring, ...
{
    if (!formatstring) return;

    va_list arglist;
    va_start(arglist, formatstring);
        NSString *outstring = [[NSString alloc]
            initWithFormat:formatstring arguments:arglist];
    va_end(arglist);

    if (!log) log = [NSMutableString string];

    NSLog(@"%@", outstring);

    [log appendString:outstring];
    [log appendString:@"\n"];
    textView.text = log;
}

- (void)locationManager:(CLLocationManager *)manager
    didFailWithError:(NSError *)error
{
    [self doLog:@"Location manager error: %@",
        error.localizedFailureReason];
    return;
}

- (void)locationManager:(CLLocationManager *)manager
    didUpdateToLocation:(CLLocation *)newLocation
    fromLocation:(CLLocation *)oldLocation
{
    [self doLog:@"%@\n", [newLocation description]];
}

- (void) loadView
{
    [super loadView];

    // For debugging feedback
    textView = [[UITextView alloc] initWithFrame:CGRectZero];
    textView.font = [UIFont fontWithName:@"Futura"
        size: IS_IPAD ? 24.0f : 12.0f];
    textView.editable = NO;
    [self.view addSubview:textView];
```

From the Library of Eric Berna

```
    // Establish the location manager if possible
    if (!CLLocationManager.locationServicesEnabled)
    {
        [self doLog:@"User has opted out of location services"];
        return;
    }

    locManager = [[CLLocationManager alloc] init];
    locManager.delegate = self;
    locManager.desiredAccuracy = kCLLocationAccuracyBest;
    locManager.distanceFilter = 5.0f; // in meters
    [locManager startUpdatingLocation];

    [self doLog:@"Starting up..."];
}
```

> **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 this book, go to the folder for Chapter 19 and open the project for this recipe.

Location Properties

Each `CLLocation` instance returned by the updated location callback contains a number of properties that describe the device as it travels. Location objects can combine their various properties into a single text result, as used in Recipe 19-1, via the `description` instance method. Alternatively, you can pull out each value on a property-by-property basis. Location properties include the following:

- `altitude`—This property returns the currently detected altitude. It returns a floating-point number in meters above sea level. Speaking as a resident of the "Mile High City," I can assure you the accuracy of this value is minimal at best. Use these results with caution.
- `coordinate`—Recover the device's detected geoposition through the `coordinate` property. A coordinate is a structure with two fields, `latitude` and `longitude`, both of which store a floating-point value. Positive values for latitude lie north of the equator; negative ones south of the equator. Positive longitudes lie east of the meridian; negative longitudes west of the meridian.
- `course`—Use the `course` value to determine the general direction in which the device is heading. This value, which is 0 degrees for North, 90 degrees for East, 180 degrees for South, and 270 degrees for West, roughly approximates the direction of travel. For better accuracy, use headings (`CLHeading` instances) rather than

From the Library of Eric Berna

courses. Headings provide access to magnetic and true North readings via the magnetometer. They are another feature of Core Location and are detailed later in this chapter.

- `horizontalAccuracy`—This property indicates the accuracy (i.e., the uncertainty or measurement error) of the current coordinates. Think of the coordinates that are returned as the center of a circle, and the horizontal accuracy as its radius. The true device location falls somewhere in that circle. The smaller the circle, the more accurate the location. The larger the circle, the less accurate it is. Negative accuracy values indicate a measurement failure.
- `verticalAccuracy`—This property offers an altitude equivalent for horizontal accuracy. It returns the accuracy related to the true value of the altitude, which may (in theory) vary between the altitude minus that amount to the altitude plus that amount. In practice, altitude readings are extremely inaccurate, and the vertical accuracy typically bears little relationship to reality.
- `speed`—In theory, this value returns the speed of the device in meters per second. In practice, this property is best reserved for car travel rather than walking. Recipes later in this chapter demonstrate how this raw property value is used, and that derive velocity independently.
- `timestamp`—This property identifies the time at which the location measurement took place. It returns an `NSDate` instance set to the time when the location was determined by Core Location.

Region Detection

A `CLRegion` object defines a geographical area centered on a given point, with some radius. Regions enable you to register for events whenever a user crosses its boundaries. Create new regions by calling `initCircularRegionWithCenter:radius:identifier:`. You can then pass that region to a location manager instance via `startMonitoringForRegion:desiredAccuracy:`. Call this method for each region you want to monitor. Each identifier must be unique; otherwise, the new region will overwrite any old one.

Upon crossing a region boundary, the location manager delegate receives either the `locationManager:didEnterRegion:` or `locationManager:didExitRegion:` callback. If your app is not running, the system will re-launch it when registered boundaries are crossed, and the `UIApplicationLaunchOptionsLocationKey` will appear in the launch options dictionary.

Recipe: Tracking Speed

The built-in `speed` property returned by each `CLLocation` instance allows you to track the device's velocity over time. Recipe 19-2 highlights its use. When the location manager

From the Library of Eric Berna

callback updates the device's location, the code recovers the speed and logs it. This recipe computes the current speed in miles per hour by multiplying the meters-per-second value by 2.23693629.

The following method sets the desired accuracy to the nearest 10 meters, skipping the distance filtering used by Recipe 19-1. This example is intended for use in a vehicle rather than walking. For walking, running, or biking, you want to use a higher level of accuracy and use a strategy that eliminates inaccurate readings. Recipe 19-2 shows one way to do so.

Recipe 19-2 **Recovering the speed Property from a Location Instance**

```
- (void)locationManager:(CLLocationManager *)manager
    didUpdateToLocation:(CLLocation *)newLocation
    fromLocation:(CLLocation *)oldLocation
{
    // If a speed is detected, log that data in miles per hour
    if (newLocation.speed > 0.0f)
    {
        NSString *speedFeedback = [NSString stringWithFormat:
            @"Speed is %0.1f miles per hour",
            2.23693629 * newLocation.speed];
        NSLog(@"%@", speedFeedback);
    }
}

- (void) viewDidLoad
{
    if (!CLLocationManager.locationServicesEnabled)
    {
        NSLog(@"User has opted out of location services");
        return;
    }

    // Set the delegate and requested accuracy
    locManager = [[CLLocationManager alloc] init];
    locManager.delegate = self;
    locManager.desiredAccuracy =
        kCLLocationAccuracyNearestTenMeters;

    // Start capturing location data
    [self.locManager startUpdatingLocation];
}
```

From the Library of Eric Berna

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 this book, go to the folder for Chapter 19 and open the project for this recipe.

Recipe: Computing Speed and Distance

When moving slowly, or at least more slowly than a car typically moves, you may want to make two specific code accommodations. First, increase your desired accuracy to the highest possible value. Second, ignore the built-in speed property and calculate your speed from scratch. Recipe 19-3 meets these two goals by keeping track of the last-detected most accurate location possible. For purposes of this recipe, "most accurate" is defined as within 100 meters, that is, a likely GPS position.

It uses "accurate" positions to calculate a distance by calling `CLLocation`'s `distanceFromLocation:` method. Dividing the distance by the change in time yields the device's velocity. The method discards values with lower accuracy and values where the device has not moved at least a meter in distance.

For walking and biking, this method produces a reasonably accurate speed while still falling far short of "precise." This is best demonstrated by testing the sample code in the real world with a 3G or later GPS-enabled iPhone or iPad. Should you need to deploy a Core Location–based application with higher accuracy than these samples provide, tweak your sample rates and feedback based on the likely real-world use for the device.

Recipe 19-3 **Deriving Location Information**

```
- (void)locationManager:(CLLocationManager *)manager
    didUpdateToLocation:(CLLocation *)newLocation
    fromLocation:(CLLocation *)oldLocation
{
    if (newLocation.horizontalAccuracy <
        kCLLocationAccuracyHundredMeters) // within 300 feet or so
    {
        // Need a baseline to compute from
        if (lastAccurateLocation)
        {
            // Calculate the change in time and distance
            NSTimeInterval dTime = [newLocation.timestamp
                timeIntervalSinceDate:
                    lastAccurateLocation.timestamp];
            float distance = [newLocation
```

From the Library of Eric Berna

```
                distanceFromLocation:lastAccurateLocation];
            if (distance < 1.0f) return;

            // Sum up the aggregate distance
            aggregateDistance += distance;

            // Report the speed and distance
            NSString *reportString = [NSString stringWithFormat:
                @"Speed: %0.1f miles per hour. %0.1f meters.",
                2.23693629 * distance / dTime, aggregateDistance];
            NSLog(@"%@", reportString);
        }

        // Update the last accurate location
        lastAccurateLocation = newLocation;
    }
}

- (void) loadView
{
    [super loadView];

    // Debugging output
    textView = [[UITextView alloc] initWithFrame:CGRectZero];
    textView.font = [UIFont fontWithName:@"Futura"
        size: IS_IPAD ? 24.0f : 12.0f];
    textView.editable = NO;
    [self.view addSubview:textView];

    if (!CLLocationManager.locationServicesEnabled)
    {
        [self doLog:@"User has opted out of location services"];
        return;
    }

    // Set up the location manager
    locManager = [[CLLocationManager alloc] init];
    locManager.delegate = self;
    locManager.desiredAccuracy = kCLLocationAccuracyBest;
    locManager.distanceFilter = 5.0f; // in meters
    [locManager startUpdatingLocation];

    [self doLog:@"Starting up..."];
}
```

From the Library of Eric Berna

> **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 this book, go to the folder for Chapter 19 and open the project for this recipe.

Recipe: Keeping Track of "North" by Using Heading Values

The onboard location manager can return a computed `course` value that indicates the current direction of travel, that is, North, South, Southeast, and so on. These values take the form of a floating-point number between 0 and 360, with 0 degrees indicating North, 90 degrees being East, and so forth. This computed value is derived from tracking a user's location over time. Newer units have a better way to determine a user's course. Recent devices provide an onboard magnetometer, which can return both magnetic North and true North values.

Not every iOS device supports headings, although at this point, most modern units do. A magnetometer was first released on the iPhone 3GS. Test each device for this ability before subscribing to heading callbacks. If the location manager can generate heading events, the `headingAvailable` property returns `YES`. Use this result to control your `startUpdatingHeading` requests:

```
if (CLLocationManager.headingAvailable)
    [locManager startUpdatingHeading];
```

Cocoa Touch allows you to filter heading callbacks just as you do with distance ones. Set the location manager's `headingFilter` property to a minimal angular change, specified as a floating-point number. For example, if you don't want to receive feedback until the device has rotated at least 5 degrees, set the property to 5.0. All heading values use degrees, between 0.0 and 360.0. To convert a heading value to radians, divide by 180.0 and multiply it by pi.

Heading callbacks return a `CLHeading` object. You can query the heading for two properties: `magneticHeading` and `trueHeading`. The former returns the relative location of magnetic North, the latter true North. True North always points to the geographic North Pole. Magnetic North corresponds to the pole of the Earth's geomagnetic field, which changes over time. The iPhone uses a computed offset (called a *declination*) to determine the difference between these two.

On an enabled device, magnetic heading updates are available even if the user has switched off location updates in the Settings application. What's more, users are not prompted to give permission to use heading data. Magnetic heading information cannot compromise user privacy so it remains freely available to your applications.

You can only use the `trueHeading` property in conjunction with location detection. The iPhone requires a device's location to compute the declination needed to determine

From the Library of Eric Berna

true North. Declinations vary by geoposition. The declination for Los Angeles is different from Perth's, which is different from Moscow's, and London's, and so forth. Some locations cannot use magnetometer readings at all. Certain anomalous regions, like Michipicoten Island in Lake Superior and Grants, New Mexico, offer iron deposits and lava flows that interfere with normal magnetic compass use. Metallic and magnetic sources, such as your computer, car, or refrigerator, may also affect the magnetometer. Several "metal detector" applications in App Store leverage this quirk.

The `headingAccuracy` property provides an error value. This number indicates a plus or minus range that the actual heading falls within. A smaller error bar indicates a more accurate reading. A negative value represents an error in reading the heading.

You can retrieve raw magnetic values along the X, Y, and Z axes using the `x`, `y`, and `z` `CLHeading` properties. These values are measured in microteslas and normalized into a range that Apple states is -128 to 128. (The actual range is more likely to be -128 to 127 based on standard bit math.) Each axis value represents an offset from the magnetic field lines tracked by the device's built-in magnetometer.

Recipe 19-4 uses `CLHeading` data to rotate a small image view with an arrow pointer. The rotation ensures that the arrow always points North. Figure 19-1 shows the interface in action.

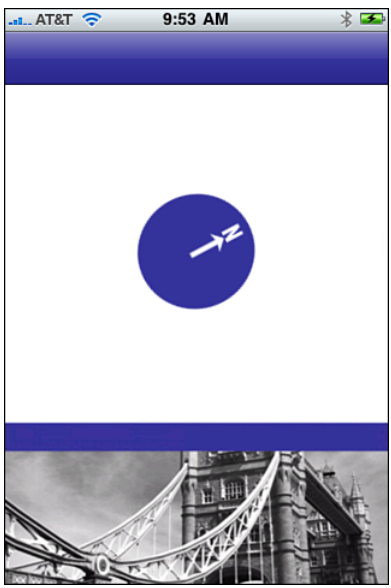


Figure 19-1 The iPhone's built-in magnetometer and the code from Recipe 19-4 ensure that this arrow always points North.

From the Library of Eric Berna

Recipe 19-4 **Detecting the Direction of North**

```
- (void)locationManager:(CLLocationManager *)manager
    didFailWithError:(NSError *)error
{
    NSLog(@"Location manager error: %@",
        error.localizedFailureReason);
}

// Allow calibration
- (BOOL)locationManagerShouldDisplayHeadingCalibration:
    (CLLocationManager *)manager
{
    return YES;
}

// Respond to heading updates
- (void)locationManager:(CLLocationManager *)manager
   didUpdateHeading:(CLHeading *)newHeading
{
    CGFloat heading = -M_PI * newHeading.magneticHeading / 180.0f;
    arrowView.transform = CGAffineTransformMakeRotation(heading);
}

- (void) loadView
{
    [super loadView];

    // Create a view that shows "north"
    arrowView = [[UIImageView alloc]
        initWithImage:[UIImage imageNamed:@"arrow.png"]];
    [self.view addSubview:arrowView];

    // Monitor headings
    locManager = [[CLLocationManager alloc] init];
    locManager.delegate = self;

    if (CLLocationManager.headingAvailable)
        [locManager startUpdatingHeading];
    else
        arrowView.alpha = 0.0f;

}
```

From the Library of Eric Berna

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 this book, go to the folder for Chapter 19 and open the project for this recipe.

Recipe: Forward and Reverse Geocoding

The phrase *reverse geocoding* means transforming latitude and longitude information into human-recognizable address information; *forward geocoding* takes a user-readable address and converts it to latitude and longitude values. MapKit offers both forward and reverse geocoder routines as part of its `CLGeocoder` class.

The `CLPlacemark` class stores all this information. Its `location` property contains the latitude and longitude `coordinate` values. Its real-world properties (`country`, `postalCode`, `locality`, etc.) store the human-readable elements. This one class can be fully populated in either direction—from a real-world address or a coordinate.

Recipe 19-5 demonstrates how to perform both forward and reverse geocoding, using `CLGeocoder`'s block-based encoding approach. The user's device must have network access in order to perform geocoding.

For forward geocoding, supply the address string. For reverse, create a `CLLocation` instance and pass that. The completion block is passed an array of matching placemarks. If that array is nil, an error explains why.

Apple enumerates several rules for using this class. First, only perform one geocoding request at a time. Second, always reuse existing geocoded results when possible, rather than recalculate already known details. For real-world updates, do not send more than one request per minute and do not send any request until the user has moved at least some significant distance. Finally, only perform geocoding when there is a user around to see the results (i.e., when the application is not suspended or running in the background).

The `CLPlacemark` object also offers individual properties with the same information outside the dictionary structure. These properties include the following:

- `subThoroughfare` stores the street number (e.g., the "1600" for 1600 Pennsylvania Avenue).
- `thoroughfare` contains the street name (e.g., Pennsylvania Avenue).
- `sublocality`, when available, refers to the local neighborhood name or a landmark (e.g., White House).
- `subAdministrativeArea` is typically the local county, parish, or other administrative area.
- `locality` stores the city (e.g., Washington, D.C).
- `administrativeArea` corresponds to the state, such as Maryland or Virginia.
- `postalCode` is the postal or zip code (e.g., 20500).

From the Library of Eric Berna

- `country` is self-explanatory, storing the country name, such as the United States.
- `ISOcountryCode` provides an abbreviated country name, like "U.S."

The `addressDictionary` stores an AddressBook framework-style version of the location. Additional properties include `inlandWater` and `ocean`, which describe any local lakes or oceans that are associated with the placemark and, charmingly, `areasOfInterest` local to the placemark, such as national parks or attractions.

Recipe 19-5 **Recovering Address Information from Coordinates and Descriptions**

```
- (void) reverseGeocode: (id) sender
{
    // Starting location
    CLLocation *location = [[CLLocation alloc]
        initWithLatitude:37.33168400
        longitude:-122.03075800];
    CLGeocoder *geocoder = [[CLGeocoder alloc] init];
    [geocoder reverseGeocodeLocation:location
        completionHandler:^(NSArray *placemarks, NSError *error)
     {
         if (!placemarks)
         {
             [self doLog:@"Error retrieving placemarks: %@",
                 error.localizedFailureReason];
             return;
         }

         [self doLog:@"Placemarks from Location: %f, %f",
             location.coordinate.latitude,
             location.coordinate.longitude];

         for (CLPlacemark *placemark in placemarks)
         {
             [self doLog:@"%@", placemark.description];
         }
     }];

}
- (void) geocode: (id) sender
{
    // Retrieve coordinates from an address string
    CLGeocoder *geocoder = [[CLGeocoder alloc] init];
    NSString *address = @"1 Infinite Loop, Cupertino, CA 95014";

    [geocoder geocodeAddressString:address completionHandler:
        ^(NSArray *placemarks, NSError *error)
```

From the Library of Eric Berna

```
    {
        if (!placemarks)
        {
            [self doLog:@"Error retrieving placemarks: %@",
                error.localizedFailureReason];
            return;
        }

        [self doLog:@"Placemarks from Description (%@):",
            address];

        for (CLPlacemark *placemark in placemarks)
        {
            [self doLog:@"%@", placemark.description];
        }
    }];
}

- (void) loadView
{
    [super loadView];
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Reverse", @selector(reverseGeocode:));
    self.navigationItem.leftBarButtonItem =
        BARBUTTON(@"Forward", @selector(geocode:));

    textView = [[UITextView alloc] initWithFrame:CGRectZero];
    textView.font = [UIFont fontWithName:@"Futura"
        size: IS_IPAD ? 24.0f : 12.0f];
    textView.editable = NO;
    [self.view addSubview:textView];
}
```

> **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 this book, go to the folder for Chapter 19 and open the project for this recipe.

Recipe: Viewing a Location

The `MKMapView` class presents users with interactive maps built on the coordinates and scale you provide. The following code snippet sets a map's region to a Core Location coordinate, showing 0.1 degrees of latitude and longitude around 1 Infinite Loop. In the

From the Library of Eric Berna

United States, a region with that range corresponds to the size of a relatively small city or large town, about 5 by 7 miles. Figure 19-2 (left) shows that 0.1 degree-by-0.1 degree range on a map view.

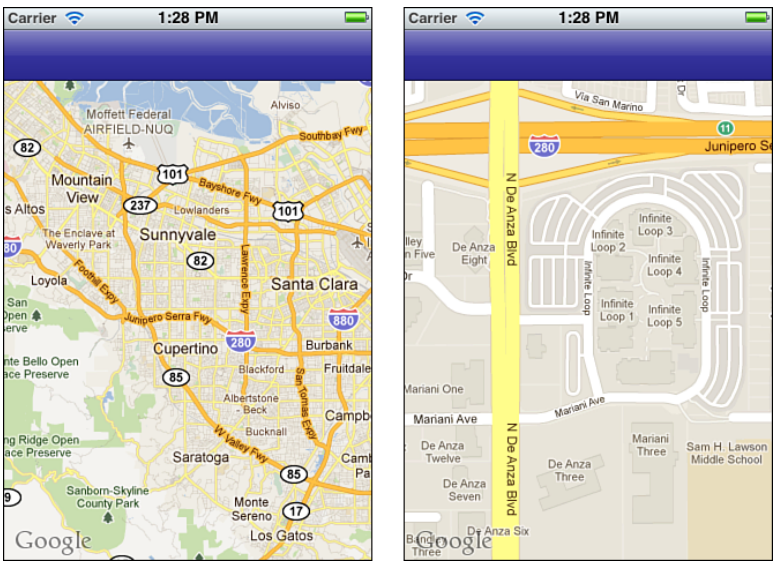


Figure 19-2 A coordinate region of a tenth of a degree latitude by a tenth of a degree longitude covers an area the size of a smallish city or large town, approximately 5 to 7 miles on a side (left). Shrinking that region down to 0.005 degrees on a side produces a street-level display (right).

```
// 1 Infinite Loop
CLLocation *location = [[CLLocation alloc]
    initWithLatitude:37.33168400 longitude:-122.03075800];
mapView.region = MKCoordinateRegionMake(
    location.coordinate, MKCoordinateSpanMake(0.1f, 0.1f));
```

Region size changes occur due to the curvature of the earth. At the equator, one degree of longitude corresponds to about 69 miles (~111 kilometers). This shrinks to 0 at the poles. Latitude is not affected by position. One degree of latitude is always approximately 69 miles (~111 km).

To view map data on a neighborhood level, cut the coordinate span down to 0.01 by 0.01. For a street-by-street level, you can use a smaller span, say, 0.005 degrees latitude by 0.005 degrees longitude. Figure 19-2 (right) shows Infinite Loop at this range.

You can avoid dealing with latitude and longitude degrees and create regions by specifying distance in meters. This snippet sets the view region to a 500-by-500 meter square

From the Library of Eric Berna

around the central coordinate. That roughly approximates the 0.005 by 0.005 degree lat/lon span, showing a street-by-street presentation.

```
mapView.region = MKCoordinateRegionMakeWithDistance(
    location.coordinate, 500.0f, 500.0f);
```

Finding the Best Location Match

Recipe 19-6 performs an on-demand location search using a timed approach. When the user taps the Find Me button, the code starts a 10-second timer. During this search, it attempts to find the best possible location. It uses the horizontal accuracy returned by each location hit to choose and retain the most accurate geoposition. When the time ends, the view controller zooms in its map view, revealing the detected location.

Recipe 19-6 displays the current user location both during and after the search. It does this by setting the `showsUserLocation` property to `YES`. When enabled, this property produces a pulsing pushpin that initially appears at the center of the map view at the device location. That location is detected with Core Location and updates according to user movement. You can test this in Simulator using the Debug > Location options.

Whenever this property is enabled, the map view tasks Core Location with finding the device's current location. So long as this property remains set to `YES`, the map will continue to track and periodically update the user location. A pulsing circle that surrounds the pushpin indicates the most recent search accuracy.

Once the location is set, the Recipe 19-6 permits the user to start interacting with the map. Enabling the `zoomEnabled` property means users can pinch, drag, and otherwise interact with and explore the displayed map. This recipe waits until the full search completes before allowing this interaction, ensuring that the user location remains centered until control returns to the user.

Upon finishing the search, the recipe stops requesting location callbacks by calling `stopUpdatingLocation`. At the same time, it permits the map view to continue tracking the user, leaving the `showsUserLocation` property set to `YES`.

After unsubscribing to updates, the view controller instance sets its location manager delegate to `nil`. This assignment prevents any outstanding callbacks from reaching the controller after the timer finishes. Otherwise, the user and the outstanding callbacks might compete for control of the screen.

Recipe 19-6 **Presenting User Location Within a Map**

```
- (void)locationManager:(CLLocationManager *)manager
    didFailWithError:(NSError *)error
{
    NSLog(@"Location manager error: %@",
        error.localizedFailureReason);
}
```

From the Library of Eric Berna

```
- (void)locationManager:(CLLocationManager *)manager
    didUpdateToLocation:(CLLocation *)newLocation
    fromLocation:(CLLocation *)oldLocation
{
    // Keep track of the best location found
    if (!bestLocation)
        bestLocation = newLocation;
    else if (newLocation.horizontalAccuracy <
        bestLocation.horizontalAccuracy)
        bestLocation = newLocation;

    mapView.region = MKCoordinateRegionMake(bestLocation.coordinate,
        MKCoordinateSpanMake(0.1f, 0.1f));
    mapView.showsUserLocation = YES;
    mapView.zoomEnabled = NO;
}

// Search for n seconds to get the best location during that time
- (void) tick: (NSTimer *) timer
{
    if (++timespent == MAX_TIME)
    {
        // Invalidate the timer
        [timer invalidate];

        // Stop the location task
        [locManager stopUpdatingLocation];
        locManager.delegate = nil;

        // Restore the find me button
        self.navigationItem.rightBarButtonItem =
            BARBUTTON(@"Find Me", @selector(findme));

        if (!bestLocation)
        {
            // no location found
            self.title = @"";
            return;
        }

        // Note the accuracy in the title bar
        self.title = [NSString stringWithFormat:@"%0.1f meters",
            bestLocation.horizontalAccuracy];

        // Update the map and allow user interaction
        [mapView setRegion:
```

From the Library of Eric Berna

```
            MKCoordinateRegionMakeWithDistance(
                bestLocation.coordinate, 500.0f, 500.0f)
            animated:YES];

        mapView.showsUserLocation = YES;
        mapView.zoomEnabled = YES;
    }
    else
        self.title =
            [NSString stringWithFormat:@"%d secs remaining",
                MAX_TIME - timespent];
}

// Perform user-request for location
- (void) findme
{
    // disable right button
    self.navigationItem.rightBarButtonItem = nil;

    // Search for the best location
    timespent = 0;
    bestLocation = nil;
    locManager.delegate = self;
    [locManager startUpdatingLocation];
    [NSTimer scheduledTimerWithTimeInterval:1.0f
        target:self selector:@selector(tick:)
        userInfo:nil repeats:YES];
}

- (void) loadView
{
    [super loadView];

    // Add a map view
    mapView = [[MKMapView alloc] initWithFrame:self.view.bounds];
    [self.view addSubview:mapView];

    if (!CLLocationManager.locationServicesEnabled)
    {
        NSLog(@"User has opted out of location services");
        return;
    }
    else
    {
        // User generally allows location calls
        locManager = [[CLLocationManager alloc] init];
```

From the Library of Eric Berna

```
        locManager.desiredAccuracy = kCLLocationAccuracyBest;
        self.navigationItem.rightBarButtonItem =
            BARBUTTON(@"Find Me", @selector(findme));
    }
}
```

> **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 this book, go to the folder for Chapter 19 and open the project for this recipe.

Recipe: User Location Annotations

Recipe 19-6 provided a way to visually track a location event as it focused over time. Recipe 19-7 kicks this idea up a notch to track a device as it moves over time. Instead of sampling locations over time and picking the best result, it employs a far easier approach while achieving similar results. It hands over all responsibility for user location to the map view and its `userLocation` property, using a timer to check in occasionally and update.

As mentioned in the discussion for Recipe 19-6, enabling the `showsUserLocation` property automatically tasks Core Location to track the device. Recipe 19-7 leverages this capability by checking that location once every few seconds. It updates the map view to reflect that location, keeping the map centered on the user and adding a custom annotation to the user pin to display the current coordinates.

Annotations are pop-up views that attach to locations on the map. They offer a title and a subtitle, which you can set as desired. The `MKUserLocation` class provides direct access to the user location pin and its associated annotation. It offers two readable and writable properties called `title` and `subtitle`. Set these properties as desired. Recipe 19-7 sets the title to "Location Coordinates" and the subtitle to a string containing the latitude and longitude.

The `MKUserLocation` class greatly simplifies annotation editing, but you are limited to working with the map view's user location property. The more general case for annotations proves more complicated. It is detailed in Recipe 19-8, which follows this section.

Recipe 19-7 **Tracking the Device Through the MapView**

```
// Search for n seconds to get the best location during that time
- (void) tick: (NSTimer *) timer
{
    self.title = @"Searching...";
    if (mapView.userLocation)
    {
        // Check for valid coordinate
```

From the Library of Eric Berna

```
        CLLocationCoordinate2D coord =
            mapView.userLocation.location.coordinate;
        if (!coord.latitude && !coord.longitude) return;

        // Update titles
        self.title = @"Found!";
        [mapView setRegion:MKCoordinateRegionMake(coord,
            MKCoordinateSpanMake(0.005f, 0.005f)) animated:NO];
        mapView.userLocation.title = @"Location Coordinates";
        mapView.userLocation.subtitle = [NSString stringWithFormat:
            @"%f, %f", coord.latitude, coord.longitude];
    }
}

- (void) loadView
{
    [super loadView];
    self.view.backgroundColor = [UIColor whiteColor];

    // Add map
    mapView = [[MKMapView alloc] initWithFrame:
        CGRectMake(0.0f, 0.0f, 320.0f, 320.0f)];
    mapView.showsUserLocation = YES;
    mapView.userTrackingMode = MKUserTrackingModeFollowWithHeading;
    [self.view addSubview:mapView];

    if (!CLLocationManager.locationServicesEnabled)
    {
        NSLog(@"User has opted out of location services");
        return;
    }
    else
    {
        [NSTimer scheduledTimerWithTimeInterval:5.0f target:self
            selector:@selector(tick:) userInfo:nil repeats:YES];
    }
}
```

Get This Recipe's Code

To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from this book, go to the folder for Chapter 19 and open the project for this recipe.

From the Library of Eric Berna

Recipe: Creating Map Annotations

Cocoa Touch does not provide a map annotation class. This is surprising, because annotations play such an important role in most map-based applications. Instead, Cocoa Touch defines an `MKAnnotation` protocol. You must design your own classes that conform to this protocol, which demands a `coordinate` property and `title` and `subtitle` instance methods. Listing 19-1 demonstrates how to do this. It builds a simple `MapAnnotation` class, providing the coordinate, title, and subtitle features demanded by the protocol. The extra `tag` integer is not required by the protocol, and is added solely for convenience.

Listing 19-1 **Building a Map Annotation Object**

```
@interface MapAnnotation : NSObject <MKAnnotation>
- (id) initWithCoordinate: (CLLocationCoordinate2D) aCoordinate;
@property (nonatomic, readonly) CLLocationCoordinate2D coordinate;
@property (nonatomic, copy) NSString *title;
@property (nonatomic, copy) NSString *subtitle;
@property (nonatomic, assign) NSUInteger tag;
@end

@implementation MapAnnotation
@synthesize coordinate, title, subtitle;
@synthesize tag;

- (id) initWithCoordinate: (CLLocationCoordinate2D) aCoordinate
{
        if (self = [super init])
        coordinate = aCoordinate;
        return self;
}
@end
```

Creating, Adding, and Removing Annotations

To use annotations, you must create them and add them to a map view. You can do so by adding a single annotation at a time:

```
anAnnotation = [[MapAnnotation alloc] initWithCoordinate:coord];
[mapView addAnnotation:anAnnotation];
```

Alternatively, you can build an array of annotations and add them all at once:

```
[annotations addObject:annotation];
[mapView addAnnotations:annotations];
```

From the Library of Eric Berna

Delete annotations from a map by performing `removeAnnotation:` to remove just one annotation or `removeAnnotations:` to remove all items in an array.

If you need to return a map view to a no-annotations state, remove all its existing annotations. This call recovers the array of existing annotations via the `annotations` property. It then removes these from the map.

```
[mapView removeAnnotations:mapView.annotations];
```

This removes the automated user location and any other programmatic annotations, so to clear just the application, you might want to test each annotation to ensure it's not of the `MKUserLocation` class.

```
- (void) clear
{
    NSArray *annotations =
        [NSArray arrayWithArray:mapView.annotations];
    for (id annotation in annotations)
        if (![annotation isKindOfClass:[MKUserLocation class]])
            [mapView removeAnnotation:annotation];
}
```

Annotation Views

Annotation objects are not views. The `MapAnnotation` class laid out in Listing 19-1 does not create any onscreen elements. It is an abstract class that describes an annotation. It's the map view's job to convert that annotation description into an actual onscreen view. Those views belong to the `MKAnnotationView` class. You can retrieve the annotation view for an existing annotation by querying the map. Supply the annotation and request the matching view.

```
annotationView = [mapView viewForAnnotation:annotation];
```

Nearly all annotation views you'll work with belong to an `MKAnnotationView` subclass, namely `MKPinAnnotationView`. These are the pins that you can drop onto maps. When tapped, they display a callout view.

Customizing Annotation Views

After adding annotations, via `addAnnotation:` or `addAnnotations:`, the map view starts building the annotation views that correspond to those annotations. When it finishes, its delegate, which must declare the `MKMapViewDelegate` protocol, receives a callback. The delegate is notified with `mapView:didAddAnnotationViews:` once the views are built and added to the map. This callback provides your application with an opportunity to customize those annotation views.

From the Library of Eric Berna

An array of annotation views is passed as the second parameter to that callback. You can iterate through this array to set features like the view's `image` or to customize its accessory buttons. Listing 19-2 shows how you might prepare each of these annotation views for use based on their annotations.

Listing 19-2 **Preparing Annotation Views for Use**

```
- (void)mapView:(MKMapView *)mapView
    didAddAnnotationViews:(NSArray *)views
{
    // Initialize each view
    for (MKPinAnnotationView *mkaview in views)
    {
        // Only update custom items
        if (![mkaview isKindOfClass:
            [MKPinAnnotationView class]])
            continue;

        // Set the color to purple
        MapAnnotation *annotation = mkaview.annotation;
        mkaview.pinColor = MKPinAnnotationColorPurple;

        // Add buttons to each one
        UIButton *button = [UIButton
            buttonWithType:UIButtonTypeDetailDisclosure];
        mkaview.rightCalloutAccessoryView = button;
    }
}
```

This example sets a pin color and displays a button, which allows the application to respond to user interactions with annotation views. You are not limited to the built-in annotation protocol, which was minimally satisfied with the class defined in Listing 19-1. Design your annotation class with any instance variables and methods you like for more control over how you query the annotations to prepare your annotation views.

Each annotation view provides direct access to its annotation via its `annotation` property. Use that annotation data to build the exact view you need. Here are some of the annotation view properties you'll want to customize in your MapKit applications.

Each `MKPinAnnotationView` uses a color. You set this color via the `pinColor` property. MapKit provides three color choices: red (`MKPinAnnotationColorRed`), green (`MKPinAnnotationColorGreen`), and purple (`MKPinAnnotationColorPurple`). According to Apple's human-interface guidelines, red pins indicate destination points,

From the Library of Eric Berna

places that the user may want to explore or navigate to. Green pins are starting points, places from which the user can begin a journey. Purple pins are user-specified. When you encourage users to add new data into the map, use purple to indicate that the user has defined them. As you saw in previous recipes, a map view-defined light-blue pin indicates the current user location.

Each annotation view offers two slots, on the left and right of the callout bubble. The `rightCalloutAccessoryView` and `leftCalloutAccessoryView` properties allow you to add buttons or any other custom subview to your callout. Listing 19-2 adds a detail disclosure right callout. You are not limited to buttons, however. You might add image views or other standard Cocoa Touch views as needed.

The `canShowCallout` property controls whether tapping a button produces a callout view. Enabled by default, you can set this property to `NO` if you do not want user taps to open callouts.

You can offset the callouts (normally, they appear directly above the pin in question) by changing the `calloutOffset` property to a new `CGPoint`. You can also change the position for the annotation view itself by adjusting its `centerOffset` property. With pin annotations, the view's art is set by default, but you can create custom annotation art by assigning a `UIImage` to the view's `image` property. Combine custom art with the center offset to produce the exact map look you want.

Responding to Annotation Button Taps

MapKit simplifies button tap management. Whenever you set a callout accessory view property to a control, MapKit takes over the control callback. You do not need to add a target and action. MapKit handles that for you. All you have to do is implement the `mapView:annotationView:calloutAccessoryControlTapped:` delegate callback, as demonstrated in Recipe 19-8.

Recipe 19-8 allows users to add pins to the current map by tapping a Tag button, which sets the pin at the center of the map. Whenever the user adjusts the map, the map view delegate receives a `mapView:regionDidChangeAnimated:` callback. The callback pulls the coordinate of the map center via its `centerCoordinate` property and stores it as the current coordinate.

When the user taps the disclosure button on an accessory view, the application calculates the distance from that pin to the current user location and displays it (in meters) at the top of the screen. Figure 19-3 shows what this might look like, if used near Apple's Infinite Loop headquarters.

From the Library of Eric Berna

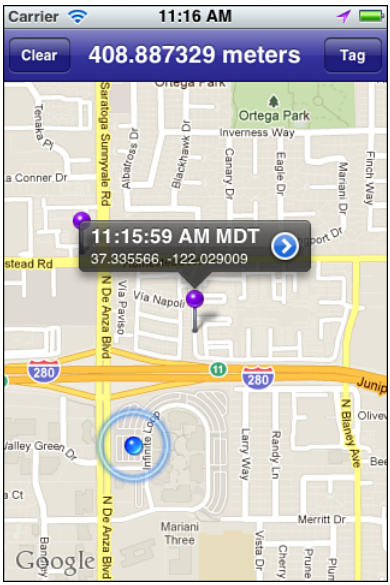


Figure 19-3 These custom annotation views report the time and location when the user sets the pin. Tapping on the accessory view calculates the distance from the pin to the current user location, displaying it at in the title bar.

Recipe 19-8 **Creating an Annotated, Interactive Map**

```
// Update current location when the user interacts with map
- (void)mapView:(MKMapView *)aMapView
    regionDidChangeAnimated:(BOOL)animated
{
    current = [[CLLocation alloc]
        initWithLatitude:mapView.centerCoordinate.latitude
        longitude:mapView.centerCoordinate.longitude];
}

- (void)mapView:(MKMapView *)aMapView
    annotationView:(MKAnnotationView *)view
    calloutAccessoryControlTapped:(UIControl *)control
{
    // Calculate coordinates and distance
    CLLocationCoordinate2D viewCoord = view.annotation.coordinate;
    CLLocation *annotationLocation =
        [[CLLocation alloc] initWithLatitude:viewCoord.latitude
            longitude:viewCoord.longitude];
```

From the Library of Eric Berna

```
    CLLocation *userLocation = mapView.userLocation.location;
    float distance = [userLocation
        distanceFromLocation:annotationLocation];

    // Set the title
    self.title = [NSString stringWithFormat:@"%0f meters", distance];
}

// Set colors and add buttons
- (void)mapView:(MKMapView *)mapView
    didAddAnnotationViews:(NSArray *)views
{
    // Initialize each view
    for (MKPinAnnotationView *mkaview in views)
    {
        if (![mkaview isKindOfClass:[MKPinAnnotationView class]])
            continue;

        // Set the color to purple
        mkaview.pinColor = MKPinAnnotationColorPurple;

        // Add buttons to each one
        UIButton *button = [UIButton buttonWithType:
            UIButtonTypeDetailDisclosure];
        mkaview.rightCalloutAccessoryView = button;
    }
}

- (void) tag
{
    // Create a new annotation
    MapAnnotation *annotation =
       [[MapAnnotation alloc] initWithCoordinate:current.coordinate];

    // Label it with time and place
    NSString *locString = [NSString stringWithFormat:@"%f, %f",
        current.coordinate.latitude, current.coordinate.longitude];
    NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
    formatter.timeStyle = NSDateFormatterLongStyle;
    annotation.title = [formatter stringFromDate:[NSDate date]];
    annotation.subtitle = locString;

    // Add it
    [mapView addAnnotation:annotation];
}
```

From the Library of Eric Berna

```
// Clear all user annotations
- (void) clear
{
    NSArray *annotations = [NSArray arrayWithArray:mapView.annotations];
    for (id annotation in annotations)
        if (![annotation isKindOfClass:[MKUserLocation class]])
            [mapView removeAnnotation:annotation];
}

- (void) loadView
{
    [super loadView];
    self.view.backgroundColor = [UIColor whiteColor];

    // Add map
    mapView = [[MKMapView alloc] initWithFrame:self.view.frame];
    mapView.showsUserLocation = YES;
    mapView.userTrackingMode = MKUserTrackingModeFollowWithHeading;
    [self.view addSubview:mapView];

    if (!CLLocationManager.locationServicesEnabled)
    {
        NSLog(@"User has opted out of location services");
        return;
    }
    else
    {
        mapView.delegate = self;
        self.navigationItem.rightBarButtonItem =
            BARBUTTON(@"Tag", @selector(tag));
        self.navigationItem.leftBarButtonItem =
            BARBUTTON(@"Clear", @selector(clear));
    }
}
```

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 this book, go to the folder for Chapter 19 and open the project for this recipe.

From the Library of Eric Berna

Summary

Core Location and MapKit go hand in hand, offering ways to locate a device's geoposition and present related location information in a coherent map-based presentation. In this chapter, you discovered how to use Core Location to obtain real-time latitude and longitude coordinates and how to reverse geocode those coordinates into real address information. You read about working with speed and course headings both in their native and computed forms. You learned how to set up a map, adjust its region, and add a user location and custom annotations. Here are a few final thoughts for you before you navigate away from this chapter:

- Know your audience and how they will be using your application before deciding how you will approach your location needs. Some Core Location features work better for driving, others for walking and biking.
- Test, test, test, test, test, test. Core Location applications must be exhaustively tested and tuned in the field as well as at Xcode for best results in App Store.
- "Oh, didn't I see you at -104.28393 West today?" Addresses are a lot more meaningful to most people than coordinates. Use reverse geocoding to produce human-readable information, and use forward geocoding to translate from addresses to coordinates.
- Postal codes/zip codes are especially API-friendly. Even if you do not plan to use a map presentation in your application, zip codes are ready for traditional GUI integration. A reverse-geocoded zip code can help retrieve nearby retail information such as addresses, phone numbers, and information about nearby parks and attractions.
- Well-designed annotation views help bring meaningful interactivity into a map view. Don't be afraid to use buttons, images, and other custom elements that expand a map's utility.

From the Library of Eric Berna

This page intentionally left blank

From the Library of Eric Berna

20

Audio, Video, and MediaKit

The iOS device is a media master; its built-in iPod features expertly handle both audio and video. The iOS SDK exposes that functionality to developers. A rich suite of classes simplifies media handling via playback, search, and recording. This chapter introduces recipes that use those classes, presenting media to your users and letting your users interact with that media. You see how to build audio and video viewers, as well as audio and video recorders. You discover how to browse the iPod library and how to choose what items to play. The recipes you're about to encounter provide step-by-step demonstrations showing how to add these media-rich features to your own apps.

Recipe: Playing Audio with AVAudioPlayer

The `AVAudioPlayer` class plays back audio data. Part of the `AVFoundation` framework, it provides a simple-to-use class that offers numerous features, several of which are highlighted in Figure 20-1. With this class, you can load audio, play it, pause it, stop it, monitor average and peak levels, adjust the playback volume, and set and detect the current playback time. All these features are available with little associated development cost. As you are about to see, the `AVAudioPlayer` class provides a solid API.

Initializing an Audio Player

The audio playback features provided by `AVAudioPlayer` take little effort to implement in your code. Apple has provided an uncomplicated class that's streamlined for loading and playing files.

From the Library of Eric Berna

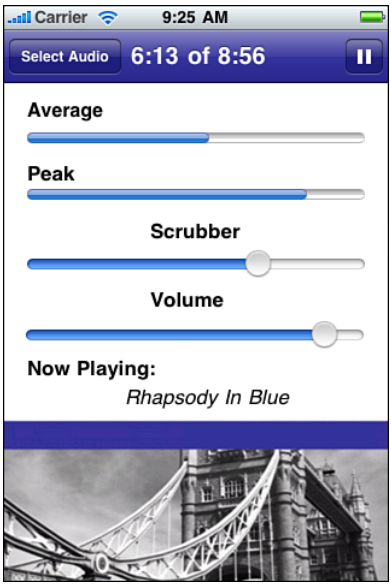


Figure 20-1 The features highlighted in this screenshot were built with a single class, `AVAudioPlayer`. This class provides time monitoring (in the title bar center), sound levels (average and peak), scrubbing and volume sliders, and play/pause control (at the right of the title bar).

To begin, allocate a player and initialize it, either with data or with the contents of a local URL. This snippet uses a file URL to point to an audio file. It reports any error involved in creating and setting up the player. You can also initialize a player with data that's already stored in memory using `initWithData:error:`. That's handy for when you've already read data into memory (such as during an audio chat) rather than reading from a file stored on the device.

```
player = [[AVAudioPlayer alloc] initWithContentsOfURL:
    [NSURL fileURLWithPath:path] error:&error];

if (!player)
{
    NSLog(@"AVAudioPlayer could not be established: %@",
        error.localizedFailureReason);
    return;
}
```

From the Library of Eric Berna

Once you've initialized the player, prepare it for playback. Calling `prepareToPlay` ensures that when you are ready to play the audio, that playback starts as quickly as possible. The call preloads the player's buffers and initializes the audio playback hardware.

```
[player prepareToPlay];
```

Pause playback at any time by calling `pause`. Pausing does not affect the player's `currentTime` property. You can resume playback from that point by calling play again.

Halt playback entirely with `stop`. Stopping playback undoes the buffered setup you initially established with `prepareToPlay`. It does not, however, set the current time back to 0.0; you can pick up from where you left off by calling `play` again, just as you would with `pause`. You may experience starting delays as the player reloads its buffers.

Monitoring Audio Levels

When you intend to monitor audio levels, start by setting the `meteringEnabled` property. Enabling metering lets you check levels as you play back or record audio.

```
player.meteringEnabled = YES;
```

The `AVAudioPlayer` class provides feedback for average and peak power, which you can retrieve on a per-channel basis. Query the player for the number of available channels (via the `numberOfChannels` property), and then request each power level by supplying a channel index. A mono signal uses channel 0, as does the left channel for a stereo recording.

In addition to enabling metering as a whole, you need to call `updateMeters` each time you want to test your levels; this AV player method updates the current meter levels. Once you've done so, use the `peakPowerForChannel:` and `averagePowerForChannel:` methods to read those levels. Recipe 20-6, which appears later in this chapter, shows the details of what's likely going on under the hood in the player when it requests those power levels. You can see that code request the meter levels and then extract either the peak or average power. The `AVAudioPlayer` class hides those details, simplifying access to these values.

The `AVAudioPlayer` measures power in Decibels, which is supplied in floating-point format. Decibels use a logarithmic scale to measure sound intensity. Power values range from 0 dB at the highest to some negative value representing less-than-maximum power. The lower the number (and they are all negative), the weaker the signal will be.

```
int channels = player.numberOfChannels;
[player updateMeters];
for (int i = 0; i < channels; i++)
{
    // Log the peak and average power
    NSLog(@"%d %0.2f %0.2f",
        [player peakPowerForChannel:i],
        [player averagePowerForChannel:i];
}
```

From the Library of Eric Berna

To query the audio player gain (i.e., its "volume"), use the `volume` property. This property returns a floating-point number, here between 0.0 and 1.0, and applies specifically to the player volume rather than the system audio volume. You can set this property as well as read it. This snippet can be used with a target-action pair to update the volume when the user manipulates an onscreen volume slider:

```
- (void) setVolume: (id) sender
{
    // Set the audio player gain to the current slider value
    if (player) player.volume = volumeSlider.value;
}
```

Playback Progress and Scrubbing

Two properties, `currentTime` and `duration`, help monitor the playback progress of your audio. To find the current playback percentage, divide the current time by the total audio duration.

```
progress = player.currentTime / player.duration;
```

To scrub your audio, that is, let your user select the current playback position within the audio track, you may want to pause playback. Although the `AVAudioPlayer` class in current iOS releases can provide audio-based scrubbing hints, these hints can sound jerky. You may want to wait until the scrubbing finishes to begin playback at the new location. Test out live scrubbing using the sample code for this recipe, and you'll hear the relative roughness of this feature.

Make sure to implement at least two target-action pairs if you base your scrubber on a standard `UISlider`. For the first target-action item, mask `UIControlEventTouchDown` with `UIControlEventValueChanged`. These event types allow you to catch the start of a user scrub and whenever the value changes. If you choose to pause on scrub, respond to these events by pausing the audio player and provide some visual feedback for the newly selected time.

```
- (void) scrub: (id) sender
{
    // Pause the player -- optional
    [self.player pause];
    self.navigationItem.leftBarButtonItem =
        SYSBARBUTTON(UIBarButtonSystemItemPlay, self, @selector(play:));

    // Calculate the new current time
    player.currentTime = scrubber.value * player.duration;

    // Update the title with the current time
    self.title = [NSString stringWithFormat:@"%@ of %@",
        [self formatTime:self.player.currentTime],
        [self formatTime:self.player.duration]];
}
```

From the Library of Eric Berna

For the second target-action pair, this mask of three values—`UIControlEvent-TouchUpInside | UIControlEventTouchUpOutside | UIControlEventCancel`—allows you to catch release events and touch interruptions. Upon release, you want to start playing at the new time set by the user's scrubbing.

```
- (void) scrubbingDone: (id) sender
{
    // resume playback here
    [self play:nil];
}
```

Catching the End of Playback

Detect the end of playback by setting the player's delegate and catching the `audioPlayerDidFinishPlaying:successfully:` delegate callback. That method is a great place to clean up any details, like reverting the pause button back to a play button. Apple provides several system bar button items specifically for media playback:

- `UIBarButtonSystemItemPlay`
- `UIBarButtonSystemItemPause`
- `UIBarButtonSystemItemRewind`
- `UIBarButtonSystemItemFastForward`

The rewind and fast forward buttons provide the double-arrowed icons that are normally used to move playback to a previous or next item in a playback queue. You could also use them to revert to the start of a track or progress to its end. Unfortunately, the Stop system item is an X, used for stopping an ongoing load operation and not the standard filled square used on many consumer devices for stopping playback or a recording.

Recipe 20-1 puts all these pieces together to create the unified (albeit horribly designed) interface you saw in Figure 15-1. Here, the user can select audio, start playing it back, pause it, adjust its volume, scrub, and so forth.

The XMAX approach you see here is a bit of a hack. It uses an arbitrary maximum value to estimate the dynamic range of the input levels. Unlike direct Audio Queue calls (that return a float value between 0.0 and 1.0), the decibel levels here have to be approximated to set a progress view value for live feedback. Feel free to adjust the XMAX values to best fit your tests during development.

Recipe 20-1 **Playing Back Audio with AVAudioPlayer**

```
// Set the meters to the current peak and average power
- (void) updateMeters
{
    // Retrieve current values
    [player updateMeters];
```

From the Library of Eric Berna

```
    // Show those values on the two meters
    float avg = -1.0f * [player averagePowerForChannel:0];
    float peak = -1.0f * [player peakPowerForChannel:0];
    meter1.progress = (XMAX - avg) / XMAX;
    meter2.progress = (XMAX - peak) / XMAX;

    // And on the scrubber
    scrubber.value = (player.currentTime / player.duration);

    // Display the current playback progress
    self.title = [NSString stringWithFormat:@"%@ of %@",
        [self formatTime:player.currentTime],
        [self formatTime:player.duration]];
}

// Pause playback
- (void) pause: (id) sender
{
    if (player)
        [player pause];

    // Update the play/pause button
    self.navigationItem.leftBarButtonItem =
        SYSBARBUTTON(UIBarButtonSystemItemPlay, self,
        @selector(play:));

    // Disable interactive elements
    meter1.progress = 0.0f;
    meter2.progress = 0.0f;
    volumeSlider.enabled = NO;
    scrubber.enabled = NO;

    // Stop listening for meter updates
    [timer invalidate];
}

// Start or resume playback
- (void) play: (id) sender
{
    if (player)
        [player play];

    // Enable the interactive elements
    volumeSlider.value = player.volume;
    volumeSlider.enabled = YES;
    scrubber.enabled = YES;
```

From the Library of Eric Berna

```
    // Update the play/pause button
    self.navigationItem.leftBarButtonItem =
        SYSBARBUTTON(UIBarButtonSystemItemPause, self,
        @selector(pause:));

    // Start listening for meter updates
    timer = [NSTimer scheduledTimerWithTimeInterval:0.1f
        target:self selector:@selector(updateMeters)
        userInfo:nil repeats:YES];

}

// Update the volume
- (IBAction) setVolume: (id) sender
{
    if (player) player.volume = volumeSlider.value;
}

// Catch the end of the scrubbing
- (IBAction) scrubbingDone: (id) sender
{
    [self play:nil];
}

// Update playback point during scrubs
- (IBAction) scrub: (id) sender
{
    // Pause the player -- now optional
    // [player pause];
    // self.navigationItem.leftBarButtonItem =
    //   SYSBARBUTTON(UIBarButtonSystemItemPlay, self, @selector(play:));

    // Calculate the new current time
    player.currentTime = scrubber.value * player.duration;

    // Update the title, nav bar
    self.title = [NSString stringWithFormat:@"%@ of %@",
        [self formatTime:player.currentTime],
        [self formatTime:player.duration]];
}

// Prepare but do not play audio
- (BOOL) prepAudio
{
    NSError *error;
    if (![[NSFileManager defaultManager] fileExistsAtPath:path])
        return NO;
```

From the Library of Eric Berna

```
    // Establish player
    player = [[AVAudioPlayer alloc] initWithContentsOfURL:
        [NSURL fileURLWithPath:path] error:&error];
    if (!player)
    {
        NSLog(@"AVAudioPlayer could not be established: %@",
            error.localizedFailureReason);
        return NO;
    }

    [player prepareToPlay];
    player.meteringEnabled = YES;
    player.delegate = self;

    // Initialize GUI
    meter1.progress = 0.0f;
    meter2.progress = 0.0f;
    self.navigationItem.leftBarButtonItem =
        SYSBARBUTTON(UIBarButtonSystemItemPlay, self,
        @selector(play:));
    scrubber.enabled = NO;

    return YES;
}

// On finishing, return to quiescent state
- (void)audioPlayerDidFinishPlaying:(AVAudioPlayer *)player
    successfully:(BOOL)flag
{
    self.navigationItem.leftBarButtonItem = nil;
    scrubber.value = 0.0f;
    scrubber.enabled = NO;
    volumeSlider.enabled = NO;
    [self prepAudio];
}

// Select a media file
- (void) pick: (UIBarButtonItem *) sender
{
    self.navigationItem.rightBarButtonItem.enabled = NO;

    // Each media file is in the public domain via archive.org
    NSArray *choices = [@"Alexander's Ragtime Band*Hello My Baby*\
        Ragtime Echoes*Rhapsody In Blue*A Tisket A Tasket*\
        In the Mood" componentsSeparatedByString:@"*"];
    NSArray *media = [@"ARB-AJ*HMB1936*ragtime*\
```

From the Library of Eric Berna

```
        RhapsodyInBlue*Tisket*InTheMood"
        componentsSeparatedByString:@"*"];

    UIActionSheet *actionSheet = [[UIActionSheet alloc]
        initWithTitle:@"Musical selections"
        delegate:nil
        cancelButtonTitle:IS_IPAD ? nil : @"Cancel"
        destructiveButtonTitle:nil otherButtonTitles:nil];
    for (NSString *choice in choices)
        [actionSheet addButtonWithTitle:choice];

    ModalSheetDelegate *msd = [ModalSheetDelegate
        delegateWithSheet:actionSheet];
    actionSheet.delegate = msd;
    int answer = [msd showFromBarButtonItem:sender animated:YES];
    self.navigationItem.rightBarButtonItem.enabled = YES;

    if (IS_IPAD)
    {
        if (answer == -1) return; // cancel
        if (answer >= choices.count)
            return;
    }
    else
    {
        if (answer == 0) return; // cancel
        if (answer > choices.count)
            return;
        answer--;
    }

    // no action, if already playing
    if ([nowPlaying.text isEqualToString:
        [choices objectAtIndex:answer]])
        return;

    // stop any current item
    if (player)
        [player stop];

    // Load in the new audio and play it
    path = [[NSBundle mainBundle] pathForResource:
        [media objectAtIndex:answer] ofType:@"mp3"];
    nowPlaying.text = [choices objectAtIndex:answer];
    [self prepAudio];
    [self play:nil];
}
```

From the Library of Eric Berna

```
- (void) viewDidLoad
{
    self.view.backgroundColor = [UIColor whiteColor];
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Select Audio", @selector(pick:));

    [self.view addSubview:controlView];
    path= [[NSBundle mainBundle] pathForResource:@"ARB" ofType:@"mp3"];
}
```

> **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 this book, go to the folder for Chapter 20 and open the project for this recipe.

Recipe: Looping Audio

Loops help present ambient background audio without having to respond to delegate callbacks each time the loop finishes a single playthrough. You can use a loop to play an audio snippet several times or play it continuously. Recipe 20-2 demonstrates an audio loop that plays only during the presentation of a particular video controller, providing an aural backdrop for that controller.

You set the number of times an audio plays before the playback ends. A high number (like 999999) essentially provides for an unlimited number of loops. For example, a 4-second loop would take more than 1,000 hours to play back fully with a loop number that high.

```
// Prepare the player and set the loops
[self.player prepareToPlay];
[self.player setNumberOfLoops:999999];
```

Recipe 20-2 uses looped audio for its primary view controller. Whenever its view is onscreen the loop plays in the background. Hopefully you choose a loop that's unobtrusive, sets the mood for your application, and smoothly transitions from the end of playback to the beginning.

This recipe uses a fading effect to introduce and hide the audio. It fades the loop into hearing when the view appears and fades it out when the view disappears. It accomplishes this with a simple approach. A loop iterates through volume levels, from 0.0 to 1.0 on appearing, and 1.0 down to 0.0 on disappearing. A call to `NSThread`'s built-in sleep functionality adds the time delays (a tenth of a second between each volume change) without affecting the audio playback.

From the Library of Eric Berna

Recipe 20-2 **Creating Ambient Audio Through Looping**

```
- (BOOL) prepAudio
{
    // Check for the file.
    NSError *error;
    NSString *path = [[NSBundle mainBundle]
        pathForResource:@"loop" ofType:@"mp3"];
    if (![[NSFileManager defaultManager]
        fileExistsAtPath:path]) return NO;

    // Initialize the player
    player = [[AVAudioPlayer alloc] initWithContentsOfURL:
        [NSURL fileURLWithPath:path] error:&error];
    if (!player)
    {
        NSLog(@"Could not establish AV Player: %@",
            error.localizedFailureReason);
        return NO;
    }

    // Prepare the player and set the loops large
    [player prepareToPlay];
    [player setNumberOfLoops:999999];

    return YES;
}

- (void) viewDidAppear: (BOOL) animated
{
    // Start playing at no-volume
    player.volume = 0.0f;
    [player play];

    // fade in the audio over a second
    for (int i = 1; i <= 10; i++)
    {
        player.volume = i / 10.0f;
        [NSThread sleepUntilDate:
            [NSDate dateWithTimeIntervalSinceNow:0.1f]];
    }

    // Add the push button
    self.navigationItem.rightBarButtonItem.enabled = YES;
}
```

From the Library of Eric Berna

```
- (void) viewWillDisappear: (BOOL) animated
{
    // fade out the audio over a second
    for (int i = 9; i >= 0; i--)
    {
        player.volume = i / 10.0f;
        [NSThread sleepUntilDate:
            [NSDate dateWithTimeIntervalSinceNow:0.1f]];
    }
    [player pause];
}

- (void) push
{
    // Create a simple new view controller
    UIViewController *vc = [[UIViewController alloc] init];
    vc.view.backgroundColor = [UIColor whiteColor];
    vc.title = @"No Sounds";

    // Disable the now-pressed right-button
    self.navigationItem.rightBarButtonItem.enabled = NO;

    // Push the new view controller
    [self.navigationController
        pushViewController:vc animated:YES];
}

- (void) loadView
{
    [super loadView];
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Push", @selector(push));
    self.title = @"Looped Sounds";
    [self prepAudio];
}
```

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 this book, go to the folder for Chapter 20 and open the project for this recipe.

From the Library of Eric Berna

Recipe: Handling Audio Interruptions

When users receive phone calls during audio playback, that audio fades away. The standard answer/decline screen appears. As this happens, `AVAudioPlayer` delegates receive the `audioPlayerBeginInterruption:` callback that is shown in Recipe 20-3. The audio session deactivates, and the player pauses. You cannot restart playback until the interruption ends.

Should the user accept the call, the application terminates, and the application delegate receives an `applicationWillResignActive:` callback. When the call ends, the application relaunches (with an `applicationDidBecomeActive:` callback). If the user declines the call or if the call ends without an answer, the delegate is instead sent `audioPlayerEndInterruption:`. You can resume playback from this method.

If it is vital that playback resumes after accepting a call, and the application needs to relaunch, you can save the current time, as shown in Recipe 20-3. The `viewDidAppear:` method in this recipe, which is called on most kinds of resumptions, checks for a stored interruption value in the user defaults. When it finds one, it uses this to set the current time for resuming playback.

This approach takes into account the fact that the application relaunches rather than resumes after the call finishes. You do not receive the end interruption callback when the user accepts a call.

Recipe 20-3 **Storing the Interruption Time for Later Pickup**

```
- (BOOL) prepAudio
{
    NSError *error;
    NSString *path = [[NSBundle mainBundle]
        pathForResource:@"MeetMeInSt.Louis1904" ofType:@"mp3"];
    if (![[NSFileManager defaultManager]
        fileExistsAtPath:path]) return NO;

    // Initialize the player
    player = [[AVAudioPlayer alloc]
        initWithContentsOfURL:[NSURL fileURLWithPath:path]
        error:&error];
    player.delegate = self;
    if (!player)
    {
        NSLog(@"Could not establish player: %@",
            error.localizedFailureReason);
        return NO;
    }

    [player prepareToPlay];
```

From the Library of Eric Berna

```
    return YES;
}

- (void)audioPlayerDidFinishPlaying:(AVAudioPlayer *)aPlayer
    successfully:(BOOL)flag
{
    // just keep playing
    [player play];
}

- (void)audioPlayerBeginInterruption:(AVAudioPlayer *)aPlayer
{
    // perform any interruption handling here
    fprintf(stderr, "Interruption Detected\n");
    [[NSUserDefaults standardUserDefaults]
        setFloat:[player currentTime] forKey:@"Interruption"];
}

- (void)audioPlayerEndInterruption:(AVAudioPlayer *)aPlayer
{
    // resume playback at the end of the interruption
    fprintf(stderr, "Interruption ended\n");
    [player play];

    // remove the interruption key. it won't be needed
    [[NSUserDefaults standardUserDefaults]
        removeObjectForKey:@"Interruption"];
}

- (void) viewDidAppear:(BOOL)animated
{
    [self prepAudio];
    // Check for previous interruption
    if ([[NSUserDefaults standardUserDefaults]
        objectForKey:@"Interruption"])
    {
        player.currentTime =
            [[NSUserDefaults standardUserDefaults]
                floatForKey:@"Interruption"];
        [[NSUserDefaults standardUserDefaults]
            removeObjectForKey:@"Interruption"];
    }

    // Start playback
    [player play];
}
```

From the Library of Eric Berna

> **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 this book, go to the folder for Chapter 20 and open the project for this recipe.

Audio That Ignores Sleep

Locking an iOS device by pressing the sleep/wake button causes the device to experience the same interruption events that occur with phone calls. When the unit locks, the `AVAudioPlayer` issues an interruption callback. The audio fades away and stops playback. On unlock, the `audioPlayerEndInterruption:` callback triggers and the audio playback continues from where it left off. Try testing Recipe 20-3 by locking and unlocking an iOS device to see this behavior in action.

When you need your audio to continue playing regardless of whether a user locks a phone, respond by updating the current audio session. (You will also see this approach used in Recipe 20-4, which appears in the next section.) Audio sessions set the context for an application's audio, providing direct control over the playback hardware.

To keep playing audio, you need to use a session style that doesn't respond to autolock. For example, you might use a play and record session:

```
if (![[AVAudioSession sharedInstance]
    setCategory:AVAudioSessionCategoryPlayAndRecord error:&error])
{
    // Error establishing the play & record session
    NSLog(@"Could not establish session %@",
        Error.localizedFailureReason);
    return NO;
}
```

Add this snippet to your code before you allocate a new player and, sure enough, your audio will ignore lock events. You can tap the sleep/wake button, causing your iOS device screen to go black. The audio will continue to play.

There's a problem, though. When you use a play and record session, iOS automatically lowers the volume on speaker output. This is by design. Lowering the playback volume avoids feedback loops when a user records audio at the same time as playing audio back. That's great for two-way voice chat, but bad for general playback when you need a full range of audio levels.

Fortunately, there's a workaround that preserves the audio dynamic range while ignoring lock events by calling a low-level C-language audio session function to set the session category. The "media" playback category it uses is not available as a standard `AVAudioSession` constant. That is why you need this alternative approach. Like play and record, a media session ignores sleep/wake button events and continues playback, but unlike play and record, it provides full volume playback.

From the Library of Eric Berna

When initializing the audio session in this manner, you supply a callback function rather than a method. The following code demonstrates this by implementing `interruptionListenerCallback()`, a basic skeleton. Because all interruptions are already caught in the delegate code from Recipe 20-3, this function simply adds a couple of print statements. You may omit those if you want.

When phone calls arrive, the delegate callbacks from Recipe 20-3 handle the interruption and possible relaunch of the application. However, the application never responds to lock/unlock events. You can see this in action by running the sample code and testing for the five primary interruption configurations: call answered, call declined, call ignored, lock, and unlock. By changing the audio session type, those callbacks are no longer generated, and the audio remains unaffected by the sleep/wake button.

```
void interruptionListenerCallback (void *userData,
    UInt32  interruptionState)
{
    if (interruptionState == kAudioSessionBeginInterruption)
        fprintf(stderr, "(ilc) Interruption Detected\n");
    else if (interruptionState == kAudioSessionEndInterruption)
        fprintf(stderr, "(ilc) Interruption ended\n");
}

- (BOOL) prepAudio
{
    NSError *error;
    NSString *path = [[NSBundle mainBundle]
        pathForResource:@"MeetMeInSt.Louis1904" ofType:@"mp3"];
    if (![[NSFileManager defaultManager] fileExistsAtPath:path])
        return NO;

    /* Not this: Audio ends up too low!
    if (![[AVAudioSession sharedInstance]
        setCategory:AVAudioSessionCategoryPlayAndRecord error:&error])
    {
        NSLog(@"Error %@", [error localizedDescription]);
        return NO;
    }
     */


    // Catch interruptions via callback
    AudioSessionInitialize(NULL, NULL,
        interruptionListenerCallback, self);
    AudioSessionSetActive(true);
    UInt32 sessionCategory = kAudioSessionCategory_MediaPlayback;
    AudioSessionSetProperty( kAudioSessionProperty_AudioCategory,
        sizeof(sessionCategory), &sessionCategory);
```

From the Library of Eric Berna

```
    // Initialize the player
    player = [[AVAudioPlayer alloc] initWithContentsOfURL:
        [NSURL fileURLWithPath:path] error:&error];
    player.delegate = self;
    if (!player)
    {
        NSLog(@"Could not establish player: %@",
            error.localizedFailureReason);
        return NO;
    }

    [player prepareToPlay];

    return YES;
}
```

Recipe: Adding Background Audio

iOS offers limited multitasking support. Instead of suspending when the user presses the Home button, a multitasking application may continue execution in the background. To enable this feature, your application declares its `UIBackgroundModes`. Under the current iOS release, these modes are limited to audio (`audio`), location (`location`), and Voice over IP (`voip`) services. Your application can continue processing and producing these elements, even when the application itself is not in the foreground.

You can check if the current device supports multitasking by querying the current UIDevice. This snippet checks the `isMultitaskingSupported` property:

```
if ([UIDevice currentDevice].isMultitaskingSupported)
{
    // do something here...
}
```

Good multitasking citizens are limited. You should shut down networking features, save your application state, release as much memory as possible, and focus on the three core multitasking activities. Do not attempt to update the GUI while your application runs in the background.

To ensure that the audio plays back properly requires a few tweaks to the `AVAudioPlayer`'s session. The `setupAudio` method initializes a new audio session, sets its category to media playback, and ensures that the audio it plays can be mixed with other audio.

Application delegate methods allow you to respond the multitasking events. Recipe 20-4 starts playing back audio upon entering the background and pauses that audio when the application regains primacy.

Recipe 20-4 demonstrates how to programmatically start and stop audio playback upon entering and leaving the background, but it provides a poor example for real-world

From the Library of Eric Berna

best practices. That's because it works completely passively. It will start its audio playback whether the user has requested that audio or not. For example, try locking your test device. If you've been running the sample code, its music will immediately start playing. Adapt this code to provide more meaningful audio backgrounding for your users.

Recipe 20-4 **Background Audio**

```
@interface TestBedViewController : UIViewController
    <AVAudioPlayerDelegate>
{
    AVAudioPlayer *player;
}
@end

@implementation TestBedViewController

- (void) setupAudio
{
    NSString *path = [[NSBundle mainBundle]
        pathForResource:@"InTheMood" ofType:@"mp3"];
    if (![[NSFileManager defaultManager] fileExistsAtPath:path])
        return;

    NSError *error;
    player = [[AVAudioPlayer alloc] initWithContentsOfURL:
        [NSURL fileURLWithPath:path] error:&error];
    if (!player)
    {
        NSLog(@"Could not establish player: %@",
            error.localizedFailureReason);
        return;
    }
    player.delegate = self;
    player.numberOfLoops = 999;

    // Set up a media playback session that mixes its audio
    AudioSessionInitialize(NULL, NULL, NULL,
        (void *) (__bridge CFTypeRef) self);
    AudioSessionSetActive(true);
    UInt32 sessionCategory = kAudioSessionCategory_MediaPlayback;
    AudioSessionSetProperty(
        kAudioSessionProperty_AudioCategory,
        sizeof(sessionCategory), &sessionCategory);
    UInt32 mixWithOthers = 1;
    AudioSessionSetProperty(
        kAudioSessionProperty_OverrideCategoryMixWithOthers,
        sizeof(mixWithOthers), &mixWithOthers);
}
```

From the Library of Eric Berna

```
- (void) play
{
    // start playback
    [player play];
}

- (void)  pause
{
    // pause playback
    [player pause];
}

- (void) loadView
{
    [super loadView];
    self.view.backgroundColor = [UIColor whiteColor];
    [self setupAudio];
    self.title = @"Press the HOME button";
}
@end

#pragma mark -

#pragma mark Application Setup
@interface TestBedAppDelegate : NSObject <UIApplicationDelegate>
{
    UIWindow *window;
    TestBedViewController *tbvc;
}
@end
@implementation TestBedAppDelegate
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    [tbvc play];
}


- (void)applicationWillEnterForeground:(UIApplication *)application
{
    [tbvc pause];
}

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Set up the screen
```

From the Library of Eric Berna

```
    window = [[UIWindow alloc] initWithFrame:
        [[UIScreen mainScreen] bounds]];
    tbvc = [[TestBedViewController alloc] init];
    UINavigationController *nav = [[UINavigationController alloc]
        initWithRootViewController:tbvc];
    window.rootViewController = nav;
    [window makeKeyAndVisible];

    return YES;
}
@end
```

> **Get This Recipe's Code**
>
> To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from this book, go to the folder for Chapter 20 and open the project for this recipe.

Finishing Up Processing

Apple also provides a time-limited mechanism to finish up any current tasks. The time to finish these elements must be finite, and you must supply a way to handle the expiration of that time. Provide a handler, which is called in the event that your task runs out of time before it completes.

```
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    // Establish a way to handle the expiration of your time
    bgStatus = [application beginBackgroundTaskWithExpirationHandler:^
    {
        // Handle expiration
        [application endBackgroundTask:bgTask];
        bgStatus = UIBackgroundTaskInvalid;
    }];

    // Start the long-running task and return immediately.
    dispatch_async(dispatch_get_global_queue(
        DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
    ^{

        // Do the work associated with the task.

        [application endBackgroundTask:bgTask];
        bgStatus = UIBackgroundTaskInvalid;
    });
}
```

From the Library of Eric Berna

Recipe: Recording Audio

The `AVAudioRecorder` class simplifies audio recording in your applications. It provides the same API friendliness as `AVAudioPlayer`, along with similar feedback properties. Together, these two classes leverage development for many standard application audio tasks.

Start your recordings by establishing an `AVAudioSession`. Use a play and record session if you intend to switch between recording and playback in the same application. Otherwise, use a simple record session (via `AVAudioSessionCategoryRecord`). When you have a session, you can check its `inputIsAvailable` property. This property indicates that the current device has access to a microphone.

```
- (BOOL) startAudioSession
{
    // Prepare the audio session
    NSError *error;
    AVAudioSession *session = [AVAudioSession sharedInstance];

    if (![session
        setCategory:AVAudioSessionCategoryPlayAndRecord
        error:&error])
    {
        NSLog(@"Error setting session category: %@",
            error.localizedFailureReason);
        return NO;
    }

    if (![session setActive:YES error:&error])
    {
        NSLog(@"Error activating audio session: %@",
            error.localizedFailureReason);
        return NO;
    }

    return session.inputIsAvailable;
}
```

Recipe 20-5 demonstrates the next step after creating the session. It sets up the recorder and provides methods for pausing, resuming, and stopping the recording.

To start recording, it creates a settings dictionary and populates it with keys and values that describe how the recording should be sampled. This example uses mono Linear PCM sampled 8,000 times a second, which is a fairly low sample rate. Here are a few examples of customizing formats:

From the Library of Eric Berna

- Set `AVNumberOfChannelsKey` to 1 for mono audio, 2 for stereo.
- Audio formats (`AVFormatIDKey`) that work well on iOS include `kAudioFormatLinearPCM` (very large files) and `kAudioFormatAppleIMA4` (compact files).
- Standard `AVSampleRateKey` sampling rates include 8,000; 11,025; 22,050; and 44,100.
- For the linear PCM-only bit depth (`AVLinearPCMBitDepthKey`), use either 16 or 32 bits.

The code allocates a new `AVRecorder` instance and initializes it with both a file URL and the settings dictionary. Once created, this code sets the recorder's delegate and enables metering. Metering for `AVAudioRecorder` instances works like metering for `AVAudioPlayer` instances. You must update the meter before requesting average and peak power levels.

This method uses the same XMAX approach to create an approximate dynamic range for the feedback meters that was shown in Recipe 20-1. Feel free to adjust XMAX to best match the actual dynamic range for your application.

```
- (void) updateMeters
{
    // Show the current power levels
    [self.recorder updateMeters];
    float avg = [self.recorder averagePowerForChannel:0];
    float peak = [self.recorder peakPowerForChannel:0];
    meter1.progress = (XMAX + avg) / XMAX;
    meter2.progress = (XMAX + peak) / XMAX;

    // Update the current recording time
    self.title = [NSString stringWithFormat:@"%
        [self formatTime:self.recorder.currentTime]];
}
```

This code also tracks the recording's `currentTime`. When you pause a recording, the current time stays still until you resume. Basically, the current time indicates the recording duration to date.

When you're ready to proceed with the recording, use `prepareToRecord` and start the recording with `record`. Issue `pause` to take a break in recording; resume again with another call to `record`. The recording picks up where it left off. To finish a recording, use `stop`. This produces a callback to `audioRecorderDidFinishRecording:successfully:`. That's where you can clean up your interface and finalize any recording details.

To retrieve recordings, set the `UIFileSharingEnabled` key to `YES` in the application's Info.plist. This allows your users to access all the recorded material through iTunes' device-specific Apps pane.

From the Library of Eric Berna

Recipe 20-5 **Audio Recording with AVAudioRecorder**

```
- (void)audioPlayerDidFinishPlaying:(AVAudioPlayer *)player
    successfully:(BOOL)flag
{
    // Prepare UI for recording
    self.title = nil;
    meter1.hidden = NO;
    meter2.hidden = NO;
    {
        // Return to play and record session
        NSError *error;
        if (![[AVAudioSession sharedInstance]
            setCategory:AVAudioSessionCategoryPlayAndRecord
            error:&error])
        {
            NSLog(@"Error: %@", error.localizedFailureReason);
            return;
        }
        self.navigationItem.rightBarButtonItem =
            BARBUTTON(@"Record", @selector(record));
    }
}

- (void)audioRecorderDidFinishRecording:(AVAudioRecorder *)aRecorder
    successfully:(BOOL)flag
{
    // Stop monitoring levels, time
    [timer invalidate];
    meter1.progress = 0.0f;
    meter1.hidden = YES;
    meter2.progress = 0.0f;
    meter2.hidden = YES;
    self.navigationItem.leftBarButtonItem = nil;
    self.navigationItem.rightBarButtonItem = nil;

    NSURL *url = recorder.url;
    NSError *error;

    // Start playback
    player = [[AVAudioPlayer alloc] initWithContentsOfURL:url
        error:&error];
    if (!player)
    {
        NSLog(@"Error establishing player for %@: %@",
            url, error.localizedFailureReason);
        return;
    }
    player.delegate = self;
```

From the Library of Eric Berna

```
    // Change audio session for playback
    if (![[AVAudioSession sharedInstance]
        setCategory:AVAudioSessionCategoryPlayback
        error:&error])
    {
        NSLog(@"Error updating audio session: %@",
            error.localizedFailureReason);
        return;
    }

    self.title = @"Playing back recording...";
    [player prepareToPlay];
    [player play];
}

- (void) stopRecording
{
    // This causes the didFinishRecording delegate method to fire
    [recorder stop];
}

- (void) continueRecording
{
    // resume from a paused recording
    [recorder record];
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Done", @selector(stopRecording));
    self.navigationItem.leftBarButtonItem =
        SYSBARBUTTON(UIBarButtonSystemItemPause, self,
        @selector(pauseRecording));
}

- (void) pauseRecording
{
    // pause an ongoing recording
    [recorder pause];
    self.navigationItem.leftBarButtonItem =
        BARBUTTON(@"Continue", @selector(continueRecording));
    self.navigationItem.rightBarButtonItem = nil;
}

- (BOOL) record
{
    NSError *error;

    // Recording settings
```

From the Library of Eric Berna

```
NSMutableDictionary *settings = [NSMutableDictionary dictionary];
[settings setValue:
    [NSNumber numberWithInt:kAudioFormatLinearPCM]
    forKey:AVFormatIDKey];
[settings setValue: [NSNumber numberWithFloat:8000.0]
    forKey:AVSampleRateKey];
[settings setValue: [NSNumber numberWithInt: 1]
    forKey:AVNumberOfChannelsKey]; // mono
[settings setValue: [NSNumber numberWithInt:16]
    forKey:AVLinearPCMBitDepthKey];
[settings setValue: [NSNumber numberWithBool:NO]
    forKey:AVLinearPCMIsBigEndianKey];
[settings setValue: [NSNumber numberWithBool:NO]
    forKey:AVLinearPCMIsFloatKey];

// File URL
NSURL *url = [NSURL fileURLWithPath:FILEPATH];

// Create recorder
recorder = [[AVAudioRecorder alloc] initWithURL:url
    settings:settings error:&error];
if (!recorder)
{
    NSLog(@"Error establishing recorder: %@",
        error.localizedFailureReason);
    return NO;
}

// Initialize degate, metering, etc.
recorder.delegate = self;
recorder.meteringEnabled = YES;
meter1.progress = 0.0f;
meter2.progress = 0.0f;
self.title = @"0:00";

if (![recorder prepareToRecord])
{
    NSLog(@"Error: Prepare to record failed");
    return NO;
}

if (![recorder record])
{
    NSLog(@"Error: Record failed");
    return NO;
}
```

From the Library of Eric Berna

```
    // Set a timer to monitor levels, current time
    timer = [NSTimer scheduledTimerWithTimeInterval:0.1f
        target:self selector:@selector(updateMeters)
        userInfo:nil repeats:YES];

    // Update the navigation bar
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Done", @selector(stopRecording));
    self.navigationItem.leftBarButtonItem =
        SYSBARBUTTON(UIBarButtonSystemItemPause, self,
        @selector(pauseRecording));

    return YES;
}

- (BOOL) startAudioSession
{
    // Prepare the audio session
    NSError *error;
    AVAudioSession *session = [AVAudioSession sharedInstance];

    if (![session setCategory:AVAudioSessionCategoryPlayAndRecord
        error:&error])
    {
        NSLog(@"Error setting session category: %@",
            error.localizedFailureReason);
        return NO;
    }

    if (![session setActive:YES error:&error])
    {
        NSLog(@"Error activating audio session: %@",
            error.localizedFailureReason);
        return NO;
    }

    return session.inputIsAvailable;
}

- (void) viewDidLoad
{
    [self.view addSubview:controlView];

    if ([self startAudioSession])
        self.navigationItem.rightBarButtonItem =
            BARBUTTON(@"Record", @selector(record));
    else
        self.title = @"No Audio Input Available";
}
```

From the Library of Eric Berna

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 this book, go to the folder for Chapter 20 and open the project for this recipe.

Recipe: Recording Audio with Audio Queues

In addition to the `AVAudioPlayer` class, Audio Queues can handle recording and playing tasks in your applications. Audio Queues were needed for recording before the `AVAudioRecorder` class debuted. Using queues directly helps demonstrate what's going on under the hood of the `AVAudioRecorder` class.

Recipe 20-6 records audio at the Audio Queue level, providing a taste of the C-style functions and callbacks used. This code is heavily based on Apple sample code and specifically showcases functionality that is hidden behind the `AVAudioRecorder` wrapper.

The settings used in Recipe 20-6's `setupAudioFormat:` method have been tested and work reliably on the iOS family of devices. It's easy, however, to mess up these parameters when trying to customize your audio quality. If you don't have the parameters set up just right, the queue may fail with little feedback. Google provides copious settings examples.

Recipe 20-6 **Recording with Audio Queues: The Recorder.m Implementation**

```
// Write out current packets as the input buffer is filled
static void HandleInputBuffer (void *aqData,
    AudioQueueRef inAQ, AudioQueueBufferRef inBuffer,
    const AudioTimeStamp *inStartTime,
    UInt32 inNumPackets,
    const AudioStreamPacketDescription *inPacketDesc)
{
    RecordState *pAqData = (RecordState *) aqData;

    if (inNumPackets == 0 &&
        pAqData->dataFormat.mBytesPerPacket != 0)
        inNumPackets = inBuffer->mAudioDataByteSize /
            pAqData->dataFormat.mBytesPerPacket;

    if (AudioFileWritePackets(pAqData->audioFile, NO,
        inBuffer->mAudioDataByteSize, inPacketDesc,
        pAqData->currentPacket, &inNumPackets,
        inBuffer->mAudioData) == noErr)
    {
        pAqData->currentPacket += inNumPackets;
        if (pAqData->recording == 0) return;
        AudioQueueEnqueueBuffer (pAqData->queue, inBuffer,
```

From the Library of Eric Berna

```
            0, NULL);
    }
}

@implementation Recorder

// Set up the recording format as low quality mono AIFF
- (void)setupAudioFormat:(AudioStreamBasicDescription*)format
{
    format->mSampleRate = 8000.0;
    format->mFormatID = kAudioFormatLinearPCM;
    format->mFormatFlags = kLinearPCMFormatFlagIsBigEndian |
        kLinearPCMFormatFlagIsSignedInteger |
        kLinearPCMFormatFlagIsPacked;

    format->mChannelsPerFrame = 1; // mono
    format->mBitsPerChannel = 16;
    format->mFramesPerPacket = 1;
    format->mBytesPerPacket = 2;
    format->mBytesPerFrame = 2;
    format->mReserved = 0;
}

// Begin recording
- (BOOL) startRecording: (NSString *) filePath
{
    // Many of these calls mirror the process for AVAudioRecorder

    // Set up the audio format and the url to record to
    [self setupAudioFormat:&recordState.dataFormat];
    CFURLRef fileURL =  CFURLCreateFromFileSystemRepresentation(
        NULL, (const UInt8 *) [filePath UTF8String],
        [filePath length], NO);
    recordState.currentPacket = 0;

    // Initialize the queue with the format choices
    OSStatus status;
    status = AudioQueueNewInput(&recordState.dataFormat,
        HandleInputBuffer, &recordState,
        CFRunLoopGetCurrent(),kCFRunLoopCommonModes, 0,
        &recordState.queue);
    if (status) {
        fprintf(stderr, "Could not establish new queue\n");
        return NO;
    }
```

From the Library of Eric Berna

```
// Create the output file
status = AudioFileCreateWithURL(fileURL,
    kAudioFileAIFFType, &recordState.dataFormat,
    kAudioFileFlags_EraseFile, &recordState.audioFile);
if (status)
{
    fprintf(stderr, "Could not create file to record audio\n");
    return NO;
}

// Set up the buffers
DeriveBufferSize(recordState.queue, recordState.dataFormat,
    0.5, &recordState.bufferByteSize);
for(int i = 0; i < NUM_BUFFERS; i++)
{
    status = AudioQueueAllocateBuffer(recordState.queue,
        recordState.bufferByteSize, &recordState.buffers[i]);
    if (status) {
        fprintf(stderr, "Error allocating buffer %d\n", i);
        return NO;
    }
    status = AudioQueueEnqueueBuffer(recordState.queue,
        recordState.buffers[i], 0, NULL);
    if (status) {
        fprintf(stderr, "Error enqueuing buffer %d\n", i);
        return NO;
    }
}

// Enable metering
UInt32 enableMetering = YES;
status = AudioQueueSetProperty(recordState.queue,
    kAudioQueueProperty_EnableLevelMetering,
    &enableMetering,sizeof(enableMetering));
if (status)
{
    fprintf(stderr, "Could not enable metering\n");
    return NO;
}

// Start the recording
status = AudioQueueStart(recordState.queue, NULL);
if (status)
{
    fprintf(stderr, "Could not start Audio Queue\n");
    return NO;
}
```

From the Library of Eric Berna

```
    recordState.currentPacket = 0;
    recordState.recording = YES;
    return YES;
}

// Return the average power level
- (CGFloat) averagePower
{
    AudioQueueLevelMeterState state[1];
    UInt32  statesize = sizeof(state);
    OSStatus status;
    status = AudioQueueGetProperty(recordState.queue,
        kAudioQueueProperty_CurrentLevelMeter, &state, &statesize);
    if (status)
    {
        fprintf(stderr, "Error retrieving meter data\n");
        return 0.0f;
    }
    return state[0].mAveragePower;
}

// Return the peak power level
- (CGFloat) peakPower
{
    AudioQueueLevelMeterState state[1];
    UInt32  statesize = sizeof(state);
    OSStatus status;
    status = AudioQueueGetProperty(recordState.queue,
        kAudioQueueProperty_CurrentLevelMeter, &state, &statesize);
    if (status)
    {
        fprintf(stderr, "Error retrieving meter data\n");
        return 0.0f;
    }
    return state[0].mPeakPower;
}


// There's generally about a one-second delay before the
// buffers fully empty
- (void) reallyStopRecording
{
    AudioQueueFlush(recordState.queue);
    AudioQueueStop(recordState.queue, NO);
    recordState.recording = NO;
```

From the Library of Eric Berna

```
    for(int i = 0; i < NUM_BUFFERS; i++)
    {
        AudioQueueFreeBuffer(recordState.queue,
            recordState.buffers[i]);
    }

    AudioQueueDispose(recordState.queue, YES);
    AudioFileClose(recordState.audioFile);
}

// Stop the recording after waiting just a second
- (void) stopRecording
{
    [self performSelector:@selector(reallyStopRecording)
        withObject:NULL afterDelay:1.0f];
}

// Pause after allowing buffers to catch up
- (void) reallyPauseRecording
{
    if (!recordState.queue) {
        fprintf(stderr, "Nothing to pause\n"); return;}
    OSStatus status = AudioQueuePause(recordState.queue);
    if (status)
    {
        fprintf(stderr, "Error pausing audio queue\n");
        return;
    }
}

// Pause the recording after waiting a half second
- (void) pause
{
    [self performSelector:@selector(reallyPauseRecording)
        withObject:NULL afterDelay:0.5f];
}

// Resume recording from a paused queue
- (BOOL) resume
{
    if (!recordState.queue)
    {
        fprintf(stderr, "Nothing to resume\n");
        return NO;
    }
```

From the Library of Eric Berna

```
    OSStatus status = AudioQueueStart(recordState.queue, NULL);
    if (status)
    {
        fprintf(stderr, "Error restarting audio queue\n");
        return NO;
    }

    return YES;
}

// Return the current recording duration
- (NSTimeInterval) currentTime
{
    AudioTimeStamp outTimeStamp;
    OSStatus status = AudioQueueGetCurrentTime (
        recordState.queue, NULL, &outTimeStamp, NULL);
    if (status)
    {
        fprintf(stderr, "Error: Could not retrieve current time\n");
        return 0.0f;
    }

    // 8000 samples per second
    return outTimeStamp.mSampleTime / 8000.0f;
}

// Return whether the recording is active
- (BOOL) isRecording
{
    return recordState.recording;
}
@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 this book, go to the folder for Chapter 20 and open the project for this recipe.

Recipe: Playing Video with the Media Player

The `MPMoviePlayerController` class simplifies video display in your applications. This class, which is part of the MediaPlayer framework, allows you to embed video or to play movies back full screen. The player offers the controls shown in Figure 20-2.

From the Library of Eric Berna

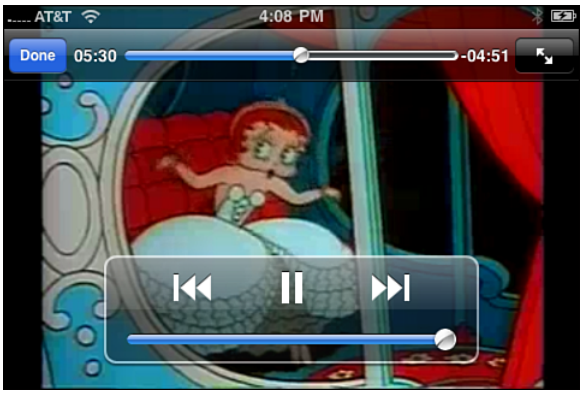


Figure 20-2 The full-screen media player interface offers extensive user control over video playback. This is the same video interface used in the iPod and YouTube applications. (This screenshot is from a public domain Betty Boop cartoon, courtesy of the Internet Archive at archive.org.)

You create a new controller and initialize it with a content URL. The following snippet establishes its controller and sets its control style to full-screen. You can also use the `MPMovieControlStyleEmbedded` style, which adds a button for toggling between embedded and full-screen display modes.

```
movieController = [[MPMoviePlayerController alloc]
    initWithContentURL:[NSURL fileURLWithPath:DEST_PATH]];
movieController.view.frame = self.view.bounds;
movieController.controlStyle = MPMovieControlStyleFullscreen;
[self.view addSubview:movieController.view];
```

Your application can subscribe to the `MPMoviePlayerPlaybackDidFinish-Notification` notification. This notification is sent under two circumstances: when playback naturally ends or when the user taps *Done*.

```
[[NSNotificationCenter defaultCenter]
    addObserver:self selector:@selector(myMovieFinishedCallback:)
    name:MPMoviePlayerPlaybackDidFinishNotification object:nil];
```

To get going, just tell the instance to `play`. Recipe 20-7 includes two methods. One starts playback; the other cleans up when playback ends. This exact code can be used to play back either video or audio. Just provide a source URL pointing to a supported file type. Supported file types include MOV, MP4, MPV, M4V, and 3GP, as well as MP3, AIFF, and M4A, among others.

This code uses an off-device resource via an external URL. Be aware that such connections can be slow (and possibly nonexistent), so prepare for possible lags during playback and always check for connectivity before offering this option in real-world

From the Library of Eric Berna

deployment. Local file URLs produce more reliable playback for video and audio resources.

Unfortunately, the `MPMoviePlayer` object offers limited API control. If you need to loop a movie, you can restart playback after catching the finish notification, as shown here:

```
-(void)myMovieFinishedCallback:(NSNotification*)aNotification
{
    MPMoviePlayerController* controller=[aNotification object];
    [controller play];
}
```

Recipe 20-7 **Playing Back Videos Using the MPMoviePlayer**

```
// Large Movie (35 MB)
#define LARGE_URL @"http://www.archive.org/download/BettyBoopCartoons/\
    Betty_Boop_More_Pep_1936_512kb.mp4"

// Short movie (3 MB)
#define SMALL_URL @"http://www.archive.org/download/\
    Drive-inSaveFreeTv/Drive-in--SaveFreeTv_512kb.mp4"

// Catch the end of playback
-(void)myMovieFinishedCallback:(NSNotification*)aNotification
{
    [[NSNotificationCenter defaultCenter]
        removeObserver:self];
    [movieController.view removeFromSuperview];
    movieController = nil;

    // Restore interface
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Go", @selector(Play));
    self.view.backgroundColor = [UIColor whiteColor];
}

- (void) playMovie
{
    // Disable the interface
    self.navigationItem.rightBarButtonItem = nil;
    seg.enabled = NO;

    // Make the background less noticeable in orientation changes
    self.view.backgroundColor = [UIColor blackColor];

    // Choose which item to play
    NSArray *items = [NSArray arrayWithObjects:
        SMALL_URL, LARGE_URL, nil];
```

From the Library of Eric Berna

```
    NSString *whichItem = [items objectAtIndex:seg.selectedSegmentIndex];
    NSURL *sourceURL = [NSURL URLWithString:whichItem];

    // Create the movie playback controller
    movieController = [[MPMoviePlayerController alloc]
        initWithContentURL:sourceURL];
    movieController.view.frame = [[UIScreen mainScreen] bounds];
    movieController.controlStyle = MPMovieControlStyleFullscreen;
    [self.navigationController.view addSubview:movieController.view];

    // Subscribe to the end notification
    [[NSNotificationCenter defaultCenter]
        addObserver:self selector:@selector(myMovieFinishedCallback:)
        name:MPMoviePlayerPlaybackDidFinishNotification
        object:nil];

    // Start playback
    [movieController play];
}

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

    // Allow user to pick short or long data
    NSArray *items = [@"Short Long" componentsSeparatedByString:@" "];
    seg = [[UISegmentedControl alloc] initWithItems:items];
    seg.selectedSegmentIndex = 0;
    seg.segmentedControlStyle = UISegmentedControlStyleBar;
    self.navigationItem.titleView = seg;
}

// Automatically resize the playback controller
// to the navigation controller's view (including bar)
- (void) viewDidAppear:(BOOL)animated
{
    movieController.view.frame =
        self.navigationController.view.bounds;
}

- (void) didRotateFromInterfaceOrientation:
    (UIInterfaceOrientation)fromInterfaceOrientation
{
    [self viewDidAppear:NO];
}
```

From the Library of Eric Berna

> **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 this book, go to the folder for Chapter 20 and open the project for this recipe.

Recipe: Recording Video

Before you can record video, you must detect whether the device supports camera-based video recording. Checking for an onboard camera, such as those in the first generation and 3G iPhones, is not sufficient. Only the 3GS and newer units provide video recording capabilities. Perform two checks: first, that a camera is available, and second, that the available capture types includes video. The following method performs those checks.

```
- (BOOL) videoRecordingAvailable
{
    // The source type must be available
    if (![UIImagePickerController isSourceTypeAvailable:
        UIImagePickerControllerSourceTypeCamera])
    return NO;

    // And the media type must include the movie type
    NSArray *mediaTypes = [UIImagePickerController
        availableMediaTypesForSourceType:
        UIImagePickerControllerSourceTypeCamera]

    return  [mediaTypes containsObject:@"public.movie"];
}
```

This method searches for `public.movie` in the available types using a constant string. (Refer to Chapter 16, "Documents and Data Sharing," for an in-depth discussion of UTIs.)

As Recipe 20-8 demonstrates, recording video proves to be similar to capturing still images with the onboard camera. You allocate and initialize a new image picker, set its delegate, select whether to allow editing, and present it.

Unlike still image capture, you must set three key properties. First, select a video quality. Recipe 20-8 uses a medium quality, but you can also choose high (`UIImagePickerControllerQualityTypeHigh`), low (`UIImagePickerControllerQualityTypeLow`), or VGA (`UIImagePickerControllerQualityType640x480`). Second, specify a maximum video duration in seconds. Recipe 20-8 allows the user to record up to 30 seconds of video. (The maximum can range up to 10 minutes using the video image picker.) Finally, set the media type array for the picker to a one-object list using the movie media type. You can include `public.image` to provide the user with the option to switch between still and video capture, as highlighted in Figure 20-3.

From the Library of Eric Berna

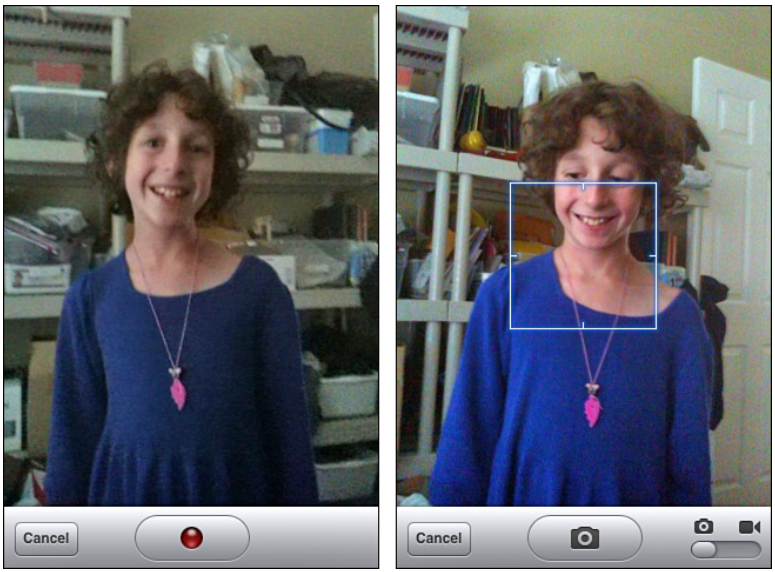


Figure 20-3 Developers may specify one or more media types for the image picker controller. The screenshot on the left demonstrates a video-only capture. The right screenshot shows a picker that supports both image and video capture.

Use the approaches introduced in Chapter 7, "Working with Images," to switch between front and back cameras and to enable or disable the onboard flash lighting if it exists for the current unit.

By setting the `allowsEditing` property to `YES`, you permit users to use the built-in video editor to trim their clips before saving or otherwise working with that data. Whether you allow editing or not, when the user finishes the video capture, the standard image picker callbacks inform the delegate.

With video, you retrieve a URL instead of the actual data. As Recipe 20-8 demonstrates, you should check whether that video is compatible with the built-in album before attempting to save. If it is, you may save it using the `UISaveVideoAtPathToSavedPhotosAlbum()` function. Examples of incompatible clips include MPEG-4 videos downloaded from the Internet Archive (http://archive.org) that you might want to add to your photo album. These files are not album compatible.

Recipe 20-8 **Using UIImagePickerController to Record Video**

```
- (void)video:(NSString *)videoPath
    didFinishSavingWithError:(NSError *)error
    contextInfo:(void *)contextInfo
{
    if (!error)
        self.title = @"Saved!";
    else
```

From the Library of Eric Berna

```
        NSLog(@"Error saving video: %@", error.localizedFailureReason);
}

- (void)imagePickerController:(UIImagePickerController *)picker
    didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    // recover video URL
    NSURL *url =
        [info objectForKey:UIImagePickerControllerMediaURL];

    // check if video is compatible with album
    BOOL compatible =
        UIVideoAtPathIsCompatibleWithSavedPhotosAlbum(url.path);

    // save
    if (compatible)
        UISaveVideoAtPathToSavedPhotosAlbum(url.path,
            self,
            @selector(video:didFinishSavingWithError:contextInfo:),
            NULL);

    if (IS_IPHONE)
    {
        [self dismissModalViewControllerAnimated:YES];
        imagePickerController = nil;
    }
}

- (void) imagePickerControllerDidCancel:
    (UIImagePickerController *) picker
{
    [self dismissModalViewControllerAnimated:YES];
    imagePickerController = nil;
}

// Popover was dismissed
- (void)popoverControllerDidDismissPopover:
    (UIPopoverController *)aPopoverController
{
    imagePickerController = nil;
    popoverController = nil;
}

- (void) recordVideo: (id) sender
{
    if (popoverController)
        return;
```

From the Library of Eric Berna

```
    imagePickerController = [[UIImagePickerController alloc] init];
    imagePickerController.sourceType =
        UIImagePickerControllerSourceTypeCamera;
    imagePickerController.delegate = self;
    imagePickerController.allowsEditing = YES;
    imagePickerController.videoQuality =
        UIImagePickerControllerQualityTypeMedium;
    imagePickerController.videoMaximumDuration = 30.0f; // 30 secs
    imagePickerController.mediaTypes =
        [NSArray arrayWithObject:@"public.movie"];

    // Use this instead to allow the toggle between still and video
    // imagePickerController.mediaTypes =
    //     [NSArray arrayWithObjects:@"public.movie",
    //         @"public.image", nil];

    if (IS_IPHONE)
    {
        [self presentModalViewController:
            imagePickerController animated:YES];
    }
    else
    {
        if (popoverController)
            [popoverController dismissPopoverAnimated:NO];
        popoverController = [[UIPopoverController alloc]
            initWithContentViewController:imagePickerController];
        popoverController.delegate = self;
        [popoverController
             presentPopoverFromBarButtonItem:
                 self.navigationItem.rightBarButtonItem
             permittedArrowDirections:UIPopoverArrowDirectionAny
             animated:YES];
    }
}

- (void) loadView
{
    [super loadView];
    self.view.backgroundColor = [UIColor whiteColor];
    if ([self videoRecordingAvailable])
        self.navigationItem.rightBarButtonItem =
            BARBUTTON(@"Record", @selector(recordVideo:));
    else
        self.title = @"No Video Recording";
}
```

From the Library of Eric Berna

> **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 this book, go to the folder for Chapter 20 and open the project for this recipe.

Recipe: Picking and Editing Video

The `mediaTypes` property used in Recipe 20-8 affects media selection and media capture. To request a picker that presents video assets only, create a photo library picker and use a media array that consists of a single `public.movie` UTI. The following method creates a video-only picker:

```
- (void) pickVideo: (id) sender
{
    imagePickerController =
        [[UIImagePickerController alloc] init];
    imagePickerController.sourceType =
        UIImagePickerControllerSourceTypePhotoLibrary;
    imagePickerController.delegate = self;
    imagePickerController.allowsEditing = NO;
    imagePickerController.mediaTypes =
        [NSArray arrayWithObject:@"public.movie"];

    if (IS_IPHONE)
    {
        [self presentModalViewController:imagePickerController
            animated:YES];
    }
    else
    {
        if (popoverController)
            [popoverController dismissPopoverAnimated:YES];
        popoverController = [[UIPopoverController alloc]
            initWithContentViewController:imagePickerController];
        popoverController.delegate = self;
        [popoverController presentPopoverFromBarButtonItem:
                self.navigationItem.rightBarButtonItem
            permittedArrowDirections:UIPopoverArrowDirectionAny
            animated:YES];
    }
}
```

To edit an already existing video, start by checking that the video asset can be modified, as shown in Recipe 20-9. Call the `UIVideoEditorController` class method `canEditVideoAtPath:`. This returns a Boolean value that indicates whether the video is compatible with the editor controller.

From the Library of Eric Berna

```
if (![UIVideoEditorController canEditVideoAtPath:vpath])
{
    self.title = @"Cannot Edit Video";
    return;
}
```

If it is, you can allocate a new editor, set its `delegate` and `videoPath` properties, and present it. The editor uses a set of delegate callbacks that are similar to but not identical to the ones used by the `UIImagePickerController` class. Callbacks include methods for success, failure, and user cancellation.

When a user has finished editing the video, the controller saves that video to a temporary path and calls `videoEditorController:didSaveEditedVideoToPath:`. That path resides in the application sandbox's tmp folder. If you do nothing with the data, it will be deleted the next time the device reboots. You can, however, save that data either locally into the sandbox's Documents folder or into the shared iOS photo album. To do so, follow the example shown in Recipe 20-8. Regardless of which callback is sent, it's up to you to dismiss the editor and release it.

Recipe 20-9 **Using the Video Editor Controller**

```
// Finished saving to album
- (void)video:(NSString *)videoPath
    didFinishSavingWithError:(NSError *)error
    contextInfo:(void *)contextInfo
{
    if (!error)
        self.title = @"Saved!";
    else
        NSLog(@"Error saving video: %@", error.localizedFailureReason);
}

// Finished saving to temporary trimmed video file
- (void)videoEditorController:(UIVideoEditorController *)editor
    didSaveEditedVideoToPath:(NSString *)editedVideoPath
{
    videoEditorController = nil;
    vpath = nil;

    if (IS_IPHONE)
        [self dismissModalViewControllerAnimated:YES];

    if (popoverController)
    {
        [popoverController dismissPopoverAnimated:NO];
        popoverController = nil;
    }
```

From the Library of Eric Berna

```
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Pick", @selector(pickVideo:));

    // check if video is compatible with album
    BOOL compatible =
        UIVideoAtPathIsCompatibleWithSavedPhotosAlbum(editedVideoPath);

    // save
    if (compatible)
        UISaveVideoAtPathToSavedPhotosAlbum(editedVideoPath,
            self,
            @selector(video:didFinishSavingWithError:contextInfo:),
            NULL);
}

- (void) doEdit
{
    if (![UIVideoEditorController canEditVideoAtPath:vpath])
    {
        self.title = @"Cannot Edit Video";
        return;
    }

    // Can edit
    videoEditorController = [[UIVideoEditorController alloc] init];
    videoEditorController.videoPath = vpath;
    videoEditorController.delegate = self;

    if (IS_IPHONE)
    {
        [self presentModalViewController:videoEditorController
            animated:YES];
    }
    else
    {
        if (popoverController)
            [popoverController dismissPopoverAnimated:NO];
        popoverController = [[UIPopoverController alloc]
            initWithContentViewController:videoEditorController];
        popoverController.delegate = self;
        [popoverController presentPopoverFromBarButtonItem:
                self.navigationItem.rightBarButtonItem
            permittedArrowDirections:UIPopoverArrowDirectionAny
            animated:YES];
    }
}
```

From the Library of Eric Berna

> **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 this book, go to the folder for Chapter 20 and open the project for this recipe.

Recipe: Picking Audio with the MPMediaPickerController

The `MPMediaPickerController` class provides an audio equivalent for the image-picking facilities of the `UIImagePickerController` class. It allows users to choose an item or items from their music library including music, podcasts, and audio books. The iPod-style interface allows users to browse via playlists, artists, songs, albums, and more.

To use this class, allocate a new picker and initialize it with the kinds of media to be used. For audio, you can choose from `MPMediaTypeMusic`, `MPMediaTypePodcast`, `MPMediaTypeAudioBook`, and `MPMediaTypeAnyAudio`. Video types are also available for movies, TV shows, video podcasts, music videos, and iTunesU. These are flags and can be or'ed together to form a mask.

```
MPMediaPickerController *mpc = [[MPMediaPickerController alloc]
    initWithMediaTypes:MPMediaTypeAnyAudio];
mpc.delegate = self;
mpc.prompt = @"Please select an item";
mpc.allowsPickingMultipleItems = NO;
[self presentModalViewController:mpc animated:YES];
```

Set a delegate that conforms to `MPMediaPickerControllerDelegate` and optionally set a prompt. The prompt is text that appears at the top of the media picker, as shown in Figure 20-4. When you choose to allow multiple item selection, the *Cancel* button on the standard picker is replaced by the label *Done*. Normally, the dialog ends when a user taps a track. With multiple selection, users can keep picking items until they press the *Done* button. Selected items are updated to use gray labels.

The `mediaPicker:didPickMediaItems:` delegate callback handles the completion of a user selection. The `MPMediaItemCollection` instance that is passed as a parameter can be enumerated by accessing its `items`. Each item is a member of the `MPMediaItem` class and can be queried for its properties, as shown in Recipe 20-10. Recipe 20-10 uses a media picker to select multiple music tracks. It logs the items the user selected by artist and title.

Consult Apple's `MPMediaItem` class documentation for the available properties for media items, the type they return, and whether they can be used to construct a media property predicate. Building queries and using predicates is discussed in the next section.

From the Library of Eric Berna

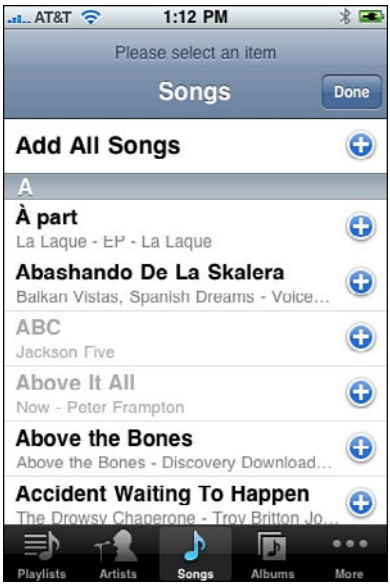


Figure 20-4 In this multiple selection media picker, already selected items appear in gray (see ABC and Above It All). Users tap *Done* when finished. In the normal selection picker, the *Done* button is replaced by *Cancel*, allowing users to leave without selecting an item. An optional prompt field (here, Please Select an Item) appears above the normal picker elements.

Keep in mind that media item properties are not Objective-C properties. You must retrieve them using the `valueForProperty:` method, although it's easy to create a wrapper class that does provide these items as real properties via Objective-C method calls.

Recipe 20-10 **Selecting Music Items from the iPod Library**

```
- (void)mediaPicker: (MPMediaPickerController *)mediaPicker
    didPickMediaItems:(MPMediaItemCollection *)mediaItemCollection
{
    for (MPMediaItem *item in [mediaItemCollection items])
        NSLog(@"[%@] %@",
            [item valueForProperty:MPMediaItemPropertyArtist],
            [item valueForProperty:MPMediaItemPropertyTitle]);

    if (IS_IPHONE)
        [self dismissModalViewControllerAnimated:YES];
    else
    {
        [popoverController dismissPopoverAnimated:YES];
```

From the Library of Eric Berna

```
        popoverController = nil;
    }
}

- (void)mediaPickerDidCancel:(MPMediaPickerController *)mediaPicker
{
    if (IS_IPHONE)
        [self dismissModalViewControllerAnimated:YES];
}

// Popover was dismissed
- (void)popoverControllerDidDismissPopover:
    (UIPopoverController *)aPopoverController
{
    popoverController = nil;
}

// Start the picker session
- (void) action: (UIBarButtonItem *) bbi
{
    MPMediaPickerController *mpc =
        [[MPMediaPickerController alloc]
            initWithMediaTypes:MPMediaTypeMusic];
    mpc.delegate = self;
    mpc.prompt = @"Please select an item";
    mpc.allowsPickingMultipleItems = YES;

    if (IS_IPHONE)
    {
        [self presentModalViewController:mpc animated:YES];
    }
    else
    {
        if (popoverController)
            [popoverController dismissPopoverAnimated:NO];
        popoverController = [[UIPopoverController alloc]
            initWithContentViewController:mpc];
        popoverController.delegate = self;
        [popoverController presentPopoverFromBarButtonItem:
                self.navigationItem.rightBarButtonItem
            permittedArrowDirections:UIPopoverArrowDirectionAny
            animated:YES];
    }
}
```

From the Library of Eric Berna

> **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 this book, go to the folder for Chapter 20 and open the project for this recipe.

Creating a Media Query

Media Queries allow you to filter your iPod library contents, limiting the scope of your search. Table 20-1 lists the nine class methods that `MPMediaQuery` provides for predefined searches. Each query type controls the grouping of the data returned. Each collection is organized as tracks by album, by artist, by audio book, and so on.

Table 20-1 **Query Types**

Class Method	Global?	Filter	Group Type
`albumsQuery`	No	`MPMediaTypeMusic`	`MPMediaGrouping Album`
`artistsQuery`	No	`MPMediaTypeMusic`	`MPMediaGrouping Artist`
`audiobooksQuery` No compilations ➥`Query`	No	`MPMediaTypeAny | MPMediaItemPropertyIs ➥Compilation`	`MPMediaGroupingAlbum`
`composersQuery`	Yes	`MPMediaTypeAny`	`MPMediaGrouping ➥Composer`
`genresQuery`	Yes	`MPMediaTypeAny`	`MPMediaGroupingGenre`
`playlistsQuery`	Yes	`MPMediaTypeAny`	`MPMediaGrouping ➥Playlist`
`podcastsQuery`	No	`MPMediaTypePodcast`	`MPMediaGrouping ➥PodcastTitle`
`songsQuery`	No	`MPMediaTypeMusic`	`MPMediaGroupingTitle`

This approach reflects the way that iTunes works on the desktop. In iTunes, you select a column to organize your results, but you search by entering text into the application's Search field.

Building a Query

Count the number of albums in your library using an album query. This snippet creates that query and then retrieves an array, each item of which represents a single album. These album items are collections of individual media items. A collection may contain a single track or many.

From the Library of Eric Berna

```
MPMediaQuery *query = [MPMediaQuery albumsQuery];
NSArray *collections = query.collections;
NSLog(@"You have %d albums in your library\n", collections.count);
```

Many iOS users have extensive media collections often containing hundreds or thousands of albums, let alone individual tracks. A simple query like this one may take several seconds to run and return a data structure that represents the entire library.

A search using a different query type allows returns collections organized by that type. You can use a similar approach to recover the number of artists, songs, composers, and so on.

Using Predicates

A media property predicate efficiently filters the items returned by a query. For example, you might want to find only those songs whose title matches the phrase "road." The following snippet creates a new songs query and adds a filter predicate to search for that phrase. The predicate is constructed with a value (the search phrase), a property (searching the song title), and a comparison type (in this case, "contains"). Use `MPMediaPredicateComparisonEqualTo` for exact matches and `MPMediaPredicateComparisonContains` for substring matching.

```
MPMediaQuery *query = [MPMediaQuery songsQuery];

// Construct a title comparison predicate
MPMediaPropertyPredicate *mpp = [MPMediaPropertyPredicate
    predicateWithValue:@"road"
    forProperty:MPMediaItemPropertyTitle
    comparisonType:MPMediaPredicateComparisonContains];
[query addFilterPredicate:mpp];

// Recover the collections
NSArray *collections = query.collections;
NSLog(@"You have %d matching tracks in your library\n",
    collections.count);

// Iterate through each item, logging the song and artist
for (MPMediaItemCollection *collection in collections)
{
      for (MPMediaItem *item in [collection items])
      {
            NSString *song = [item valueForProperty:
                MPMediaItemPropertyTitle];
            NSString *artist = [item valueForProperty:
                MPMediaItemPropertyArtist];
            NSLog(@"%@, %@", song, artist);
      }
}
```

From the Library of Eric Berna

> **Note**
>
> If you'd rather use regular predicates with your media collections than media property predicates, I created an `MPMediaItem` properties category (http://github.com/erica/MPMediaItem-Properties). This category allows you to apply standard `NSPredicate` queries against collections, such as those returned by a multiple-item selection picker.

Recipe: Using the MPMusicPlayerController

Cocoa Touch includes a simple-to-use music player class that works seamlessly with media collections. Despite what its name implies, the `MPMusicPlayerController` class is not a view controller. It provides no onscreen elements for playing back music. Instead, it offers an abstract controller that handles playing and pausing music.

It publishes optional notifications when its playback state changes. The class offers two shared instances: the `iPodMusicPlayer` and an `applicationMusicPlayer`. Always use the former. It provides reliable state change feedback, which you will want to catch programmatically.

Initialize the player controller by calling `setQueueWithItemCollection:` with an `MPMediaItemCollection`.

```
[[MPMusicPlayerController iPodMusicPlayer]
    setQueueWithItemCollection: songs];
```

Alternatively, you can load a queue with a media query. For example, you might set a `playlistsQuery` matching a specific playlist phrase, or an artist query to search for songs by a given artist. Use `setQueueWithQuery:` to generate a queue from an `MPMediaQuery` instance.

If you want to shuffle playback, assign a value to the controller's `shuffleMode` property. Choose from `MPMusicShuffleModeDefault`, which respects the user's current setting, `MPMusicShuffleModeOff` (no shuffle), `MPMusicShuffleModeSongs` (song-by-song shuffle), and `MPMusicShuffleModeAlbums` (album-by-album shuffle). A similar set of options exists for the music's `repeatMode`.

Once you set the item collection, you can play, pause, skip to the next item in the queue, go back to a previous item, and so forth. To rewind without moving back to a previous item, issue `skipToBeginning`. You can also seek within the currently playing item, moving the playback point forward or backward.

Recipe 20-11 offers a simple media player that shows the currently playing song (along with its artwork, if available). When run, the user selects a group of items using an `MPMediaPickerController`. This item collection is returned and assigned to the player, which begins playing back the group.

A pair of observers use the default notification center to watch for two key changes: when the current item changes and when the playback state changes. To catch these changes, you must manually request notifications. This allows you to update the interface with new "now playing" information when the playback item changes.

From the Library of Eric Berna

```
[[MPMusicPlayerController iPodMusicPlayer]
    beginGeneratingPlaybackNotifications];
```

You may undo this request by issuing `endGeneratingPlaybackNotifications`, or you can simply allow the program to tear down all observers when the application naturally suspends or terminates. Note that, because this recipe uses iPod music player, playback continues after leaving the application unless you specifically stop it. Playback is not affected by the application suspension or teardown.

```
- (void) applicationWillResignActive: (UIApplication *) application
{
    // Stop player when the application quits
    [[MPMusicPlayerController iPodMusicPlayer] stop];
}
```

In addition to demonstrating playback control, Recipe 20-11 shows how to display album art during playback. It uses the same kind of `MPItem` property retrieval used in previous recipes. In this case, it queries for `MPMediaItemPropertyArtwork` and, if artwork is found, it uses the `MPMediaItemArtwork` class to convert that artwork to an image of a given size.

Recipe 20-11 **Simple Media Playback with the iPod Music Player**

```
#define PLAYER [MPMusicPlayerController iPodMusicPlayer]

#pragma mark PLAYBACK
- (void) pause
{
    // Pause playback
    [PLAYER pause];
    toolbar.items = [self playItems];
}

- (void) play
{
    // Restart play
    [PLAYER play];
    toolbar.items = [self pauseItems];
}

- (void) fastforward
{
    // Skip to the next item
    [PLAYER skipToNextItem];
}

- (void) rewind
{
```

From the Library of Eric Berna

```
    // Skip to the previous item
    [PLAYER skipToPreviousItem];
}


#pragma mark STATE CHANGES
- (void) playbackItemChanged: (NSNotification *) notification
{
    // Update title and artwork
    self.title = [PLAYER.nowPlayingItem
        valueForProperty:MPMediaItemPropertyTitle];
    MPMediaItemArtwork *artwork = [PLAYER.nowPlayingItem
        valueForProperty: MPMediaItemPropertyArtwork];
    imageView.image = [artwork imageWithSize:[imageView frame].size];
}

- (void) playbackStateChanged: (NSNotification *) notification
{
    // On stop, clear title, toolbar, artwork
    if (PLAYER.playbackState == MPMusicPlaybackStateStopped)
    {
        self.title = nil;
        toolbar.items = nil;
        imageView.image = nil;
    }
}

#pragma mark MEDIA PICKING
- (void)mediaPicker: (MPMediaPickerController *)mediaPicker
    didPickMediaItems:(MPMediaItemCollection *)mediaItemCollection
{
    // Set the songs to the collection selected by the user
    songs = mediaItemCollection;

    // Update the playback queue
    [PLAYER setQueueWithItemCollection:songs];

    // Display the play items in the toolbar
    [toolbar setItems:[self playItems]];

    // Clean up the picker
    [self dismissModalViewControllerAnimated:YES];
    [mediaPicker release];
}
```

From the Library of Eric Berna

```
- (void)mediaPickerDidCancel:(MPMediaPickerController *)mediaPicker
{
    // User has canceled
    [self dismissModalViewControllerAnimated:YES];
    [mediaPicker release];
}

- (void) pick: (UIBarButtonItem *) bbi
{
    // Select the songs for the playback queue
    MPMediaPickerController *mpc = [[MPMediaPickerController alloc]
        initWithMediaTypes:MPMediaTypeMusic];
    mpc.delegate = self;
    mpc.prompt = @"Please select items to play";
    mpc.allowsPickingMultipleItems = YES;

    if (IS_IPHONE)
    {
        [self presentModalViewController:mpc animated:YES];
    }
    else
    {
        if (popoverController)
            [popoverController dismissPopoverAnimated:NO];
        popoverController = [[UIPopoverController alloc]
            initWithContentViewController:mpc];
        popoverController.delegate = self;
        [popoverController presentPopoverFromBarButtonItem:
                self.navigationItem.rightBarButtonItem
            permittedArrowDirections:UIPopoverArrowDirectionAny
            animated:YES];
    }
}

#pragma mark INIT VIEW
- (void) viewDidLoad
{
    self.navigationItem.rightBarButtonItem = BARBUTTON(@"Pick",
        @selector(pick));

    // Stop any ongoing music
    [PLAYER stop];

    // Add observers for state and item changes
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(playbackStateChanged)
```

From the Library of Eric Berna

```
        name:MPMusicPlayerControllerPlaybackStateDidChangeNotification
        object:PLAYER];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(playbackItemChanged)
        name:MPMusicPlayerControllerNowPlayingItemDidChangeNotification
        object:PLAYER];
    [PLAYER beginGeneratingPlaybackNotifications];
}
@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 this book, go to the folder for Chapter 20 and open the project for this recipe.

Summary

This chapter introduced many ways to handle audio and video media, including playback and recording. You saw recipes that worked with high-level Objective-C classes, and those that worked with lower-level C functions. You discovered local and remote data sources and read about media pickers, controllers, and more. Here are some final thoughts to take from this chapter:

- Apple is still in the process of building its AV media playback classes, even in the current iOS 5 release. They are becoming more and more stunningly powerful over time. Because of time and space limitations, this chapter didn't address many of the technologies that power audio on iOS: AVFoundation, OpenAL, Core Audio, Audio Units, or Core MIDI. These topics are huge, and each is worthy of its own book.
- This chapter also did not discuss `ALAssetsLibrary`, although examples of using this class are covered in Chapter 7. You may want to look into the `MPMediaItemPropertyAssetURL` and `AVAssetExportSession` classes for working with audio library assets.
- When you plan to work with video, be aware of bandwidth limitations. Local phone carriers may not appreciate you burdening their networks with excessive data transfer, and that burden may affect your ability to place your application in App Store.
- Audio Queue provides powerful low-level audio routines, but they're not for the faint of heart or for anyone who just wants a quick solution. If you need the kind of fine-grained audio control that Audio Queues bring, Apple supplies extensive documentation on achieving your goals.

From the Library of Eric Berna

- When creating local assets using recording features, make sure to enable `UIFileSharingEnabled` in your Info.plist to allow users to access and manage the files they create through iTunes.
- The `MPMusicPlayerController` class introduces a simple way to interact with music from your onboard iTunes library. Be sure to master both `AVAudioPlayer` use for local data files and `MPMusicPlayerController` for iTunes.

From the Library of Eric Berna

This page intentionally left blank

From the Library of Eric Berna

21

Push Notifications

When Web services need to communicate directly with users, push notifications provide a powerful solution. They deliver messages directly to the device screen via a special Apple service. Push notifications lets devices display an alert, play a custom sound, or update an application badge. In this way, off-device services connect with an iOS-based client, letting them know about new data or updates. Unlike most other iOS development arenas, nearly all the push story takes place off the device. Developers must create Web-based services to manage and deploy these updates. In this chapter, you learn how push notifications work and dive into the details needed to create your own push-based system.

Introducing Push Notifications

Push notifications, also called **remote notifications**, refer to a kind of message sent to iOS devices by an outside service. These push-based services work with any kind of application that normally checks for information updates. For example, a service might poll for new updates on Facebook, Twitter, or Google Plus, scan for nearby dangerous weather systems, respond to sensors for in-home security systems, or invite you to a game match. When new information becomes available for a client, the service pushes that update through Apple's remote notification system. The notification transmits directly to the device, which has registered to receive those updates.

The key to push is that these messages originate from outside the device itself. They are part of a client-server paradigm that lets Web-based server components communicate with iOS clients through an Apple-supplied service. With push, developers can send nearly instant updates to iOS devices that don't rely on users launching a particular application. Instead, processing occurs on the server side of things. When push messages arrive, the iOS client can respond by displaying a badge, playing a sound, and/or showing an alert box.

Moving application logic to a server limits client-side complexity, allowing many clients to share a single update system. Consider the weather danger service just mentioned. A dedicated site can monitor NOAA and other atmospheric feeds, sending remote

From the Library of Eric Berna

notifications to each subscribed client as needed. Offsite processing also provides energy savings for iOS-based applications. They can now rely on push rather than using the iOS device's local CPU resources to monitor and react to important information changes.

Push's reason for being is not only tied into local resources. It also offers a valuable solution for communicating with Web-based services that goes beyond poll-and-update applications. For example, push might allow you to hook into a recommendation service that produces restaurant suggestions even when an application isn't running or to a gaming service that sends you reminder notices about an upcoming tournament. So don't think about push solely as a battery saver. Also think about it as a conduit for Web services as well.

From social networking to monitoring RSS feeds, push lets iOS users keep on top of asynchronous data feeds. It offers a powerful solution for connecting iOS clients to Web-based systems of all kinds. With push, the services you write can connect to your installed iOS base and communicate updates in a clean, functional manner.

How Push Works

Push notifications aren't just a general way to talk directly to iOS devices at will. They are tied to specific applications and require several security checks. A push server can only communicate with those iOS devices that host its application, that are online, and that have opted to receive remote messages. Users have the ultimate say in push updates. They can allow or disallow that kind of communication, and a well-written application lets users opt-in and opt-out of the service at will.

The chain of communication between server and client works like this. Push providers deliver message requests through a central Apple server and via that server to their iOS clients. In normal use, the server triggers on some event (like new mail or an upcoming appointment) and generates notification data aimed at a specific iOS device. It sends this message request to the Apple Push Notification Service (APNS). This notification uses JSON formatting and is limited to 256 bytes each, so the information that can be pushed through on that message is quite limited. This formatting and size ensures that APNS limits its bandwidth to the tightest possible configuration.

APNS offers a centralized system that negotiates communication with iOS devices in the real world. It passes the message through to the designated device. A handler on the iOS device decides how to process the message. As Figure 21-1 shows, push providers talk to APNS, sending their message requests, and APNS talks to iOS devices, relaying those messages to handlers on the unit.

From the Library of Eric Berna

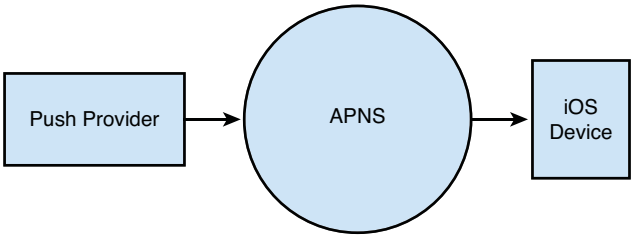


Figure 21-1 Providers send messages through Apple's centralized push notification service to communicate with an iOS device.

Multiple Provider Support

APNS was built to support multiple provider connections, allowing many services to communicate with it at once. It offers multiple gateways into the service so that each push service does not have to wait for availability before sending its message. Figure 21-2 illustrates the many-to-many relationship between providers and iOS devices. APNS allows providers to connect at once through multiple gateways. Each provider can push messages to many different devices.

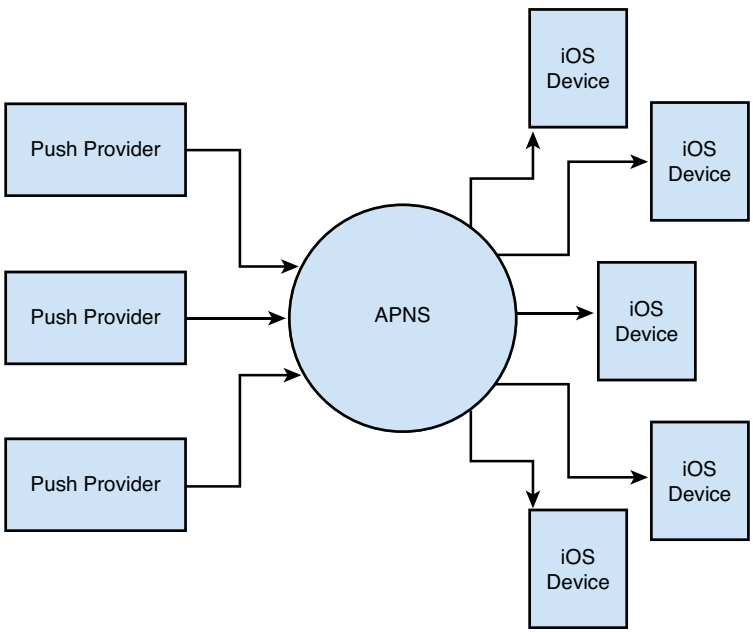


Figure 21-2 Apple's Push Notification Service offers many gateways on its provider-facing side, allowing multiple providers to connect in parallel. Each push provider may connect to any number of iOS devices.

From the Library of Eric Berna

Security

Security is a primary component of remote notifications. The push provider must sign up for a secure sockets layer certificate for each application it works with. Services cannot communicate with APNS unless they authenticate themselves with this certificate. They must also provide a unique key called a token that identifies both the device to message and the application to notify.

After receiving an authenticated message and device token, APNS contacts the device in question. Each iOS device must be online in some way to receive a notification. They can be connected to a cellular data network or to a Wi-Fi hotspot. APNS establishes a connection with the device and relays the notification request. If the device is offline and the APNS server cannot make a connection, the notification is queued for later delivery.

Upon receiving the request, iOS performs a number of checks. Push requests are ignored when the user disables push updates for a given application; users can do so in the Settings application on their device. When updates are allowed, and only then, the iOS device determines whether the client application is currently running. If so, it sends a message directly to the running application via the application delegate. If not, it performs some kind of alert, whether displaying text, playing a sound, or updating a badge.

When an alert displays, users typically have the option to close the alert or tap View. If they choose View, the iOS device launches the application in question and sends it the notification message that it would have received while running. If the user taps Close, the notification gets ignored and the application does not launch.

This pathway, from server to APNS to device to application, forms the core flow of push notifications. Each stage moves the message along the way. Although the multiple steps may sound extensive, in real life, the notification arrives almost instantaneously. Once you set up your certificates, identifiers, and connections, the actual delivery of information becomes trivial. Nearly all the work lies in first setting up that chain and then in producing the information you want to deliver.

Make sure you treat all application certificates and device tokens as sensitive information. When storing these items on your server, you must ensure that they are not generally accessible. Should this information hit the wild, it could be exploited by third parties. This would likely result in Apple revoking your SSL push certificate. This would disable all remote notifications for any apps you have sold and might force you to pull the application from the store.

Push Limitations

In real-world scenarios, push notifications are not as reliable as you might wish. They can be fairly flaky. Apple does not guarantee the delivery of each notification or the order in which notifications arrive. Never send vital information by push. Reserve this feature for helpful notifications that update and inform the user, but that the user can miss without serious consequence.

Items in the push delivery queue may be displaced by new notifications. That means that notifications may compete and may get lost along the way. Although Apple's feedback

From the Library of Eric Berna

service reports failed deliveries (i.e., messages that cannot be properly sent through the push service, specifically to applications that have been removed from a device), you cannot retrieve information regarding bumped notifications. From the APN service point of view, a lost message was still successfully "delivered."

Push Notifications Versus Local Notifications

iOS provides both remote push notification and local notification support. Local notifications, which are discussed and demonstrated in Chapter 13, "Alerting the User," are created and scheduled on-device. Push notification originate from a remote server, and are sent to the device through the Apple push notification service (APNS). This chapter exclusively covers remote notifications. Refer to Chapter 13 to learn more about local notifications.

Provisioning Push

To start push development, you must visit Apple's iOS developer Provisioning Portal. This portal is located at http://developer.apple.com/ios/manage/overview/index.action. Sign in with your iOS developer credentials to gain access to the site. Here at the portal, you can work through the steps needed to create a new application identifier that can be associated with a push service.

There's a fair amount of detail involved. Make sure you hit every point. The following sections walk you through the process. You see how to create a new identifier, generate a certificate, and request a special provisioning profile so you can build push-enabled applications. Without a push-enabled profile, your application will not be able to receive remote notifications.

Generate a New Application Identifier

At the developer portal, click App IDs. You'll find this option in the column on the left side of the Web page. This opens a page that allows you to create new application identifiers. Each push service is based on a single identifier, which you must create and then set to allow remote notification. You cannot use a wildcard identifier with push applications; every push-enabled app demands a unique identifier.

In the App IDs section, click New App ID; this button appears at the top-right of the Web page. Once clicked, the site opens a new Create App ID page, similar to Figure 21-3. Enter a name that describes your new identifier, such as "Tutorial Push Application" and a new bundle identifier.

These IDs typically use reverse domain patterns, like com.*domainname*.*appname*, such as com.sadun.pushtutorial. The identifier must be unique and may not conflict with any other registered application identifier in Apple's system. Leave the bundle seed set to your team identifier unless you have some compelling reason to do otherwise.

From the Library of Eric Berna

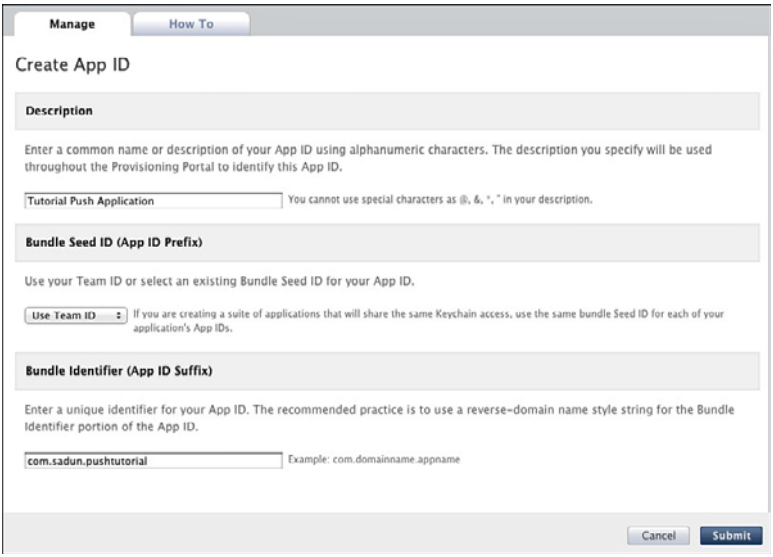


Figure 21-3 Create a new application identifier at the iOS provisioning portal.

Click Submit to add the new identifier. This adds the app ID irrevocably to Apple's system, where it is now registered to you. You return to the App ID page with its list of identifiers and are now ready to establish that identifier as push compliant.

> **Note**
>
> Apple does not provide any way to remove an application identifier from the program portal once it has been created.

Generate Your SSL Certificate

On the App ID page, you can see which identifiers work with push and which do not. The Apple Push Notification column shows whether push has been enabled for each app ID. The three states for this column are

- Unavailable (gray) for IDs that are no longer available
- Configurable (yellow) for apps that can be used with push but that haven't yet been set up to do so
- Enabled (green) for apps that are ready for push

In the push column, you'll find two dots for each application identifier—one for Development and another for Production (see Figure 21-4). These options are configured separately. Locate your new app ID, make sure the yellow Configurable for Development is shown, and click Configure. This option appears in the rightmost column. When clicked,

From the Library of Eric Berna

the browser opens a new Configure App ID page that permits you to associate your identifier with the push notification service.



Figure 21-4 Create a new application identifier at the iOS provisioning portal.

An Enable Push Notification Services check box appears about halfway down the page (see Figure 21-5). Check this box to start the certificate creation process. Once checked, the two Configure buttons on the right side of the page become enabled. Click that button. A page of instructions loads, showing you how to proceed. It guides you through creating a secure certificate that will be used by your server to sign messages it sends to the APNS.

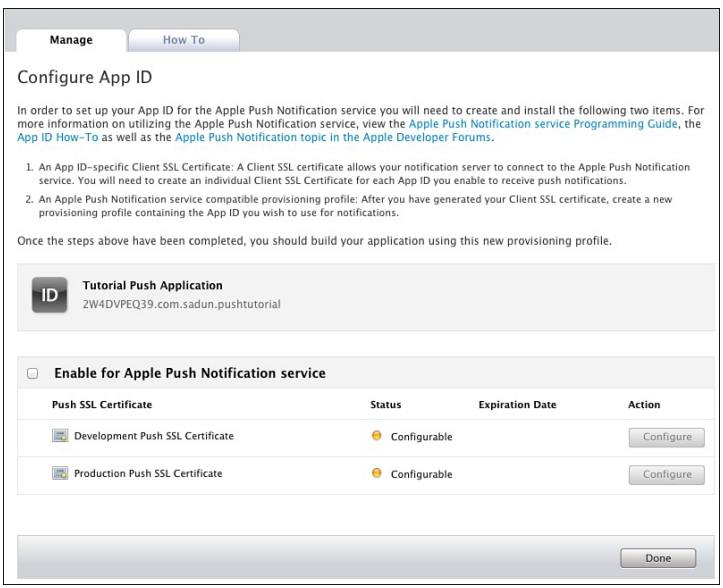


Figure 21-5 Create a new certificate request even though you've probably already done so in the past for your developer and distribution certificates.

As instructed, launch the Keychain Access application. This application is located on your Macintosh in the /Applications/Utilities folder. Once launched, choose Keychain Access > Certificate Assistant > Request a Certificate From a Certificate Authority (see Figure 21-6). You need to perform this step again even if you've already created previous requests for your developer and distribution certificates. The new request adds information that uniquely identifies the SSL certificate.

From the Library of Eric Berna

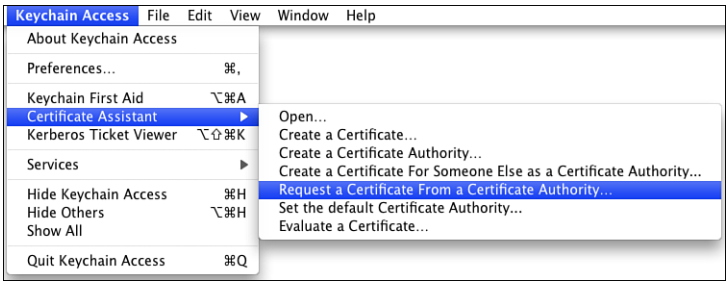


Figure 21-6 Create a new certificate request even though you've probably already done so in the past for your developer and distribution certificates.

Once the Certificate Assistant opens, enter your e-mail address and add a recognizable common name such as Push Tutorial App. This common name is important. It will come in handy for the future, so choose one that is easy to identify and accurately describes your project. The common name lets you distinguish otherwise similar looking keychain items from each other in the OS X Keychain Access utility.

After specifying a common name, choose Saved to Disk and click *Continue*. The Certificate Assistant prompts you to choose a location to save to (the Desktop is handy). Click Save, wait for the certificate to be generated, and then click *Done*. Return to your Web browser and click Continue. You are now ready to submit the certificate-signing request.

Click Choose File and navigate to the request you just generated. Select it and click Choose. Click Generate to build your new SSL push service certificate. This can take a minute or two, so be patient and do not close the Web page. Once the certificate has been generated, click Continue. Download the new certificate by clicking Download (see Figure 21-7). Finally, click Done. You return to the App ID page where a new, green Enabled indicator should appear next to your app ID. Apple also e-mails you a confirmation that your certificate request was approved.

> **Note**
>
> Should you ever need to download your SSL certificate again, click Configure to return to the Configure App ID page. There, you can click Download to request another copy.

If you plan to run your Push Server from your Macintosh, add the new certificate to your keychain by double-clicking the downloaded .cer file. It will be added to your login keychain and appear in your Certificates. You can identify the certificate by clicking the small triangle next to it to reveal the common name you used when creating the certificate request.

From the Library of Eric Berna



Figure 21-7 Download your SSL certificate to your Mac.

Push-Specific Provisions

You don't use wild-card team provisions for push-enabled applications. Instead, you must create a single provision for just that application. This means that if you intend to create development, ad hoc, and distribution versions of your app, you must request three new mobile provision files in addition to whatever provisions you have already created for other work.

Go to the Provisioning section of the developer portal. Select Development or Distribution profile by clicking the appropriate tab. Click New Profile to begin creating your new provision. A Create Provisioning Profile page opens.

- **Development Provision**—For development, enter a profile name such as "Push Tutorial Development." Check the certificate you will be using and choose your application identifier from the pop-up list. Select the devices you will be using and click Submit.
- **Distribution Provision**—For distribution, select App Store or Ad Hoc. Enter a name for your new provision such as "Push Tutorial Distribution" or "Push Tutorial Ad Hoc." Choose your application identifier from the pop-up list. For ad-hoc distribution only, select the devices to include in your provision. Click Submit to finish.

From the Library of Eric Berna

It may take a moment or two for your profile to generate. Wait a short while and reload the page. The provision status should change from Pending to Active. Download your new provision and add it to Xcode through the Organizer > Devices > Provisioning Profiles. Click Import, select the provision from your downloads folder, and click Open.

Creating a Push-Compatible Application

After installing your push provision into Xcode, you are ready to create a push-compatible application. Start by ensuring that your new bundle identifier matches the provision you just created in TARGETS > Info > Bundle identifier. Then, in Build Settings > Code Signing Identity, choose the developer identity for the new provision.

Registering Your Application

Signing an application with a push-compatible mobile provision is just the first step to working with push notifications. The application must request to register itself with the iOS device's remote notification system. You do this with a single `UIApplication` call, as follows. The `application: didFinishLaunchingWithOptions:` delegate method provides a particularly convenient place to call this.

```
[[UIApplication sharedApplication]
    registerForRemoteNotificationTypes:types];
```

This call tells iOS that your application wants to accept push messages. The types you pass specify what kinds of alerts your application will receive. iOS offers three types of notifications:

- `UIRemoteNotificationTypeBadge`—This notification adds a red badge to your application icon on SpringBoard.
- `UIRemoteNotificationTypeSound`—Sound notifications let you play sound files from your application bundle.
- `UIRemoteNotificationTypeAlert`—This style displays a text alert box in SpringBoard or any other application with a custom message using the alert notification.

Choose the types you want to use and `or` them together. They are bit flags, which combine to tell the notification registration process how you want to proceed. For example, the following flags allow alerts and badges but not sounds:

```
types = UIRemoteNotificationTypeBadge | UIRemoteNotificationTypeAlert;
```

Performing the registration updates user settings and allows users to customize their push preferences from Settings > Notifications. There, users can choose how to present the notification alert style (banner, alert, none), which types to allow (badge, alert, sound), whether to show the alert on the lock screen, and so forth.

From the Library of Eric Berna

To remove your application from active participation in push notifications, send `unregisterForRemoteNotifications`. This unregisters your application for all notification types and does not take any arguments.

```
[[UIApplication sharedApplication] unregisterForRemoteNotifications];
```

Retrieving the Device Token

Your application cannot receive push messages until it generates and delivers a device token to your server. It must send that device token to the offsite service that pushes the actual notifications. Recipe 21-1, which follows this section, does not implement server functionality. It provides only the client software.

A token is tied to one device. In combination with the SSL certificate, it uniquely identifies the iOS device and can be used to send messages back to the device in question. Be aware that device tokens can change after you restore device firmware.

Device tokens are created as a byproduct of registration. Upon receiving a registration request, iOS contacts the Apple Push Notification Service. It uses a Secure Socket Layer (SSL) request. Somewhat obviously, the unit must be connected to the Internet. If it is not, the request will fail. iOS forwards the request to APNS and waits for it to respond with a device token.

APNS builds the device token and returns it to iOS, which in turn passes it back to the application via an application delegate callback, namely `application:didRegisterForRemoteNotificationsWithDeviceToken:`. Your application must retrieve the device token and pass it to the provider component of your service, where it needs to be stored securely. Anyone who gains access to a device token and the application's SSL certificate could spam messages to devices. You must treat this information as sensitive and protect it accordingly.

> **Note**
>
> At times, the token may take time to generate. Consider designing around possible delays into your application by registering at each application run. Until the token is created and uploaded to your site, you will not be able to provide remote notifications to your users.

Handling Token Request Errors

At times, APNS is unable to create a token or your device may not be able to send a request. For example, you cannot generate tokens from the simulator. A `UIApplicationDelegate` method `application: didFailToRegisterForRemoteNotificationsWithError:` lets you handle these token request errors. For the most part, you'll want to retrieve the error and display it to the user.

```
// Provide a user explanation for when the registration fails
- (void)application:(UIApplication *)application
    didFailToRegisterForRemoteNotificationsWithError:(NSError *)error
```

From the Library of Eric Berna

```
{
    NSString *status = [NSString stringWithFormat:
        @"%@\nRegistration failed.\n\nError: %@", pushStatus(),
        error.localizedFailureReason];
    tbvc.textView.text = status;
}
```

Responding to Notifications

iOS uses a set chain of operations (see Figure 21-8) to respond to push notifications. When an application is running, the notification is sent directly to a `UIApplicationDelegate` method, `application:didReceiveRemoteNotification:`. The payload, which is sent in JSON format, is converted automatically into an `NSDictionary`, and the application is free to use the information in that payload however it wants. As the application is already running, no further sounds, badges, or alerts are invoked.

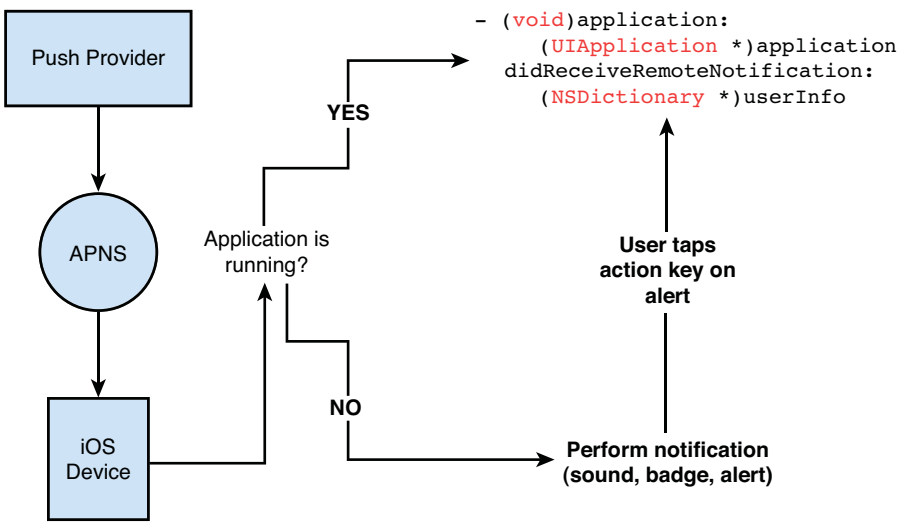


Figure 21-8 Visible and audible notification are only presented when the application is not running. Should the user tap on an alert's action key (normally View) or the application name (in the notification center), the application launches and the payload is sent as a notification to the application delegate.

```
// Handle an actual notification
- (void)application:(UIApplication *)application
    didReceiveRemoteNotification:(NSDictionary *)userInfo
{
    NSString *status = [NSString stringWithFormat:
```

From the Library of Eric Berna

```
        @"Notification received:\n%@",userInfo.description];
    tbvc.textView.text = status;
    NSLog(@"%@", userInfo);
}
```

When an application is not running, iOS performs all requested notifications that are allowed by registration and by user settings. These notifications may include playing a sound, badging the application, and/or displaying an alert. Playing a sound can also trigger iPhone vibration when a notification is received.

By default, notifications are added to the Notification Center. (Users can disable this via Settings > Notification > Application Name > Notification Center.) When the user taps the notification in the list, the application launches with the notification as a launch option.

In the case of an alert, all two-buttoned alerts offer a pair of choices. The user can tap Close (the left-most button) and close the alert or tap the alert's action key (the right-most button) and launch the app. Upon launch, the application delegate receives the same remote notification callback that an already running application would have seen (see Figure 21-9). Alerts appear on the lock screen when the device is locked and in the Notification Center, unless the user overrides the default settings.

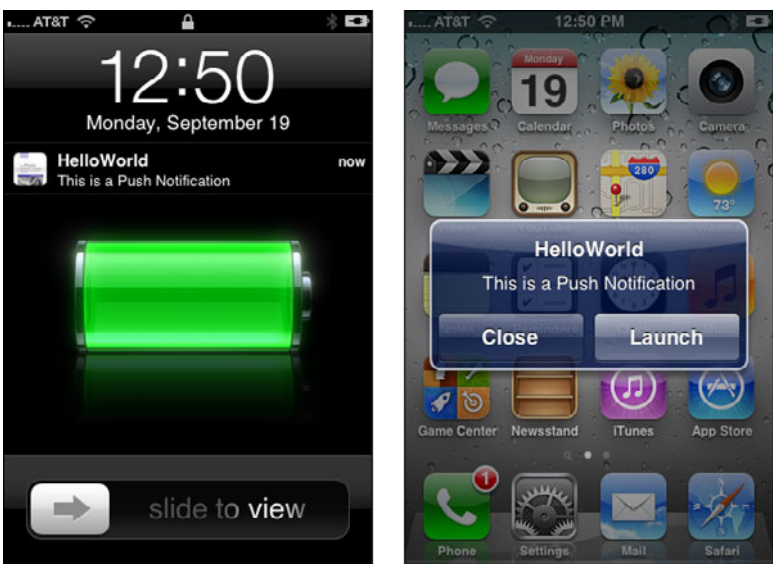


Figure 21-9 Remote alerts can appear on the Lock Screen, in SpringBoard, in applications, and in the Notification Center. Users may Close the alert or, by pressing the action button on the right (here, Launch), switch to the notifying application. In this case, that application is HelloWorld, whose name is clearly seen on the alert. The action button text is customizable.

From the Library of Eric Berna

Recipe: Push Client Skeleton

Recipe 21-1 introduces a basic client that allows users to register and unregister for push notifications. The interface (shown in Figure 21-10) uses three switches that control the services to be registered. When the application launches, it queries the app's enabled remote notification types and updates the switches to match. Thereafter, the client keeps track of registrations and unregistrations, adjusting the switches to keep sync with the reality of the settings.

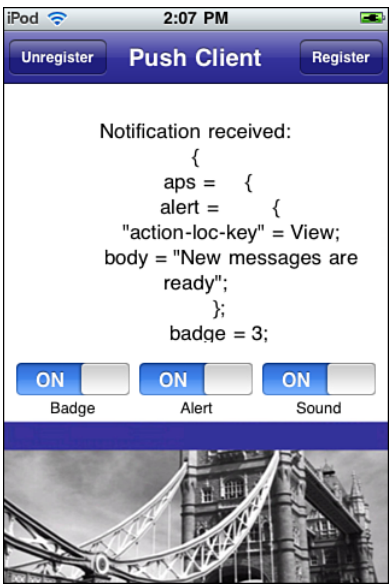


Figure 21-10 The Push Client skeleton introduced in Recipe 21-1 lets users specify which services they want to register.

Two buttons at the top left and right of the interface let users unregister and register their application. As mentioned earlier, unregistering disables all services associated with the app. It provides a clean sweep. In contrast, registering apps requires flags to indicate which services are requested.

When requesting new services, the user is always prompted to approve. Figure 21-11 shows the dialog that appears. The user must confirm by explicitly granting the application permission. If the user does not, by tapping Don't Allow, the flags remain at their previous settings.

From the Library of Eric Berna

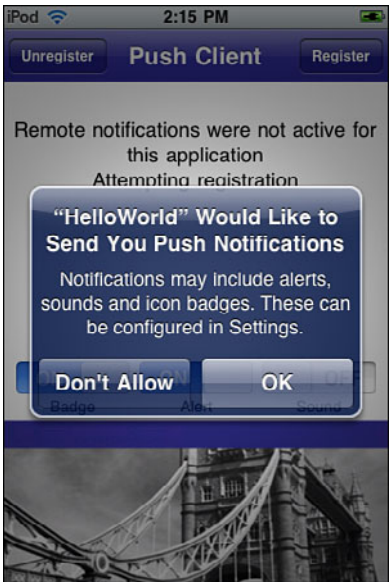


Figure 21-11 Users must explicitly grant permission for an application to receive remote notifications.

Unfortunately, the `confirmation` dialog does not generate a callback when it is dismissed, regardless of whether the user agreed or not. To catch this event, you can listen for a general notification (`UIApplicationDidBecomeActiveNotification`) that gets generated when the dialog returns control to the application. It's a hack, but it lets you know when the user responded and how the user responded. In Recipe 21-1, the `confirmationWasHidden:` method catches this notification and updates the switches to match any new registration settings.

Being something of a skeletal system, this push client doesn't actually respond to push notifications beyond showing the contents of the user info payload that gets delivered. Figure 21-10 illustrates the actual payload that was sent in Figure 21-9. This display is performed in the `application:didReceiveRemoteNotification:` method in the application delegate.

> **Note**
>
> The three sound files included in the online sample project (ping1.caf, ping2.caf, and ping3.caf) let you test sound notifications with real audio.

From the Library of Eric Berna

Recipe 21-1 **Push Client Skeleton**

```
@implementation TestBedViewController
@synthesize textView;

// Basic status
NSString *pushStatus ()
{
    return [[UIApplication sharedApplication] enabledRemoteNotificationTypes] ?
        @"Notifications were active for this application" :
        @"Remote notifications were not active for this application";
}

// Fetch the current switch settings
- (NSUInteger) switchSettings
{
    NSUInteger settings = 0;
    if ([self.view switchNamed:@"BadgeSwitch"].isOn) settings =
        settings | UIRemoteNotificationTypeBadge;
    if ([self.view switchNamed:@"AlertSwitch"].isOn) settings =
        settings | UIRemoteNotificationTypeAlert;
    if ([self.view switchNamed:@"SoundSwitch"].isOn) settings =
        settings | UIRemoteNotificationTypeSound;
    return settings;
}

// Change the switches to match reality
- (void) updateSwitches
{
    NSUInteger rntypes = [[UIApplication sharedApplication]
        enabledRemoteNotificationTypes];
    [self.view switchNamed:@"BadgeSwitch"].on =
        (rntypes & UIRemoteNotificationTypeBadge);
    [self.view switchNamed:@"AlertSwitch"].on =
        (rntypes & UIRemoteNotificationTypeAlert);
    [self.view switchNamed:@"SoundSwitch"].on =
        (rntypes & UIRemoteNotificationTypeSound);
}

// Little hack work-around to catch the end when the confirmation
// dialog goes away
- (void) confirmationWasHidden: (NSNotification *) notification
{
    // It costs nothing and has no ill side effects, so can be used
    // without worry.
    [[UIApplication sharedApplication]
        registerForRemoteNotificationTypes:[self switchSettings]];
```

From the Library of Eric Berna

```
    [self updateSwitches];
}

// Register application for the services set out by the switches
- (void) registerServices
{
    if (![self switchSettings])
    {
        textView.text = [NSString stringWithFormat:
            @"%@\nNothing to register. Skipping.", pushStatus()];
        [self updateSwitches];
        return;
    }

    NSString *status = [NSString stringWithFormat:
        @"%@\nAttempting registration", pushStatus()];
    textView.text = status;
    [[UIApplication sharedApplication]
        registerForRemoteNotificationTypes:[self switchSettings]];
}

// Unregister application for all push notifications
- (void) unregisterServices
{
    NSString *status = [NSString stringWithFormat:@"%@\nUnregistering.",
        pushStatus()];
    textView.text = status;

    [[UIApplication sharedApplication] unregisterForRemoteNotifications];
    [self updateSwitches];
}

- (void) loadView
{
    [super loadView];
    [[NSNotificationCenter defaultCenter]
        addObserver:self selector:@selector(confirmationWasHidden:)
        name:@"UIApplicationDidBecomeActiveNotification" object:nil];

    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Register", @selector(registerServices));
    self.navigationItem.leftBarButtonItem =
        BARBUTTON(@"Unregister", @selector(unregisterServices));
    [self updateSwitches];
}
@end
```

From the Library of Eric Berna

```
#pragma mark - Application Delegate

@implementation TestBedAppDelegate
// Retrieve the device token
- (void)application:(UIApplication *)application
    didRegisterForRemoteNotificationsWithDeviceToken:(NSData *)deviceToken
{
    NSUInteger rntypes = [[UIApplication sharedApplication]
        enabledRemoteNotificationTypes];
    NSString *results = [NSString stringWithFormat:
        @"Badge: %@, Alert:%@, Sound: %@",
        (rntypes & UIRemoteNotificationTypeBadge) ? @"Yes" : @"No",
        (rntypes & UIRemoteNotificationTypeAlert) ? @"Yes" : @"No",
        (rntypes & UIRemoteNotificationTypeSound) ? @"Yes" : @"No"];

    // Show the retrieved Device Token
    NSString *status = [NSString stringWithFormat:
        @"%@\nRegistration succeeded.\n\nDevice Token: %@\n%@",
        pushStatus(), deviceToken, results];
    tbvc.textView.text = status;
    NSLog(@"deviceToken: %@", deviceToken);

    // Write token to file for easy retrieval in iTunes.
    // Handy for learning push dev - but not a great idea for
    // App Store deployment. Set UIFileSharingEnabled to YES
    [deviceToken.description writeToFile:[NSHomeDirectory()
        stringByAppendingPathComponent:@"Documents/DeviceToken.txt"]
        atomically:YES encoding:NSUTF8StringEncoding error:nil];
}

// Respond to failed registration
- (void)application:(UIApplication *)application
    didFailToRegisterForRemoteNotificationsWithError:(NSError *)error
{
    NSLog(@"Error registering for remote notifications: %@",
        error.localizedFailureReason);
    NSString *status = [NSString stringWithFormat:
        @"%@\nRegistration failed.\n\nError: %@",
        pushStatus(), error.localizedFailureReason];
    tbvc.textView.text = status;
}

// Handle an actual notification
- (void)application:(UIApplication *)application
    didReceiveRemoteNotification:(NSDictionary *)userInfo
{
```

From the Library of Eric Berna

```
    NSString *status = [NSString stringWithFormat:
        @"Notification received:\n%@", userInfo.description];
    tbvc.textView.text = status;
    NSLog(@"%@", userInfo);
}

// Little work-around for showing text
- (void) showString: (NSString *) aString
{
    tbvc.textView.text = aString;
}

// Report the notification payload when launched by alert
- (void) launchNotification: (NSNotification *) notification
{
    // Workaround allows the text view to be created if needed first
    [self performSelector:@selector(showString:)
         withObject:[[notification userInfo] description]
         afterDelay:1.0f];
}

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    tbvc = [[TestBedViewController alloc] init];
    UINavigationController *nav =
        [[UINavigationController alloc] initWithRootViewController:tbvc];
    window.rootViewController = nav;
    [window makeKeyAndVisible];

    // Listen for remote notification launches
    [[NSNotificationCenter defaultCenter]
        addObserver:self selector:@selector(launchNotification:)
        name:@"UIApplicationDidFinishLaunchingNotification" object:nil];

    NSLog(@"Launch options: %@", launchOptions);
    return YES;
}
@end
```

Get This Recipe's Code

To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from this book, go to the folder for Chapter 21 and open the project for this recipe.

From the Library of Eric Berna

Building Notification Payloads

Delivering push notification through APNS requires three things: your SSL certificate, a device ID, and a custom payload with the notification you want to send. The payload uses JSON formatting. You've already read about generating the certificate and producing the device identifiers, which you need to pass up to your server. Building the JSON payloads basically involves transforming a small well-defined dictionary into JSON format.

JavaScript Object Notation (JSON) is a simple data interchange format based on key-value pairs. The JSON Web site (www.json.org) offers a full syntax breakdown of the format, which allows you to represent values that are strings, numbers, and arrays. The APNS payload consists of up to 256 bytes, which must contain your complete notification information.

Notification payloads must include an `aps` dictionary. This dictionary defines the properties that produce the sound, badge, and/or alert sent to the user. In addition, you can add custom dictionaries with any data you need to send to your application so long as you stay within the 256 byte limit. Figure 21-12 shows the hierarchy for basic (nonlocalized) alerts.

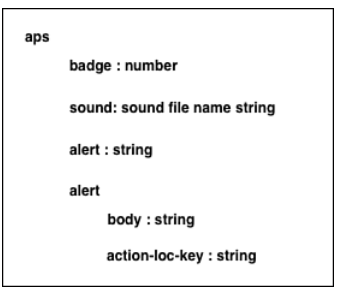


Figure 21-12 The `aps` dictionary may contain one or more notification types, including a badge request, a sound file, and/or an alert.

The `aps` dictionary contains one or more notification types. These include the standard types you've already read about: badges, sounds, and alerts. Badge and sound notifications each take one argument. The badge is set by a number, the sound by a string that refers to a file already inside the application bundle. If that file is not found (or the developer passes `default` as the argument), a default sound plays for any notification with a sound request. When a badge request is not included, iOS removes any existing badge from the application icon.

There are two ways to produce an alert. You can pass a string, which defines the message to show. This automatically produces a notification with two buttons under that message: Close and View. To customize buttons, pass a dictionary instead. Send the message text as the body and the string to use for the action key (normally *Launch*) as `action-loc-key`. This replaces View with whatever text you specify.

From the Library of Eric Berna

To produce an alert with a single OK button, pass `null` as the argument to `action-loc-key`. This creates a special alert style with one button. Just as when a user taps Close, the OK style alert will not pass any data directly to your application. The app must poll for any updates when next opened by the user.

Localized Alerts

When working with localized applications, construct your `aps > alert` dictionary with two additional keys. Use `loc-key` to pass a key that is defined in your application's Localizable.strings file. iOS looks up the key and replaces it with the string found for the current localization.

At times, localization strings use arguments like `%@` and `%n$@`. Should that hold true for the localization you are using, you can pass those arguments as an array of strings via `loc-args`. As a rule, Apple recommends against using complicated localizations, because they can consume a major portion of your 256-byte bandwidth.

Transforming from Dictionary to JSON

Once you've designed your dictionary, you must transform it to JSON. The JSON format is simple but precise. Use the `NSJSONSerialization` class to convert your dictionary to the JSON string. Here's the basic call.

```
NSData *jsonData = [NSJSONSerialization
    dataWithJSONObject:mainDict options:0 error:nil];
```

Here's a quick rundown of JSON basics. Table 21-1 offers examples of these rules in action.

- The entire payload is a dictionary. Dictionaries consist of key-value pairs stored between brackets; that is, {key:value, key:value, key:value, ...}.
- Key-value pairs are separated with commas.
- Strings use double quotes; numbers do not. Reserved words include true, false, and null. Reserved words are not quoted.
- Arrays consist of a list of items between square brackets, that is, [item, item, item,...].
- The following symbols must be escaped in strings by using a backslash literal indicator: `' " \ /`.
- You may want to remove carriage returns (`\r`) and new lines (`\n`) from your payloads when sending messages.
- Spaces are optional. Save space by omitting them between items.
- The `aps` dictionary appears within the top-level folder, so the most basic payload looks something like `{aps:{}}`.

From the Library of Eric Berna

Table 21-1 **JSON Payload Samples**

Sample Type	JSON
Hello message, displays with two buttons.	`{"aps":{"alert":"hello"}}`
Hello message, displays with two buttons, but built using JSON with an alert dictionary.	`{"aps":{"alert":{"body":"hello"}}}`
Hello message with one OK button.	`{"aps":{"alert":{"action-loc-key": null,"body":"hello"}}}`
Hello message with two buttons, Close and Open, the latter being a custom replacement for View.	`{"aps":{"alert":{"action-loc key":"Open","body":"hello"}}}`
Hello message that adds an application badge of 3.	`{"aps":{"badge":3,"alert": {"body":"hello"}}}`
Play a sound without an alert.	`{"aps":{"sound":"ping2.caf","alert":{}}}`
Play sound, display badge, display alert, use a custom button.	`{"aps":{"sound":"ping2.caf","badge":2, "alert":{"action-loc key":"Open","body":"Hello"}}}`
Add a custom payload including an array.	`{"aps":{"alert":{"body":"Hello"}}, "key1":"value1","key2":["a","b","c"]}`

Custom Data

So long as your payload has room left, keeping in mind your tight byte budget, you can send additional information in the form of key-value pairs. As Table 21-1 showed, these custom items can include arrays and dictionaries as well as strings, numbers, and constants. You define how to use and interpret this additional information. The entire payload dictionary is sent to your application so whatever information you pass along will be available to the `application:didReceiveRemoteNotification:` method via the user dictionary.

A dictionary containing custom key-value pairs does *not* need to provide an alert, although doing so allows your user to choose to open your application if it isn't running. If your application is already launched, the key-value pairs arrive as a part of the payload dictionary.

Receiving Data on Launch

When your client receives a notification, tapping the action key (by default, View) launches your application. Then, after launching, iOS sends your application delegate an

From the Library of Eric Berna

optional callback. The delegate recovers its notification dictionary by implementing a method named `application:didFinishLaunchingWithOptions:`.

iOS passes the notification dictionary to the delegate method via the launch options parameter. For remote notifications, this is the official callback to retrieve data from an alert-box launch. The `didReceiveRemoteNotification:` method is not called when iOS receives a notification and the application is not running.

This "finished launching" method is actually designed to handle many different launch circumstances. Push notification is just one. Others include opening by local notifications, by URL scheme handling, and so forth. In any case, the method must return a Boolean value. As a rule, return `YES` if you were able to process the request or `NO` if you were not. This value is actually ignored in the case of remote notification launches, but you must still return a value.

> **Note**
>
> When your user taps Close and later opens your application, the notification is not sent on launch. You must check in with your server manually to retrieve any new user information. Applications are not guaranteed to receive alerts. In addition to tapping Close, the alert may simply get lost. Always design your application so that it doesn't rely solely on receiving push notifications to update itself and its data.

Recipe: Sending Notifications

The notification process involves several steps (see Figure 21-13). First, you build your JSON payload, which you just read about. Next, you retrieve the SSL certificate and the device token for the unit you want to send to. How you store these is left up to you, but you must remember that these are sensitive pieces of information. Open a secure connection to the APNS server. Finally, you handshake with the server, send the notification package, and close the connection.

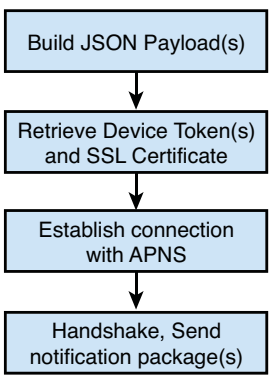


Figure 21-13 The steps for sending remote notifications.

From the Library of Eric Berna

This is the most basic way of communicating and assumes you have just one payload to send. In fact, you can establish a session and send many packets at a time; however, that is left as an exercise for the reader as is creating services in languages other than Objective-C. The Apple Developer Forums (devforums.apple.com) host ongoing discussions about push providers and offer an excellent jumping off point for finding sample code for PHP, Perl, and other languages.

Be aware that APNS may react badly to a rapid series of connections that are repeatedly established and torn down. If you have multiple notifications to send at once, go ahead and send them during a single session. Otherwise, APNS might confuse your push deliveries with a denial of service attack.

Recipe 21-2 demonstrates how to send a single payload to APNS, showing the steps needed to implement the fourth and final box in Figure 21-13. The recipe is built around code developed by Stefan Hafeneger and uses Apple's ioSock sample source code.

The individual server setups vary greatly depending on your security, databases, organization, and programming language. Recipe 21-2 demonstrates a minimum of what is required to implement this functionality and serves as a template for your own server implementation in whatever form this might take.

Sandbox and Production

Apple provides both sandbox (development) and production (distribution) environments for push notification. You must create separate SSL certificates for each. The sandbox helps you develop and test your application before submitting to App Store. It works with a smaller set of servers and is not meant for large-scale testing. The production system is reserved for deployed applications that have been accepted to App Store.

- The sandbox servers are located at gateway.sandbox.push.apple.com, port 2195.
- The production servers are located at gateway.push.apple.com, port 2195.

> **Note**
>
> Recipe 21-2 is meant to be compiled for and used on the Macintosh as a command-line utility.

Recipe 21-2 **Pushing Payloads to the APNS Server**

```
// Adapted from code by Stefan Hafeneger
- (BOOL) push: (NSString *) payload
{
    otSocket socket;
    SSLContextRef context;
    SecKeychainRef keychain;
    SecIdentityRef identity;
    SecCertificateRef certificate;
    OSStatus result;
```

From the Library of Eric Berna

```
// Ensure device token
if (!deviceTokenID)
{
    fprintf(stderr, "Error: Device Token is nil\n");
    return NO;
}

// Ensure certificate
if (!certificateData)
{
    fprintf(stderr, "Error: Certificate Data is nil\n");
    return NO;
}

// Establish connection to server.
PeerSpec peer;
result = MakeServerConnection("gateway.sandbox.push.apple.com",
    2195, &socket, &peer);
if (result)
{
    fprintf(stderr, "Error creating server connection\n");
    return NO;
}

// Create new SSL context.
result = SSLNewContext(false, &context);
if (result)
{
    fprintf(stderr, "Error creating SSL context\n");
    return NO;
}

// Set callback functions for SSL context.
result = SSLSetIOFuncs(context, SocketRead, SocketWrite);
if (result)
{
    fprintf(stderr, "Error setting SSL context callback functions\n");
    return NO;
}

// Set SSL context connection.
result = SSLSetConnection(context, socket);
if (result)
{
    fprintf(stderr, "Error setting the SSL context connection\n");
    return NO;
}
```

From the Library of Eric Berna

```
// Set server domain name.
result = SSLSetPeerDomainName(context,
    "gateway.sandbox.push.apple.com", 30);
if (result)
{
    fprintf(stderr, "Error setting the server domain name\n");
    return NO;
}

// Open keychain.
result = SecKeychainCopyDefault(&keychain);
if (result)
{
    fprintf(stderr, "Error accessing keychain\n");
    return NO;
}

// Create certificate from data
CSSM_DATA data;
data.Data = (uint8 *)certificateData.bytes;
data.Length = certificateData.length;
result = SecCertificateCreateFromData(&data, CSSM_CERT_X_509v3,
    CSSM_CERT_ENCODING_BER, &certificate);
if (result)
{
    fprintf(stderr, "Error creating certificate from data\n");
    return NO;
}

// Create identity.
result = SecIdentityCreateWithCertificate(keychain, certificate,
    &identity);
if (result)
{
    fprintf(stderr, "Error creating identity from certificate\n");
    return NO;
}

// Set client certificate.
CFArrayRef certificates = CFArrayCreate(NULL,
    (const void **)&identity, 1, NULL);
result = SSLSetCertificate(context, certificates);
if (result)
{
    fprintf(stderr, "Error setting the client certificate\n");
    return NO;
}
```

From the Library of Eric Berna

```
CFRelease(certificates);

// Perform SSL handshake.
do {result = SSLHandshake(context);}
    while(result == errSSLWouldBlock);


// Convert string into device token data.
NSMutableData *deviceToken = [NSMutableData data];
unsigned value;
NSScanner *scanner = [NSScanner
    scannerWithString:self.deviceTokenID];
while(![scanner isAtEnd]) {
    [scanner scanHexInt:&value];
    value = htonl(value);
    [deviceToken appendBytes:&value length:sizeof(value)];
}

// Create C input variables.
char *deviceTokenBinary = (char *)deviceToken.bytes;
char *payloadBinary = (char *)[payload UTF8String];
size_t payloadLength = strlen(payloadBinary);

// Prepare message
uint8_t command = 0;
char message[293];
char *pointer = message;
uint16_t networkTokenLength = htons(32);
uint16_t networkPayloadLength = htons(payloadLength);

// Compose message.
memcpy(pointer, &command, sizeof(uint8_t));
pointer += sizeof(uint8_t);
memcpy(pointer, &networkTokenLength, sizeof(uint16_t));
pointer += sizeof(uint16_t);
memcpy(pointer, deviceTokenBinary, 32);
pointer += 32;
memcpy(pointer, &networkPayloadLength, sizeof(uint16_t));
pointer += sizeof(uint16_t);
memcpy(pointer, payloadBinary, payloadLength);
pointer += payloadLength;

// Send message over SSL.
size_t processed = 0;
result = SSLWrite(context, &message, (pointer - message),
    &processed);
```

From the Library of Eric Berna

```
    if (result)
    {
        fprintf(stderr, "Error sending message via SSL.\n");
        return NO;
    }
    else
    {
        printf("Message sent.\n");
        return YES;
    }
}
```

Get This Recipe's Code

To get the code used for this recipe, go to https://github.com/erica/iOS-5-Cookbook, or if you've downloaded the disk image containing all the sample code from this book, go to the folder for Chapter 21 and open the project for this recipe.

Feedback Service

Apps don't live forever. Users add, remove, and replace applications on their devices all the time. From an APNS point of view, it's pointless to deliver notifications to devices that no longer host your application. As a push provider, it's your duty to remove inactive device tokens from your active support list. As Apple puts it, "APNS monitors providers for their diligence in checking the feedback service and refraining from sending push notifications to nonexistent applications on devices." Big Brother *is* watching.

Apple provides a simple way to manage inactive device tokens. When users uninstall apps from a device, push notifications begin to fail. Apple tracks these failures and provides reports from its APNS feedback server. The APNS feedback service lists devices that failed to receive notifications. As a provider, you need to fetch this report on a periodic basis and weed through your device tokens.

The feedback server hosts sandbox and production addresses, just like the notification server. You find these at feedback.push.apple.com (port 2196) and feedback.sandbox.push.apple.com. You contact the server with a production SSL certificate and shake hands in the same way you do to send notifications. After the handshake, read your results. The server sends data immediately without any further explicit commands on your side.

The feedback data consists of 38 bytes. This includes the time (4 bytes), the token length (2 bytes), and the token itself (32 bytes). The timestamp tells you when APNS first determined that the application no longer existed on the device. This uses a standard UNIX epoch, namely seconds since Midnight, January 1, 1970. The device token is stored in binary format. You need to convert it to a hex representation to match it to your

From the Library of Eric Berna

device tokens if you use strings to store token data. At the time of writing this book, you can ignore the length bytes. They are always 0 and 32, referring to the 32-byte length of the device token.

```
// Retrieve message from SSL.
size_t processed = 0;
char buffer[38];
do
{
    // Fetch the next item
    result = SSLRead(context, buffer, 38, &processed);
    if (result) break;

    // Recover Date from data
    char *b = buffer;
    NSTimeInterval ti = ((unsigned char)b[0] << 24) +
       ((unsigned char)b[1] << 16) +
       ((unsigned char)b[2] << 8) +
       (unsigned char)b[3];
    NSDate *date = [NSDate dateWithTimeIntervalSince1970:ti];

    // Recover Device ID
    NSMutableString *deviceID = [NSMutableString string];
    b += 6;
    for (int i = 0; i < 32; i++) [
        deviceID appendFormat:@"%02x", (unsigned char)b[i]];

    // Add dictionary to results
    [results addObject:
        [NSDictionary dictionaryWithObject:date
        forKey:deviceID]];

} while (processed > 0);
```

> **Note**
>
> Search your Xcode Organizer Console for “aps” to locate APNS error messages.

Designing for Push

When designing for push, keep scaling in mind. Normal computing doesn’t need to scale. When coding is done, an app runs on a device using the local CPU. Should a developer deploy an extra 10,000 copies, there’s no further investment involved other than increased technical support.

From the Library of Eric Berna

Push computing does scale. Whether you have 10,000 or 100,000 or 1,000,000 users matters. That's because developers must provide the service layer that handles the operations for every unit sold. The more users supported, the greater the costs will be. Consider that these services need to be completely reliable and that consumers will not be tolerant of extended downtimes.

Consider an application with just 10,000 users. It might service a million uses per day, assuming update checks every 15 minutes. More time-critical uses might demand checks every few minutes or even several times a minute. As the computational burden builds, so do the hosting costs. While cloud computing provides an excellent match to these kinds of needs, that kind of solution comes with a real price in development, maintenance, and day-to-day operations.

On top of reliability, add in security concerns. Many polled services require secure credentials. Those credentials must be uploaded to the service for remote use rather than being stored solely on the device. Even if the service in question does not use that kind of authentication, the device token that allows your service to contact a specific device is sensitive in itself. Should that identifier be stolen, it could let spammers send unsolicited alerts. Any developer who enters this arena must take these possible threats seriously and provide highly secure solutions for storing and protecting information.

These concerns, when taken together, point to the fact that push notifications are serious business. Some small development houses may completely opt out of being push providers for apps that depend on new information notifications. Between infrastructure and security concerns, the work it will take to properly offer this kind of service may price itself out of reach for those developers. Third party provider Urban Airship (urbanairship.com) offers ready-to-use push infrastructure with affordable pricing plans. It handles the remote notification deployment for you.

Summary

In this chapter, you saw push notifications both from a client-building point of view and as a provider. You learned about the kinds of notifications you can send and how to create the payload that moves those notifications to the device. You discovered registering and unregistering devices and how users can opt in and out from the service.

Much of the push story lies outside this chapter. It's up to you to set up a server and deal with security, bandwidth, and scaling issues. The reality of deployment is that there are many platforms and languages that can be used that go beyond the Objective-C sample code shown here, typically via PHP. Regardless, the concepts discussed and recipes shown in this chapter give you a good stepping-off point. You know what the issues are and how things have to work. Now it's up to you to put them to good use:

- The big wins of notifications are their instant updates and immediate presentation. Like SMS messages, they're hard to overlook when they arrive on your device. There's nothing wrong in opting out of push if your application does not demand that kind of immediacy.

From the Library of Eric Berna

- Guard your SSL certificate and device tokens. It varies how Apple will respond to security breaches, but experience suggests that it will be messy and unpleasant.
- Don't leave users without service when you have promised to provide it to them. Build a timeline into your business plan that anticipates what it will take to keep delivering notifications over time and how you will fund this. Consumers will not be tolerant of extended downtimes; your service must be completely reliable.
- Apple provides a verbose logging tool for push notification, which you can read about at its Developer Forums site. See https://devforums.apple.com/message/58759 for details. (You will need to sign in with developer credentials to read this thread.)
- Don't spam users. Push notifications are not meant to sell products or promote sales. Respect your user base.
- Build to scale. Although your application may not initially have tens of thousands of users, you must anticipate a successful app launch as well as a modest one. Create a system that can grow along with your user base.

From the Library of Eric Berna

This page intentionally left blank

From the Library of Eric Berna

22

Accessibility

Accessibility enhancements open up the iPhone to users with disabilities. iPhone OS features allow users to magnify (or "zoom") displays, invert colors, and more. As a developer, accessibility enhancement centers on VoiceOver, a way that visually impaired users can "listen" to their GUI. VoiceOver converts an application's visual presentation into an audio description.

Don't confuse VoiceOver with Voice Control or the Siri Assistant. The former is a method for presenting an audio description of a user interface and is highly gesture-based. The latter refer to Apple's proprietary voice recognition technology for hands-free interaction.

This chapter briefly overviews VoiceOver accessibility. You read about adding accessibility labels and hints to your applications and testing those features in the simulator and on the iOS device. Accessibility is available and can be tested on third generation or later devices, including all iPads, the iPhone 3GS and later, and the third-generation iPod touch and later.

Accessibility in Interface Builder

Use the Interface Builder's Identity Inspector > Accessibility pane (see Figure 22-1) to add labels and hints to the UIKit elements in your interface. These fields and the text they contain play different roles in the accessibility picture. Labels identify views; hints describe them. In addition to these fields, you'll find a general accessibility Enabled check box and a number of Traits check boxes.

From the Library of Eric Berna

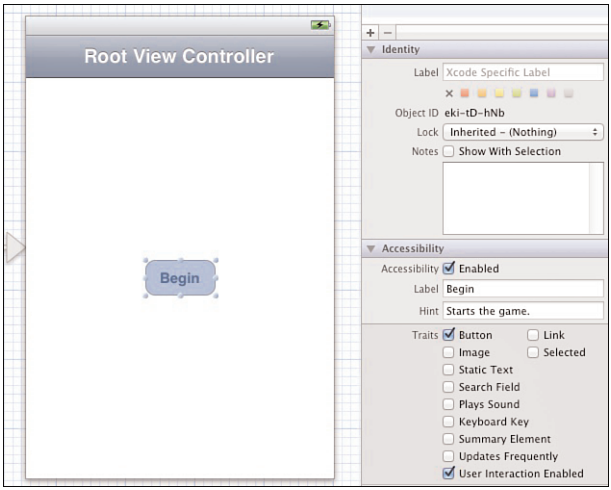


Figure 22-1 Interface Builder's Identity Inspector lets you specify object accessibility information.

Labels

A good label tells the user what an item is, often with a single word. Label an accessible GUI the same way you'd label a button with text. "Edit," "Delete," and "Add" all describe what objects do. They're excellent button text and accessibility label text.

But accessibility isn't just about buttons. "Feedback," "User Photo," and "User Name" might describe the contents and function of a text view, an image view, and a text label. If an object plays a visual role in your interface, it should play an auditory role in VoiceOver. Here are a few tips for designing your labels:

- **Do not add the view type into the label**—For example, don't use "Delete button," "Feedback text view," or "User name text field." VoiceOver adds this information automatically, so "Delete button" in the identity pane becomes "Delete button button" in the actual VoiceOver playback.
- **Capitalize the label but don t add a period**—VoiceOver uses your capitalization to properly inflect the label when it speaks. Adding a period typically causes VoiceOver to end the label with a downward tone, which does not blend well into the object-type that follows. "Delete. button" sounds wrong. "Delete button" does not.
- **Aggregate information**—When working with complex views that function as a single unit, build all the information in that view into a single descriptive label and attach it to that parent view. For example, in a table view cell with several subviews but without individual controls, you might aggregate all the text information into a single label that describes the entire cell.

From the Library of Eric Berna

- **Label only at the lowest interaction level**—When users need to interact with subviews, label at that level. Parent views, whose children are accessible, do not need labels.
- **Localize**—Localizing your accessibility strings opens them up to the widest audience of users.

Hints

Hints tell users what to expect from interaction. In particular, they describe any nonobvious results. For example, consider an interface where tapping on a name, for example, John Smith attempts to call that person by telephone. The name itself offers no information about the interaction outcome. So, offer a hint telling the user about it—for example, "Places a phone call to this person," or even better, "Places a phone call to John Smith." Here are tips for building better hints:

- **Use sentence form**—Start with a capital letter and end with a period. Do this even though each hint has a missing subject (e.g., "Clears text in the form"). Here, the missing subject is "[*This button*]". Sentence format ensures that VoiceOver speaks the hint with proper inflection.
- **Use verbs that describe what the element does, not what the user does**—*"[This text label]* Places a phone call to this person." provides the right context for the user. "*[You will]* Place a phone call to this person." does not.
- **Do not say the name or type of the GUI element**—Avoid hints that refer to the UI item being manipulated. Skip the GUI name (its label, such as "Delete") and type (its class, such as "button"). VoiceOver adds that information where needed, preventing any overly redundant playback, such as "Delete button *[label]* button *[VoiceOver description]* button *[hint]* removes item from screen." Use "Removes item from screen." instead.
- **Avoid the action**—Do not describe the action that the user takes. Do not say "Swiping places a phone call to this person" or "Tapping places a phone call to this person." VoiceOver uses its own set of gestures to activate GUI elements. Never refer to gestures directly.
- **Be verbose**—"Place call" does not describe the outcome as well as "Place a call to this person," or, even better, "Place a call to John Smith." A short but thorough explanation better helps the user than one that is so terse that the user has to guess about details. Avoid hints that require the user to listen again before proceeding.
- **Localize**—As with labels, localizing your accessibility hints works with the widest user base.

From the Library of Eric Berna

Enabling Accessibility

The Enabled check box controls whether a UIKit view works with VoiceOver. As a rule, keep this item checked unless the view is a container whose subviews need to be accessible. Enable only those items at the most direct level of interaction or presentation. Views that organize other views don't play a meaningful role in the voice presentation. Exclude them.

Table view cells offer a good example of accessibility containers, that is, objects that contain other objects. The rules for table view cells are as follows:

- A table view cell without embedded controls should be accessible.
- A table view cell with embedded controls should not be. Its child controls should be.

Nonaccessible containers are responsible for reporting how many accessible children they contain and which child views those are. See Apple's Accessibility Programming Guide for iPhone for further details about programming containers for accessibility. Custom container views need to declare and implement the `UIAccessibilityContainer` protocol.

Traits

Traits characterize UIKit item behaviors. VoiceOver uses these traits while describing interfaces. As Figure 22-1 shows, there are numerous possible traits you can assign to views. Select the traits that apply to the selected view, keeping in mind that you can always update these choices programmatically.

Apple's accessibility documents request that you only check one of the following four mutually exclusive items at any time: Button, Link, Static Text, or Search Field. If a button works as a link as well, choose either the button trait or the link trait but not both. You choose which best characterizes how that button is used. At the same time, a button might show an image and play a sound when tapped, and you can freely add those traits.

Working with Accessibility from Code

Every UIKit view conforms to the `UIAccessibility` protocol, offering properties that let you set labels and hints, along with the other accessibility features shown in Figure 22-1. You can set those properties in Interface Builder or use them directly in code. Listing 22-1 sets the `accessibilityHint` property to update a button's hint as a user types a username into a related text field. As the text in that field changes, the button's hint updates to reflect that value.

From the Library of Eric Berna

Listing 22-1 **Programmatically Updating Accessibility Information**

```
- (BOOL)textField:(UITextField *)textField
    shouldChangeCharactersInRange:(NSRange)range
    replacementString:(NSString *)string
{
    // Catch the change to the user name field and update
    // the accessibility hint to mirror that
    NSString *username = textField.text;
    if (username && username.length > 1)
        callbutton.accessibilityHint = [NSString
          stringWithFormat:@"Places a call to %@", username];
    else
        callbutton.accessibilityHint =
            @"Places a call to the person named in the text field.";
    return YES;
}
```

Testing with the Simulator

The iOS simulator's Accessibility Inspector is designed for testing accessible applications before deploying them to the iOS device. The simulator's inspector simulates VoiceOver interaction with your application, providing immediate visual feedback via a floating pane (there is no actual voice produced) without having to use the VoiceOver gesture interface directly. Because you cannot replicate many VoiceOver gestures with the simulator (such as triple-swipes and sequential hold-then-tap gestures), the inspector focuses on describing interface items rather than responding to VoiceOver gestures.

Enable this feature by opening Settings > General > Accessibility. Switch the Accessibility Inspector to On. The inspector, shown in Figure 22-2, immediately appears. It lists the current settings for the currently selected accessible element.

From the Library of Eric Berna

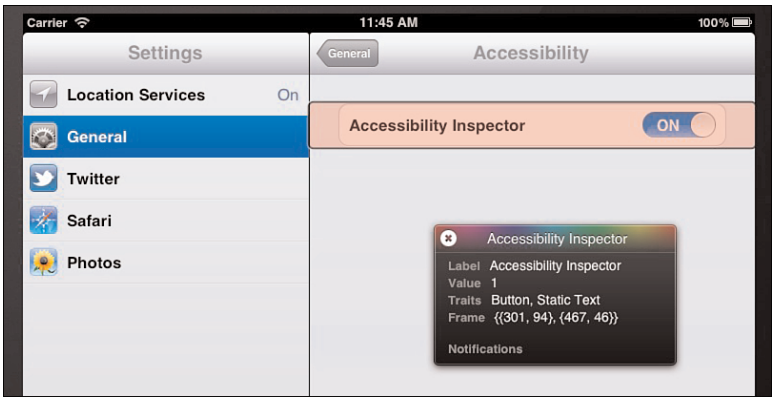


Figure 22-2 The iPhone simulator's Accessibility Inspector highlights the currently selected GUI feature, revealing its label, hint, and other accessibility properties.

Know how to enable and disable the inspector: The circled X in the top-left corner of the inspector controls that behavior. Click it once to shrink the inspector to a disabled single line. Click again to restore the inspector to active mode. For the most part, keep the inspector disabled until you actually need to inspect a GUI item. Figure 22-3 shows the button interface defined in Figure 22-1, as described in the Accessibility Inspector.

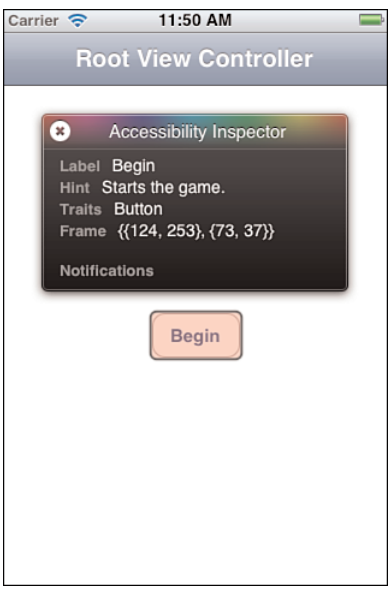


Figure 22-3 The Accessibility Inspector reflects the values set either in Interface Builder or in code that describe the currently-selected item.

From the Library of Eric Berna

Like VoiceOver, the inspector interferes with normal application gestures. It will slow down your work, so use it sparingly, normally when you are ready to test. You want to launch your application with the inspector disabled but available. Navigate to the screen you want to work with and then enable the inspector.

When you update accessibility hints in code, the inspector updates in real time to match those changes. Activating the inspector allows you to view the current hint as those changes happen, ensuring that the onscreen hints and labels reflect the up-to-date interface.

Broadcasting Updates

Your application should post notifications to let the VoiceOver accessibility system know about on-screen element changes outside of direct user interaction:

- When you add or remove a GUI element. The `UIAccessibilityLayoutChangedNotification` tells gives the VoiceOver accessibility system a heads up about those changes.
- Applications can post a `UIAccessibilityPageScrolledNotification` after completing a scroll action. The notification's object should contain a description of the new scroll position (e.g., "Page 5 of 17" or "Tab 2 of 4").
- When the zoomed frame changes, send a `UIAccessibilityZoomFocusChanged` notification. Include a user dictionary with `type`, `frame`, and `view` parameters. These parameters specify the type of zoom that has taken place, the currently zoomed frame (in screen coordinates), and the view that contains the zoomed frame.

In addition to these change updates, you can broadcast general announcements through the accessibility VoiceOver system. The `UIAccessibilityAnnouncementNotification` takes one parameter, a string, which contains an announcement. Use this to notify users when tiny GUI changes take place, or changes that only briefly appear on screen, or for changes that don't affect the user interface directly.

Testing Accessibility on the iPhone

Testing on the iPhone is a critical part of accessibility development. The iPhone allows you to work with the actual VoiceOver utility rather than a window-based inspector. You hear what your users will hear and are able to test your GUI with your fingers and ears rather than with your eyes.

Like the Simulator, the iPhone provides a way to enable and disable VoiceOver on the fly. Although you can enable VoiceOver in Settings and then test your application with VoiceOver running, you'll find that it's far easier to use a special toggle. The toggle lets you avoid the hassle of navigating out of Settings and over to your application using VoiceOver gestures. You can switch VoiceOver off, use normal iPhone interactions to get your application started, and then switch VoiceOver back on when you're ready to test.

From the Library of Eric Berna

To enable that toggle, follow these steps:

1. Go to the Accessibility settings pane. Navigate to Settings > General > Accessibility.
2. Locate the Triple-click Home choice. The Triple-click Home button provides a user-settable shortcut for accessibility choices. Tap Triple-click Home to open the Home pane.
3. Choose Toggle VoiceOver. Select Toggle VoiceOver to set it as your triple-click action. Once selected (a check appears to its right), you can enable and disable VoiceOver by triple-clicking the physical Home button at the bottom of your iPhone. A spoken prompt confirms the current VoiceOver setting.

This VoiceOver toggle offers you the ability to skip many of the laborious details involved in navigating to your application using triple-fingered drags, and multistage button clicks. At the same time, you should be conversant with VoiceOver gestures and interactions. Table 22-1 summarizes the VoiceOver gestures that you need to know to test your application.

Table 22-1 **Common VoiceOver Gestures for Applications**

Task	VoiceOver Equivalent
Toggle VoiceOver	Triple-click the physical Home button.
Toggle ScreenCurtain	Triple-tap the screen, three times (i.e., a triple-tap with three fingers).
Toggle VoiceOver speech	Toggle the VoiceOver speech entirely (not just for a single description) by triple-tapping the screen twice (i.e., a double-tap with three fingers). Neither of these options disables VoiceOver.
Stop speaking the current item	Double-tap the screen twice (i.e., double-tap with two fingers). Double-tap again to resume the description. In the home screen, when VoiceOver is not active, this gesture stops and resumes audio playback.
Tapping buttons	Method 1: Tap and hold the button with one finger. Tap the screen with another finger. Method 2: Tap the button to select it. Double-tap the screen to activate the button.
Scrolling a text view	Method 1: Tap and hold the text view with one finger. Tap with a second finger to scroll to the top or bottom of the text scroller. Method 2: Tap the text view to select it. Double-tap the screen to scroll to the top or bottom of the text scroller.
Adjusting the text insertion point	With an editable text view or field selected, adjust the insertion point by flicking up or down with a single finger. The point may move by characters or by word depending on how you have set up your preferences.

From the Library of Eric Berna

Task	VoiceOver Equivalent
Accessing the spoken text menu	Tap and hold one finger in the text view. Flick up and down with another finger to choose between character movement, word movement, and edit mode, which uses the last-chosen movement option. (This gesture, properly known as the "rotor," is supposed to be per formed as a twisted two-finger drag. The approach used here worked more consistently in testing.)
Selecting text	Set the insertion point and enter edit mode (see above). Tap and hold one finger in the text view. Drag left or drag right.
Typing text	Enter text edit mode by selecting a text field or text view and then double-tap the screen. The keyboard appears onscreen.
	Typing method 1: Tap and hold a keyboard button with your left pointer finger. Tap somewhere else on the screen with your right pointer finger. This is the best way to use the delete key repeatedly.
	Typing method 2: Tap on a key to select it. Double-tap the screen to type that key.
Moving sliders	Select the slider and then flick up or down with a single finger to adjust the slider value.
Scroll a list	Flick three fingers up or down.
Paging through the home page iPhone application launcher	Flick three fingers left or right.
Select and speak an item	Tap the item.
Spell out the selected item one character or word at a time	Flick a single finger up or down. This uses the settings from the spoken text menu.
Speak the next or previous item	Flick a single finger left or right.
Read the entire screen	Double-flick upwards. This doesn't work as consistently as it could. So, alternatively, use the following approach: Flick left repeatedly to the first item in the screen. Then, two-fingered stroke down. You can read the screen starting from the currently selected item using the double-fingered stroke down gesture.
Unlock iPhone	Select the Unlock slider. Double-tap the screen.

Take special note of ScreenCurtain, which allows you to blank your iPhone display, offering a true test of your application as an audio-based interface. Try the iPhone calculator application with ScreenCurtain enabled to gain a true sense of the challenge of using an iPhone application without sight.

From the Library of Eric Berna

Summary

When an iPhone application opens itself to Accessibility, it becomes an active participant in a wider ecosystem, with a larger potential user base. Here are a few thoughts to take with you:

- Including accessibility labels and hints create new audiences for your application, just like language localizations do. Adding these takes relatively little work to achieve and offers excellent payoffs to your users.
- Keep the role of labels and hints in mind as you prepare an auditory description of your application in Interface Builder.
- Don't be afraid to change hints programmatically. Let your interface descriptions update as your interface does, to provide the best-possible experience for your visually impaired users.
- iOS's accessibility system is an evolving system. Keep on top of Apple's documentation to find the latest updates and changes.
- Test with ScreenCurtain. A blank screen provides the best approximation of the VoiceOver user experience.

From the Library of Eric Berna

23

StoreKit: In-App Purchasing

StoreKit offers in-app purchasing that integrates into your software. With StoreKit, end users can use their iTunes credentials to buy unlockable features, media subscriptions, or consumable assets like fish food or sunlight, from within an application. They make these purchases after initially purchasing and installing the application from App Store. This chapter introduces StoreKit and shows you how to use the StoreKit API to create purchasing options for users. In this chapter, you read about getting started with StoreKit. You learn how set up products at iTunes Connect and localize their descriptions. You see what it takes to create test users and how to work your way through various development/deployment hurdles. This chapter teaches you how to solicit purchase requests from users and how to hand over those requests to the store for payment. By the time you finish this chapter, you'll have learned about the basic StoreKit picture, from product creation to sales.

Getting Started with StoreKit

When your application demands a more complex purchase model than buy-once use-always, consider StoreKit. StoreKit offers developers a way to sell products from within an application to create additional revenue streams. There are many reasons to use StoreKit. You might support a subscription model, provide extra game levels on demand, or introduce other unlockable features.

With StoreKit, you choose the items you want to sell and you set their price. StoreKit and iTunes take care of the details. They provide the infrastructure that brings that storefront into your application through a series of API calls and delegate callbacks.

From the Library of Eric Berna

Fulfillment

On purchase, users do not download new code. All StoreKit-based applications ship with their features already built in. For example, StoreKit purchases might let users access parts of your application that you initially set as off limits. They can also download or unlock new data sets or authorize access to subscription-based Web feeds or upgrade items in-game for more power or longer play. StoreKit provides the way users can pay to access these features, letting them go live after purchase.

StoreKit Limitations

You cannot use in-app purchasing to sell "hard" assets (such as T-shirts) or intermediate currency (such as store credit for a Web site). And, yes, real gambling is forbidden as well. Any goods sold via in-app purchase must be able to be delivered digitally to your application. Your purchasable items must not include pornography, hate speech, or defamation.

The StoreKit Development Paradox

Unfortunately, StoreKit presents a paradox, which is this: Although you can easily add IAP features to applications that are already for sale on App Store, you cannot fully develop and test your in-application purchasing for new apps until you have already submitted your application to iTunes. And you cannot fully submit your application to iTunes knowing that you're not done developing it. So, what's a developer to do? How do you properly develop new applications for StoreKit?

There is, fortunately, a solution. This approach is shown in Figure 23-1. To work around the StoreKit paradox, you upload a fully working but not fully fleshed-out application to iTunes Connect. You do this with the full understanding that you'll be replacing your binary at some point in the future.

Apple describes this placeholder as an approvable app, one that can be submitted to the App Store review process and be approved for sale. Should the application be rejected, your in-app items will stop working. You must have an app either already approved or in active review in order to develop IAP items for it.

From the Library of Eric Berna

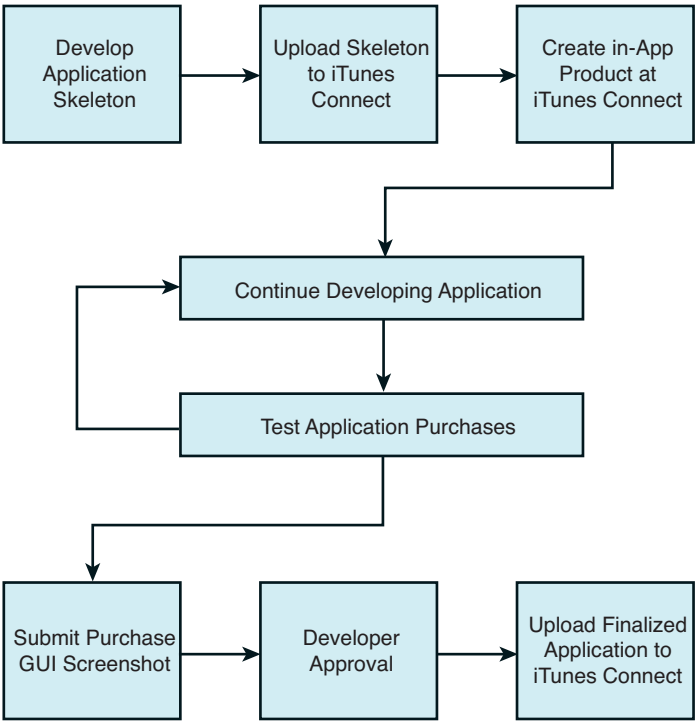


Figure 23-1 The StoreKit development process.

Apple explains it this way in its technical note TN2259; the emphatic capitals are original to the note:

> *DO NOT upload the development binary to iTunes Connect until the application is ready for App Review approval. If the binary is present in iTunes Connect and it is not fully functional, App Review will review the binary and likely reject the development binary. Testing In App Purchase will fail if you or App Review reject your most recent binary in iTunes Connect. The workaround in this case is to upload a binary without In App Purchase features that can get approved by App Review. Once the binary is approved, resume testing the binary with In App Purchase features.*

Note

When submitting your skeleton application for testing, roll back your availability date in the iTunes Connect Pricing tab. This prevents your "not ready for prime time" app from inadvertently appearing for sale on App Store until you're ready. Reset that date once you're ready to go live.

From the Library of Eric Berna

Developing and Testing

For new apps, once you've submitted your application and defined at least one in-application purchase item at iTunes Connect, you can begin to fully develop and test your application and its purchases. (For existing apps, you can create that in-app purchase item without having to submit an approvable skeleton.)

Use the sandbox version of StoreKit along with test user accounts to buy new items without charging a real credit card. The sandbox StoreKit lets you test your application features before, during, and after payment.

Submission

When you have finished development and are ready to submit a final version to App Store, you complete the StoreKit development process at iTunes Connect.

- You upload a screenshot showing the GUI for your application purchase.
- If you had uploaded a placeholder app, and it has not yet been through the review process, you self-reject it. (If it has been approved, just upload a new version instead.)
- You upload the new fully working version of your application for review that contains the in-app purchase support.
- At iTunes Connect, you submit the in-app purchase for review.

The following sections one walk you through many of the process details. You read about the StoreKit process in greater specifics and learn how to add StoreKit to your application.

Creating Test Accounts

Test accounts play a key role in the StoreKit development scenario. Create one or more new user accounts before you begin developing new StoreKit-enabled applications. These accounts allow you to log in to iTunes to test your application payments without charging real money.

Here's how you add a new user. Log in to iTunes Connect and choose Manage Users > In App Purchase Test User. Click Add New User. iTunes Connect presents the form shown in Figure 23-2. When filling out this form, keep the following points in mind:

- Each e-mail address must be unique, but it doesn't have to be real. So long as the address does not conflict with any other one in the system, you'll be fine. As you might guess, other developers have already taken the easy-to-type addresses, like abc.com, abcd.com, and so on.
- Names do not have to be real. Birthdates do not have to be real. I use a basic alphabetical naming system. My users are "a Sadun," "b Sadun," "c Sadun," and so forth. Everyone was born on January 1st.

From the Library of Eric Berna

- Passwords must be at least eight characters long and include one uppercase character, one lower case character, and a numeral. As you'd imagine, this can be a pain when it comes to typing items over and over. A single easy-to-remember disposable password can be used for all your test accounts (e.g., Aaaa1111).
- The secret question/answer fields are meaningless in this context, but they cannot be left empty. You cannot enter the same string for both fields, and the question field must be at least six characters long. Consider using a question/answer pair like "aaaaaa" and "bbbbbb" to simplify account creation.
- Selecting an iTunes Store is required. This store sets the region for your testing. If you plan to use multiple language support for various stores, make sure you create a test account in each affected region.
- You can delete user accounts and add new ones on the fly. If you run out of users who haven't yet purchased any items, just create new users as needed.
- You never want to sign into your fake user "account" in the Settings application. If you try to do so, the iOS device will force you to consent to its standard user agreement and will then try to extract a valid credit card from you. Use Settings to log out of an account but avoid it for logging in to one.
- Get into the habit of logging out your accounts in Settings each time you test your application.
- Test accounts are disposable. Once an account has purchased an item, feel free to move onto the next one, so your testing starts fresh again. There is no practical upper limit to test accounts, and you can create as many as you need to perform the testing your application demands.

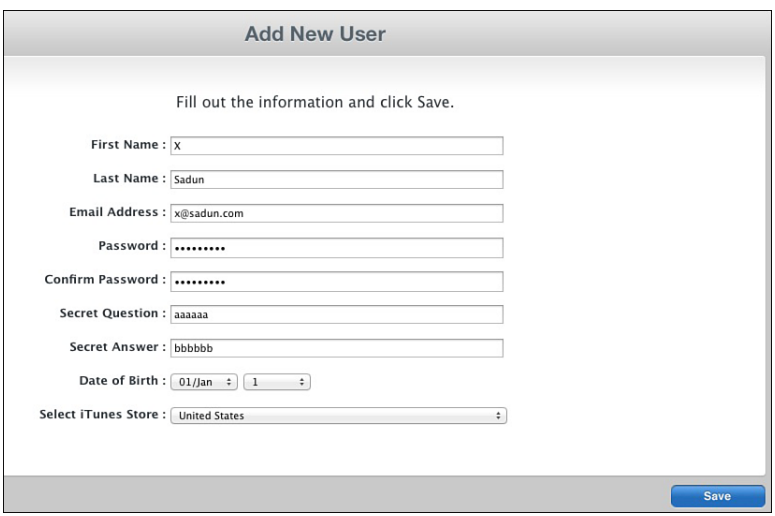


Figure 23-2 Add new test users in iTunes Connect by filling out this form.

From the Library of Eric Berna

Creating New In-App Purchase Items

Each in-application purchase item must be registered at iTunes Connect. To create a new purchase, navigate to Manage Your Apps and select any application that is either already approved or in-review. Click Manage In-App Purchases and then click Create New.

iTunes Connect prompts you to select an in-app purchase type. You can choose from

- **Consumable**—Create a purchase that is used up by the user during the normal use of the app, such as a power boost, extra bullets, or fish food. Consumables may expire, and they naturally decrease over time during use.
- **Non-Consumable**—Create a one-time purchase item, such as a specialty rifle, access to additional levels or a full-featured interface, or other unlockables. Non-consumables do not expire or decrease with use.
- **Auto-Renewable Subscriptions**—These allow users to purchase content for a set period of time, which auto-charges the user at the end of each time period until the user opts out of the service. Auto-Renewable subscriptions are automatically propagated to all devices registered to the same Apple ID account.
- **Non-Renewing Subscriptions (*not recommended*)**—This is an older feature that has mostly been replaced by Auto-Renewable Subscriptions. A Non-Renewing Subscription times out as the Auto-Renewable does, but does not automatically re-subscribe the user. Instead, the user must re-opt in to the service and materials do not automatically propagate to all Apple ID-registered devices.

Select the kind of in-app purchase you want to create, and then fill out the details for the new purchase item. You will start by entering basic details about the purchase, setting its pricing and availability, and entering a screenshot for review.

Filling Out the Details Section

The details section consists of a reference name, a product ID, and a list of languages. Figure 23-3 shows what this screen looks like. The reference name should describe the purchase in an easily understood manner. The Product ID provides unique identifier for looking up the product in the App Store database.

Reference Name

The reference name used to provide a name for iTunes Connect's search results and in the application's Top In-App Purchase section in App Store. So, enter a meaningful name (e.g., "Unlock Sky Master Level 3 Purchase") that helps you and others know what the item is and how it is used in your application.

Product ID

The product ID is a unique identifier, similar to the application identifier used for your app. As a rule, I use my application ID and append a purchase name to that such as `com.sadun.scanner.optionalDisclosure`. You need this identifier to query the store

From the Library of Eric Berna

and retrieve details about this purchase. The same rules apply to the product ID as to application IDs. You cannot use an identifier more than once. You cannot "remove" it from App Store. Once registered, it's registered forever.

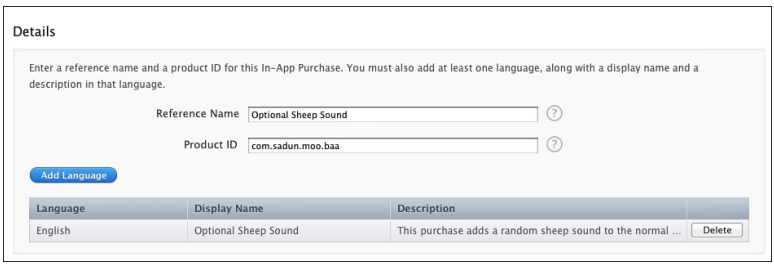


Figure 23-3 Set up the purchase details by creating a unique product ID.

You will need to add one or more localized language descriptions of the purchase as well.

Adding Localized Descriptions

Each purchasable item must be able to describe itself to your application. Keep in mind that your application is the primary consumer for this information. In the localized description, you will need to specify a display name (the name of the product that is being purchased) and description (an explanation to the user that describes what the purchase is and does).

These latter two elements are localized to specific languages. At the time of writing this book, those languages are

- English (also Australian English, Canadian English, UK English)
- Dutch
- French (also Canadian French)
- German
- Italian
- Japanese (also Korean)
- Portuguese (also Brazilian Portuguese)
- Russian
- Simplified Chinese
- Spanish (also Mexican Spanish)
- Swedish

You can create data for any or all these languages so long as you define at least one. You cannot submit a new purchase item without creating one or more name/description pairs. For developers who are primarily targeting the U.S. store, a single English entry

From the Library of Eric Berna

should cover your needs, although you may consider revisiting this with the help of professional translators at a later time.

If your application is sold worldwide, you'll likely want to mirror the existing localizations you use with your app descriptions and in-app features. If your iTunes store marketing material provides a Japanese localization, for example, and your application offers a Japanese language version, you'll want to create a Japanese-localized in-app purchase description as well. If you do not, you can still use in-app purchases, but the language will default to whatever localizations you have provided.

> **Note**
>
> Always use native speakers to localize, edit, and proof text. Google Translate is not a substitution for proper localization.

Entering Product Information

When entering the localized product information data (see Figure 23-4), keep some points in mind:

- As mentioned earlier, your application is the consumer for this information. The text you type in iTunes Connect helps create the purchase GUI that your application presents to the user.
- The user's language settings select the localization. If a localization is not available, it will default to one of the other descriptions that are provided.
- If you plan to use a simple alert sheet with a Buy/Cancel choice, keep your wording tight. Limit your verbosity. If you will use a more complex view, consider that as well.

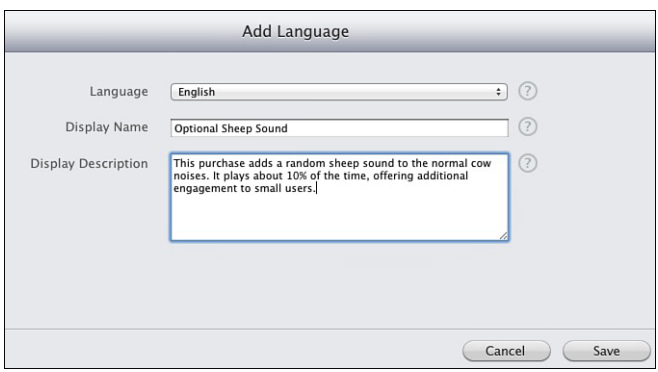


Figure 23-4 Add each language separately

From the Library of Eric Berna

No matter how you will create your GUI, remember that your description has to convey the action of purchasing as well as a description of the item being purchased—for example, "When purchased, this option unlocks this application's detail screens. These screens reveal even more data about the scanned MDNS services." A shorter description, like "Extra detail screens" or "Unlock more details," doesn't explain to users how the purchase works and what they can expect to receive.

> **Note**
>
> You can edit item display details during review at iTunes Connect. You must submit new changes for review once the purchased items have been approved.

Filling Out the Pricing Section

The pricing section specifies how a purchase is priced. Leave the Cleared for Sale item checked. The Cleared for Sale check box ensures that your applications, both development and distribution, have programmatic access to the purchase item.

> **Note**
>
> You can change the pricing tier and Cleared for Sale check box at any time during review. You must submit new changes for review once the purchased items have been approved. You cannot edit the identifier or reuse an existing identifier, nor can you change the type of product after creating the purchase item.

Submitting a Purchase GUI Screenshot

The For Review section appears at the bottom of the item sheet. You do not use this section until you have finished developing and debugging your application. You can leave the state of your product as Waiting for Screenshot until you are done testing it and are ready to upload it for review. When you are at that point, upload a screenshot into the provided field. The screenshot must show the in-app purchase in action, demonstrating the custom GUI you built.

Figure 23-5 displays the submission section. Valid pictures must be at least 320x460 pixels in size. The screenshot highlights how you have developed the purchase feature. Submit an image highlighting the purchase.

From the Library of Eric Berna

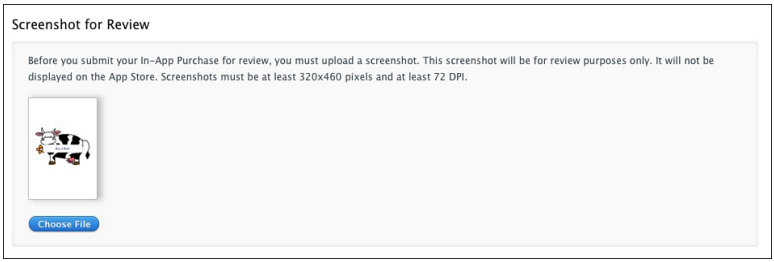


Figure 23-5 You must submit a screen shot showing your in-application purchase GUI to Apple when you are ready to have that purchase reviewed.

> **Note**
>
> A shared secret code is used when retrieving In-App Purchase receipts. Apple notes: "Without a shared secret, you will not be able to test auto-renewable In-App Purchase subscriptions in the sandbox mode. Also note that you will not be able to make them available on the App Store."

Submitting In-App Purchases Products for Review

After you have finished your sandbox testing and are confident that the application and the in-app purchasing are ready for Apple to review, you submit the purchase for its own review process. Upload your new app, return to your App's Manage In-App Purchases, select any purchase item, and click Submit for Review.

Building a Storefront GUI

Apple's StoreKit framework does not provide a built-in GUI for soliciting user purchases. You must create your own. Retrieve localized prices and descriptions from the App Store by creating `SKProductsRequest` instances. This class asks the store for that information based on the set of identifiers you provide. Each identifier must be registered at iTunes Connect as an in-app purchase item.

Allocate a new products request instance and initialize it with that set. You can add identifiers for items you've already established as well as items you're planning on adding in the future. Because each identifier is basically a string, you could create a loop that builds identifiers according to some naming scheme (e.g., com.sadun.app.sword1, com.sadun.app.sword2, etc.) to provide for future growth. This snippet searches for a single item:

From the Library of Eric Berna

```
#define PRODUCT_ID       @"com.sadun.moo.baa"

// Create the product request and start it
SKProductsRequest *productRequest =
    [[SKProductsRequest alloc] initWithProductIdentifiers:
        [NSSet setWithObject:PRODUCT_ID]];
productRequest.delegate = self;
[productRequest start];
```

When using a products request, your delegate must declare and implement the `SKProductsRequestDelegate` protocol. This consists of three simple callbacks. Listing 23-1 shows these callback methods for a simple application. When a response is received, this code looks for a product (only one was requested, per the code snippet right before this paragraph) and retrieves its localized price and description.

It then builds a simple alert using the description as the alert text and two buttons (the price and "No Thanks"). This alert functions as a basic purchase GUI. Figure 23-6 shows what this alert might look like.

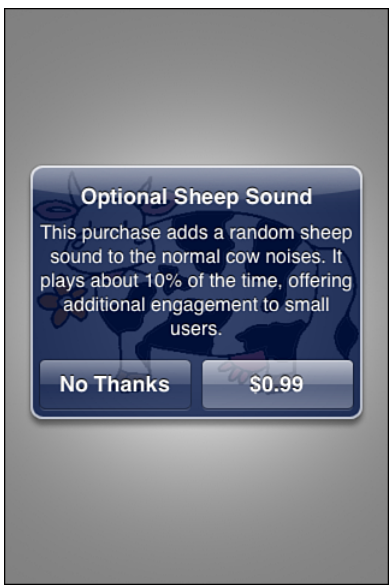


Figure 23-6 This alert is built from the localized description retrieved from the App Store.

Note

StoreKit will not work if you are not connected to the network in some way. Refer to Chapter 15, "Networking," to find recipes that help check for network access.

From the Library of Eric Berna

Listing 23-1 **Products Request Callback Methods**

```
- (void)request:(SKRequest *)request
    didFailWithError:(NSError *)error
{
    NSLog(@"Error: Could not contact App Store properly: %@",
        error.localizedFailureReason);
}

- (void)requestDidFinish:(SKRequest *)request
{
    NSLog(@"Request finished");
}

- (void)productsRequest:(SKProductsRequest *)request
    didReceiveResponse:(SKProductsResponse *)response
{
    // Find a product
    SKProduct *product = [[response products] lastObject];
    if (!product)
    {
        NSLog(@"Error: Could not find matching products");
        return;
    }

    // Retrieve the localized price
    NSNumberFormatter *numberFormatter =
        [[NSNumberFormatter alloc] init];
    [numberFormatter setFormatterBehavior:
        NSNumberFormatterBehavior10_4];
    [numberFormatter setNumberStyle:NSNumberFormatterCurrencyStyle];
    [numberFormatter setLocale:product.priceLocale];
    NSString *priceString =
        [numberFormatter stringFromNumber:product.price];

    // Show the information
    UIAlertView *alert = [[UIAlertView alloc]
        initWithTitle:product.localizedTitle
        message:product.localizedDescription
        delegate:nil
        cancelButtonTitle:@"No Thanks"
        otherButtonTitles: priceString, nil];
    ModalAlertDelegate *delegate =
        [ModalAlertDelegate delegateWithAlert:alert];
    BOOL answer = [delegate show];

    NSLog(@"User %@ buy", answer ? @"will" : @"will not");
```

From the Library of Eric Berna

```
    if (!answer)
        return;

    // Carry out the purchase here

}
```

Purchasing Items

To purchase items from your application, start by adding a transaction observer. For example, you can do this is in your application delegate's finished-launching-with-options method or in your primary view controller's load view or view did load method. Whichever observer you use, make sure that class declares and implements the `SKPaymentTransactionObserver` protocol.

```
[[SKPaymentQueue defaultQueue] addTransactionObserver:someClassInstance];
```

With an observer in place, you can use the GUI from Listing 23-1 to begin the actual purchase.

```
// Ready to purchase the item
SKPayment *payment =
    [SKPayment paymentWithProduct:product];
[[SKPaymentQueue defaultQueue] addPayment:payment];
```

StoreKit prompts the user to confirm the in-app purchase, as shown in Figure 23-7, and then takes over the purchase process. Users may need to log in to an account before they can proceed.

From the Library of Eric Berna

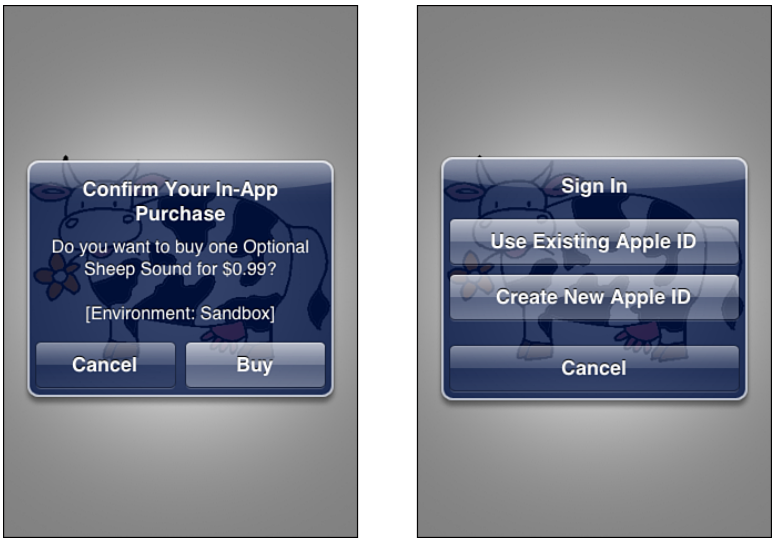


Figure 23-7 Users must confirm the purchase after moving past your user interface into the actual App Store/StoreKit purchasing system. These screenshots were taken during sandbox testing.

Signing Out of Your iTunes Account for Testing

To use the test accounts you set up in iTunes Connect, be sure to sign out of your current, real account. Launch the Settings application, choose the Store preferences, and click Sign Out.

As mentioned earlier, *do not* attempt to sign in again with your test account credentials. Just quit out of Settings and return to your application. After clicking Buy, you are prompted to sign in to iTunes. At that prompt, choose Use Existing Account and enter your account details.

Regaining Programmatic Control After a Purchase

The payments transaction observer receives callbacks based on the success or failure of the payment process. Listing 23-2 shows a skeleton for responding to both finished and unfinished payments. After the user finishes the purchase process, the transaction will have succeeded or failed. On success, perform whatever action the user has paid for, whether by downloading data or unlocking features.

From the Library of Eric Berna

Listing 23-2 **Responding to Payments**

```
- (void)paymentQueue:(SKPaymentQueue *)queue
    removedTransactions:(NSArray *)transactions
{
}

- (void) completedPurchaseTransaction:
    (SKPaymentTransaction *) transaction
{
    // PERFORM THE SUCCESS ACTION THAT UNLOCKS THE FEATURE HERE

    // For example, update user defaults, the keychain, or local variables
    [[NSUserDefaults standardUserDefaults] setBool:YES forKey:@"baa"];
    [[NSUserDefaults standardUserDefaults] synchronize];
    hasBaa = YES;

    // Update GUI accordingly
    if (purchaseButton)
    {
        [purchaseButton removeFromSuperview];
        purchaseButton = nil;
    }

    // Provide some feedback to the user that the transaction
    // was successful, e.g.
    AudioServicesPlaySystemSound(baasound);

    // Finish transaction
    [[SKPaymentQueue defaultQueue] finishTransaction: transaction];

    // Always say thank you.
    UIAlertView *okay = [[UIAlertView alloc]
        initWithTitle:@"Thank you for your purchase!"
        message:nil delegate:nil cancelButtonTitle:nil
        otherButtonTitles:@"OK", nil];
    ModalAlertDelegate *delegate =
        [ModalAlertDelegate delegateWithAlert:okay];
    [delegate show];
}

- (void) handleFailedTransaction: (SKPaymentTransaction *) transaction
{
    // Process any transaction error
    if (transaction.error.code != SKErrorPaymentCancelled)
    {
        UIAlertView *okay = [[UIAlertView alloc]
```

From the Library of Eric Berna

```
            initWithTitle:@"Transaction Error. Try again later."
            message:nil delegate:nil cancelButtonTitle:nil
            otherButtonTitles:@"OK", nil];
        ModalAlertDelegate *delegate =
            [ModalAlertDelegate delegateWithAlert:okay];
        [delegate show];
    }

    // Complete the pending transaction
    [[SKPaymentQueue defaultQueue] finishTransaction: transaction];

    // Restore the GUI
    [imageView addGestureRecognizer:longPressRecognizer];
}

- (void)paymentQueue:(SKPaymentQueue *)queue
    updatedTransactions:(NSArray *)transactions
{
    for (SKPaymentTransaction *transaction in transactions)
    {
        switch (transaction.transactionState)
        {
            case SKPaymentTransactionStatePurchased:
            case SKPaymentTransactionStateRestored:
                [self completedPurchaseTransaction:transaction];
                break;
            case SKPaymentTransactionStateFailed:
                [self handleFailedTransaction:transaction];
                break;
            case SKPaymentTransactionStatePurchasing:
                [self repurchase];
                break;
            default: break;
        }
    }
}
```

Registering Purchases

You can use any of a number of approaches to register purchases. You can synchronize with a Web server, create local files, set user defaults, or add keychain entries. The solution you choose is left up to you. Just don't lose track of purchases. Once a user buys an unlockable feature, subscription, or data, you must guarantee that your application supplies the promised element or elements.

From the Library of Eric Berna

It's easiest to unlock features through user preferences. This snippet creates a new default, indicating that the user has purchased a disclosure feature. Upon completing the purchase, the code updates the user defaults database and hides the "buy" button from the interface.

```
// Update user defaults
[[NSUserDefaults standardUserDefaults] setBool:YES
    forKey:@"upgradedScanner"];
[[NSUserDefaults standardUserDefaults] synchronize];

// Hide "buy" button
self.navigationItem.leftBarButtonItem = nil;
```

The application can check for this preference each time it launches.

Security and Piracy

For the most part, users cannot and will not hack their way into your application to update preferences settings by hand. The application is sandboxed (other applications cannot access your files), and the sandbox Library data cannot be edited from the Macintosh backup system except via Xcode. It is possible to exploit user defaults in jailbroken systems, if you use just a simple preference like this. For anyone worried about piracy, consider a more secure approach.

If you have any concerns, consider using some sort of verifiable authentication key rather than a standard Boolean value. Use the system keychain (refer to Chapter 15). The keychain provides a secure data store that cannot easily be manipulated from the jailbroken iOS command line.

A simple example of storing the purchase on the keychain would be a routine like this. (See Apple's Generic Keychain Wrapper sample code.) Alternatively use the credential spaces demonstrated in Chapter 15.

```
-(void) unlockMaxGameLevels
{
    KeychainItemWrapper *wrapper = [[KeychainItemWrapper alloc]
        initWithIdentifier:@"CustomGameApp" accessGroup:nil];
    [wrapper setObject:@"MaxUnlocked" forKey:(id)kSecValueData];
    [wrapper release];
}
```

Using the keychain provides the additional benefit that the data stored here will survive an application being deleted and then later reinstalled.

When you use an offsite server to register and authenticate purchases, make sure to echo those settings on the device. Users must be able to use their applications regardless of whether they have network access. A local setting (e.g., "Service enabled until 6 June 2011") lets the application run and provide proper feedback, even when a subscribed service is inaccessible.

From the Library of Eric Berna

Never assume the user is tied to any specific device, and be aware that Apple frowns on tracking devices by their identifiers, be it UDIDs or MAC addresses. Apple has deprecated the `UIDevice` class's `uniqueIdentifier` property and instructs developers to create unique identifiers that are specific to your application instead.

```
NSString *uuid()
{
    CFUUIDRef theUUID = CFUUIDCreate(NULL);
    NSString *uuidString =  (__bridge_transfer NSString *)
        CFUUIDCreateString(NULL, theUUID);
    CFRelease(theUUID);
    return uuidString;
}
```

Restoring Purchases

Purchases may be restored on a device where an application was uninstalled and then reinstalled, or where an application was installed on a second device associated with the same iTunes/Apple ID account. If a customer's account is used on multiple devices, like a family with a total of five iPhones, iPads, and iPods, a purchase by any of the devices with those account credentials allows all the devices to download that purchase with no additional charge.

StoreKit allows you to restore purchases, which is particularly important for consumable and subscription items where you do not want to allow the user to repurchase an already valid item. In the case of a nonconsumable item, the user can repurchase without cost ad infinitum. For these nonconsumable items, you can simply submit your purchase request. The App Store interface will present a window informing the user that he has already purchased this item, and that he can download it again for free.

To restore purchases associated with an iTunes account, call `restoreCompletedTransactions`. This works just like adding a payment and involves the same callbacks. To catch a repurchase separately from a purchase, check for `SKPaymentTransactionStateRestored` as the payment transaction state, as shown in Listing 23-2.

```
- (void) repurchase
{
    // Repurchase an already purchased item
    [[SKPaymentQueue defaultQueue] restoreCompletedTransactions];
}
```

That's because purchase events provide not one but two possible successful outcomes. The first is a completed purchase. The user has bought the item and the payment has finished processing. The second is the restored purchase described here. Make sure that your payment queue handler looks for both states.

There's a loophole here. Consider providing a consumable purchase item such as a credit to send a fax. Should the user uninstall the application and then reinstall, any repurchase functionality may restore an asset that has already been used. Applications with

From the Library of Eric Berna

consumable products must be designed with more thought for the security infrastructure and demand server-side accounting that keeps track of user credits and consumed assets.

Go ahead and restore purchases but ensure that those purchases properly coordinate with your server database. As you'll read about in the following section, Apple provides a unique identifier for each purchase by way of a purchase receipt. A repurchased item retains that original identifier, allowing you to distinguish between new purchases and restored ones.

Purchasing Multiple Items

Users can purchase more than one copy of consumable items and subscriptions. Set the `quantity` property for a payment to request a multiple purchase. This snippet adds a payment request for three copies of a product, perhaps adding three months to a subscription, 3,000 hit points to a character, or so forth.

```
SKPayment *payment = [SKPayment paymentWithProduct:product];
payment.quantity = 3;
[[SKPaymentQueue defaultQueue] addPayment:payment];
```

Handling Delays in Registering Purchases

If your purchase connects with a server and you cannot complete the purchase registration process, *do not finalize the transaction*. Do not call `finishTransaction:` until you are guaranteed that all establishment work has been done for your customer.

Should you fail to set up your user with his or her newly purchased items before the application is quit, that's okay. The transaction remains in the purchase queue until the next time the application launches. You are given another opportunity to try to finish your work.

Validating Receipts

A successful purchase transaction contains a receipt. This receipt, which is sent in raw `NSData` format, corresponds to an encoded JSON string. It contains a signature and purchase information. Here is a sample receipt from one of my purchases:

```
{
      "signature" =
"AbtAgJQlIPicxP/g4ubwT/noCER4jE+LuGNfxfy++DsiEUrdOYNcf6GqljT+/qDlLCvSZUnWGG7YrACLD
FQRREftjNDmkgekbErdP8uI9IAN0sH6vkHx5sc/2p9hHRbG6AY/CDDj11g+esLRe8HYGxCBaHlIMa+o/ZK
tHr3Rl+jUMIIDUzCCAjugAwIBAgIIZRSRTdlYBLUwDQYJKoZIhvcNAQEFBQAwfzELMAkGA1UEBhMCVVMxE
zARBgNVBAoMCkFwcGxlIEluYy4xJjAkBgNVBAsMHUFwcGxlIENlcnRpZmljYXRpb24gQXV0aG9yaXR5MTM
wMQYDVQQDDCpBcHBsZSBpVHVuZXMgU3RvcmUgQ2VydGlmaWNhdGlvbiBBdXRob3JpdHkwHhcNMDkwNjE1M
jIwNTU2WhcNMTQwNjE0MjIwNTU2WjBkMSMwIQYDVQQDDBpQdXJjaGFzZVJlY2VpcHRDZXJ0aWZpY2F0ZTE
bMBkGA1UECwwSQXBwbGUgaVR1bmVzIFN0b3JlMRMwEQYDVQQKDApBcHBsZSBJbmMuMQswCQYDVQQGEwJVU
zCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEAytGMXZy3gitJ2JMKFojSDynC/9yYezyn9HBX+u3/3Vc
pWE2XhcgGKYqNBA1+AewOzrKO774OsokTu4qymEx10ph8UTmsZewB0ESMHBEjF7FN6/HccsQUYC3WagrHn
T12HG2Ih0OAm/ZhpWzj0HS4m813LpIyo00sewMvMNL2hkcCAwEAAaNyMHAwDAYDVR0TAQH/BAIwADAfBgN
VHSMEGDAWgBQ2HejinYLSARi1MmsO10MLkVhDOjAOBgNVHQ8BAf8EBAMCB4AwHQYDVR0OBBYEFKmDg/IZS
```

From the Library of Eric Berna

```
MU+ElcIFMzNo36ZXyT1MBAGCiqGSIb3Y2QGBQEEAgUAMA0GCSqGSIb3DQEBBQUAA4IBAQARpJs+O2Y3gL8
gHdASkrfZHFpwINd1VcB5VF5LkVpnFz63zylA/3cGIDG91b/d5NIwZjkVt4Bgvd62o/mCbzCsWiNfSKTJV
FK1D78BDQoSO2oHTuQuz1BR7xzNHxQZ90zUS6ZX9SC8N3g3A1jEtAyDhZNB+CRBBXLwZdnBUeBsT9QLpjv
TnekZcGTnU08zfCjGF3eBJEu9eP6WgexK1xMSp72kEOmYbn6yTi3D4YrcYx4Q3n/57VBP2en8qXWeP5oHD
sLTGzLRsWdoB3VxJLrF2ivL8JS8zqC0qyac452pN6xunRuzyyfpaqzQL12BzFEe44xna2byektSbtquA5L
NAAAAAA==";
      "purchase-info" =
"ewoJIml0ZW0taWQiID0gIjMzMDI5MjgwNiI7Cgkib3JpZ2luYWwtdHJhbnNhY3Rpb24taWQiID0gIjEwM
DAwMDAwMDAwNTIyOTMiOwoJInB1cmNoYXNlLWRhdGUiID0gIjIwMDktMDktMDQgMTU6MzU6MjYgRXRjL0d
NVCI7CgkicHJvZHVjdC1pZCIgPSAiY29tLnNhZHVuLnNjYW5uZXIuZGlzY2xvc3VyZTIiOwoJInRyYW5zY
WN0aW9uLWlkIiA9ICIxMDAwMDAwMDAwMDUyMjkzIjsKCSJxdWFudGl0eSIgPSAiMSI7Cgkib3JpZ2luYWw
tcHVyY2hhc2UtZGF0ZSIgPSAiMjAwOS0wOS0wNCAxNTozNToyNiBFdGMvR01UIjsKCSJiaWQiID0gImNvb
S5zYWR1bi5TY2FubmVyIjsKCSJidnJzIiA9ICIxLjAiOwp9";
      "pod" = "100";
      "signing-status" = "0";
}
```

Apple strongly recommends that you validate all receipts with their servers to prevent hacking and ensure that your customers actually purchased the items they are requesting. Listing 23-3 shows how.

You must POST a request to one of Apple's two servers. The URL you use depends on the deployment of the application. Use buy.itunes.apple.com for production software and sandbox.itunes.apple.com for development.

The request body consists of a JSON dictionary. The dictionary is composed of one key ("`receipt-data`") and one value (a Base64-encoded version of the transaction receipt data). I normally use the CocoaDev NSData Base 64 extension (from www.cocoadev.com/index.pl?BaseSixtyFour) to convert `NSData` objects into Base64-encoded strings. CocoaDev provides many great resources for Mac and iOS developers.

A valid receipt returns a JSON dictionary similar to the following. The receipt includes the transaction identifier, a product ID for the item purchased, the bundle ID for the host application, and a purchase date. Most importantly, it returns a status.

```
{"receipt":
    {
        "item_id":"467440745",
        "original_transaction_id":"1000000008472082",
        "bvrs":"1.0",
        "product_id":"com.sadun.moo.baa",
        "purchase_date":"2011-09-23 15:18:22 Etc/GMT",
        "quantity":"1", "bid":"com.sadun.Moo",
        "original_purchase_date":"2011-09-22 20:48:46 Etc/GMT",
        "transaction_id":"1000000008535652"},
 "status":0}
```

A valid receipt always has a 0 status. Any number other than 0 indicates that the receipt is invalid.

From the Library of Eric Berna

Simply checking for the status may not be sufficient for validation. It's not too difficult to set up a proxy server to intercept calls to the validation server and return JSON `{"status":0}` to all requests. What's more, the receipt data that is sent along with the validation request can be easily deserialized into exactly the same data shown in the "receipt" portion of the JSON dictionary just shown. For that reason, always use receipt validation cautiously and as part of the overall purchase process, where it's less likely that proxy servers can override communications with Apple.

Apple requires you to define a custom "shared secret" item for auto-renewable subscriptions. This `password` is sent by your app as a second parameter for the dictionary, in addition to the `receipt-data`, i.e.

```
{
    "receipt-data" : (actual receipt bytes here)
    "password"     : (shared secret bytes here)
}
```

As before, this returns a status of 0 for valid receipts, but also indicates that the subscription is active. For non-zero values, consult Apple's StoreKit documentation to translate the value (e.g., 21007) to a real-world meaning (e.g., you sent a sandbox receipt to the production verification server).

Listing 23-3 **Checking a Receipt**

```
- (void) checkReceipt: (SKPaymentTransaction *) transaction
{
    // Retrieve the receipt data and encode it with Base 64
    NSString *receiptData =
        [transaction.transactionReceipt base64Encoding];

    // Construct a dictionary for the receipt-data
    NSDictionary *dictionary =
        [NSDictionary dictionaryWithObject:receiptData
            forKey:@"receipt-data"];

    // Translate to JSON data
    NSData *json = [NSJSONSerialization
        dataWithJSONObject:dictionary options:0 error:nil];
    if (!json)
    {
        NSLog(@"Error creating JSON receipt representation");
        return;
    }

    // Select target
    NSString *urlsting = SANDBOX ?
        @"https://sandbox.itunes.apple.com/verifyReceipt" :
        @"https://buy.itunes.apple.com/verifyReceipt";
```

From the Library of Eric Berna

```
    // Create the request
    NSMutableURLRequest *urlRequest = [NSMutableURLRequest
        requestWithURL:[NSURL URLWithString: urlsting]];
    if (!urlRequest)
    {
        NSLog(@"Error creating the URL request");
        return;
    }
    [urlRequest setHTTPMethod: @"POST"];
    [urlRequest setHTTPBody:json];

    // Post the request and retrieve the result
    NSError *error;
    NSURLResponse *response;
    NSData *result = [NSURLConnection
        sendSynchronousRequest:urlRequest
        returningResponse:&response error:&error];

    // Check any errors here, look for status information, store
    // transaction data, etc.

    // Just demonstrate by showing the result data
    NSString *resultString = [[NSString alloc]
        initWithData:result encoding:NSUTF8StringEncoding];
    NSLog(@"Receipt Validation: %@", resultString);
}
```

Summary

The StoreKit framework offers a great way to monetize your applications. As you read in this chapter, you can set up your own storefront to sell services and features from your application. Here are a few final thoughts:

- Although the entire setup and testing process for new applications may seem a little "Which came first? The chicken or the egg?," it is demonstrably possible to develop and deploy a StoreKit-based application with a minimum of headaches. It becomes easier the next time you add in-app purchases for an already-approved application.
- Always remember that there are two submission steps with IAP. Ensure that both the application and the IAP item(s) are ready for Apple to review.
- Avoid finalizing transactions until your new-purchase setup is completely, utterly, 100% done, even if that means waiting for an application relaunch. At the same time, inform the user that the purchase process is experiencing unexpected delays.

From the Library of Eric Berna

- Your methods can only request product information from in-app items that are registered to the currently running application.You cannot share requests across apps.
- Don't forget to set up the purchase observer! More heads have been banged against desks and hair pulled out over that one step than any other StoreKit issue.

From the Library of Eric Berna

This page intentionally left blank

From the Library of Eric Berna

24

GameKit

This chapter introduces various ways you can create connected gameplay with GameKit. GameKit offers various features that allow your applications to move beyond a single-player/single-device scenario using Game Center and device-to-device networking. Apple's Game Center adds a centralized service that allows your game to offer shared leader boards and internet-based matches. GameKit also provides an ad-hoc networking solution for peer-to-peer connectivity. Built on a technology called Bonjour, ad-hoc connection offers simple, no-configuration communications between devices. In this chapter, you discover how to use GameKit to build connected applications. You see how to create Game Center features. You read about adding GameKit Voice to your code for walkie-talkie–style voice chats.

Enabling Game Center

Game Center features help expand your users' gameplay experience. With Game Center, you can provide matches, leaderboards, and other enhancements that bring your user out of the boundaries of a single application and into a shared communal experience.

You must enable Game Center on an application-by-application basis, as shown in Figure 24-1. That means you need to create a unique application identifier at the iOS provisioning portal, register that application identifier at iTunes Connect by setting up a new application skeleton, and enable Game Center before you can really get down to development. Once you do, the Game Center features will become available for both testing and deployment. Chapters 21, "Push Notifications," and 23, "StoreKit: In-App Purchasing" introduced these steps, which are not duplicated here. (Refer to those chapters for step-by-step how-to coverage.)

From the Library of Eric Berna

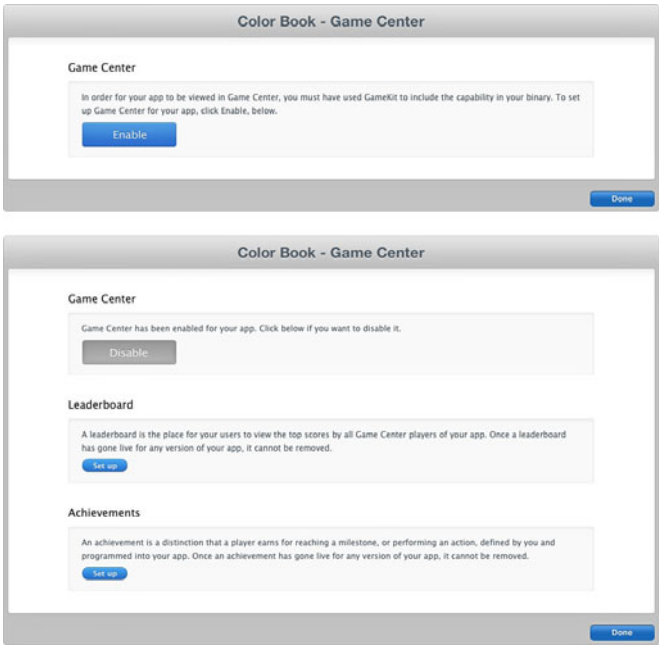


Figure 24-1 Enable Game Center on an application-by-application basis. Once established, you can set up new leaderboards and achievements, and allow your application to interact with other Game Center features.

Testing for Game Center Availability

If you plan to deploy earlier than iOS 4.1, you may want to check to see if Game Center is available on the current device. You can easily check for class existence on, for example, `GKLocalPlayer`, to infer GameKit Game Center support.

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

Testing Game Center Compliance

The easiest way to test whether your application has been properly set up for Game Center is to attempt to sign in a user. The following code snippet retrieves the shared local player singleton object and authenticates that player:

```
- (void) loadView
{
    [super loadView];
```

From the Library of Eric Berna

```
    player = [GKLocalPlayer localPlayer];
    if (player && !player.authenticated)
    {
        [player authenticateWithCompletionHandler:^(NSError *error){
            if (error)
            {
                NSLog(@"Error authenticating user: %@",
                    error.localizedFailureReason);
                return;
            }

            // User is authenticated. Do something here.
        }];
    }
}
```

You will not be able to sign in properly unless the application has been properly registered at iTunes Connect. Figure 24-2 shows the error generated when the identifier for the currently running application is not recognized as a valid Game Center app.



Figure 24-2 You can only use Game Center features with applications you have registered at iTunes Connect.

Authentication does not necessarily mean that the user must sign in each time the app is launched. Sessions can last quite a while. If the authentication remains valid (`player.authenticated`), your user will be greeted with a welcome-back notice rather than a password prompt.

Be aware that Game Center initialization can slow down app launches. Where possible, use threading to allow other elements of your application to load in the background.

> **Note**
>
> As you develop, you will test in a GameKit sandbox environment. All scores and achievements made there will not leak through to the production leaderboards. If you build ad-hoc versions of your application, they will use the production items.

From the Library of Eric Berna

Designing Leaderboards and Achievements

Game Center offers a pair of items that help to engage users further with games. Leaderboards allow users to view a shared list of top scores for your application. Achievements let users unlock goal milestones through game play. Both features are established at iTunes Connect. Create these (on a per-application basis) by clicking the Manage Game Center button in the iTunes Connect application information page. The following sections introduce the details involved in this process.

Building Leaderboards

Choose Leaderboard > Set Up > Add Leaderboard > Single Leaderboard > Choose to create a new leaderboard for your application. iTunes Connect will prompt you to specify the following information:

- **Reference Name**—Enter a human-readable private name created for your own reference. This name is not used outside iTunes Connect. For example, you might want to keep track of "Top overall point scores" or "Fastest reaction time." Keep the reference name easy to understand and simple to search for.
- **Leaderboard ID**—This is a unique string, which is used by your application to identify the specific leaderboard. For simplicity, use reverse domain naming. Append a feature name to your application identifier e.g. com.sadun.cookbook.topPoints.
- **Score Format**—Select the way the score is counted and shown. Your choices are integer (a whole number score), fixed point (a floating point number, with one, two, or three fixed decimal places), a time interval (to the minute, second, or hundredth of a second), or currency (whole units, or units and two decimal places). Choose whether to sort from low to high (higher numbers win, e.g. as in points), or high to low (lower numbers wins, e.g. as in shortest time elapsed).
- **Score Range** (optional)—Enter the lowest and highest possible scores for your application.

After entering this information, you must enter one or more localized descriptions, and then you can save the new leaderboard. Figure 24-3 shows a newly created leaderboard.

Localized Descriptions

The reference name you entered to create the leaderboard is not used when presenting that leaderboard to your users. Instead, you create localized descriptions that provide the text material and any other formatting options your application will use. Leaderboards must be localized to at least one language. Click Add Language and select a language.

Enter a real-world name for your leaderboard. This name will be used by your application and displayed to users. Keep it short and to the point, such as "Top Scores," "Best Shooters," or "Quickest Reactions."

From the Library of Eric Berna

Select a score format and enter any suffixes or prefixes used for your items. This information can vary in each localization. You can also supply an optional 512 x 512 pixel image to supplement the leaderboard. Since this image, too, is localized, you can create different images for each language you support.

Click Save to save your localization. You can revisit the description you just edited later and make any changes needed but you cannot remove the leaderboard once it goes live.



Figure 24-3 iTunes Connect allows you to create and manage leaderboards. You cannot delete leaderboards once they go live.

Creating Achievements

You build achievements by adding them in iTunes Connect, much the same way as you create leaderboards. Choose Achievements > Set Up > Add New Achievement. You must specify the following items:

- **Reference Name**—This is a human-readable private name that is not used outside iTunes Connect (e.g., "Great Start Achievement" or "100 Certified Kills").
- **Achievement ID**—This is a unique string, which is used by your application to identify the specific achievement. As with leaderboards, you'll probably want to use reverse domain naming, with your application identifier appended by some meaningful phrase (e.g., com.sadun.cookbook.greatStartAchievement).
- **Hidden**—Specify whether these items are hidden on Game Center until the user has achieved them. This does not hide the number of items; it just hides the names and descriptions. Hiding allows you to surprise and delight your user with new accomplishments that they did not expect. When not hidden, you provide emotional incentives to unlock the complete set of items and a preview of what those items are.
- **Point Value**—You can assign up to 100 points per achievement, and up to 1000 points per application across all achievements. This adds a limiting factor so you don't overwhelm your users with "You have played this game for 15 minutes and 30 seconds" achievements.

From the Library of Eric Berna

Each achievement requires at least one localized description. Click Add Language, choose a language, enter a human-readable title that will be displayed to your readers, and two descriptions. The Pre-earned Description describes the achievement before your user has unlocked it or is in the process of unlocking it; the Earned Description describes it after. For example, you might have "Prove your mettle by overcoming Master Toporek" as the pre-earned version, and "Conquered Master Toporek" as the earned one. Keep the descriptions short or they'll be automatically clipped.

Non-hidden achievements always display the pre-earned description until they are achieved. Hidden achievements will also display the pre-earned description to the user once you assign a (non-zero) percent-achieved value to it. If your user has gone 50% of the way toward "conquering the master," make sure that your per-earned text matches the presentation of that concept.

In addition to text, each achievement requires a (localized) 512 x 512 pixel art image (see Figure 24-4).



Figure 24-4 Specify the achievement elements by providing a human-readable name, a pair of descriptions for before and after the item is earned, and an image to associate with the achievement.

Accessing Leaderboards

You can easily query the entire list of leaderboard categories. Many applications have just one leaderboard for top scores. Others have leaderboards for various categories, like most accurate shooter or greatest word length. The following method retrieves the categories (e.g., com.sadun.cookbook.topPoints) and titles (e.g., Top Points) for each available leaderboard:

From the Library of Eric Berna

```
- (void) peekAtLeaderboards
{
    [GKLeaderboard loadCategoriesWithCompletionHandler:
        ^(NSArray *categories, NSArray *titles, NSError *error)
     {
         if (error)
         {
             NSLog(@"Error retrieving leaderboard categories: %@",
                 error.localizedFailureReason);
             return;
         }

         for (int i = 0; i < categories.count; i++)
         {
             NSString *category = [categories objectAtIndex:i];
             NSString *title = [titles objectAtIndex:i];
             NSLog(@"%@ : %@", category, title);
         }
     }];
}
```

The GKLeaderboard Class

Each `GKLeaderboard` instance is used to query for scores from Game Center. You can search globally for high scores or use scopes to limit the query. For example, you might specify a `playerScope` that limits the search to a player and his or her friends rather than returning all top scores outside of that group. The `range` property sets which scores to show. (The range count starts at 1, not at 0, by the way.) Normally, it defaults to the top 25 scores, but you can choose the top 10 instead. You can choose a time scope as well, to look at high scores for just the current day or week, etc.

The following method demonstrates how you might load the data for a specific category. It requests the top ten scores within the last week, and then attempts to load the game center aliases for each player. If it fails at doing so, it falls back to using the `GKScore`'s player ID property.

```
- (void) peekAtLeaderboardForCategory: (NSString *) category
{
    GKLeaderboard *leaderBoard = [[GKLeaderboard alloc] init];
    leaderBoard.category = category;

    // Top ten scores (default is 25) within the last week
    leaderBoard.range = NSMakeRange(1, 10);
    leaderBoard.timeScope = GKLeaderboardTimeScopeWeek;

    [leaderBoard loadScoresWithCompletionHandler:
        ^(NSArray *scores, NSError *error){
```

From the Library of Eric Berna

```
        if (error)
        {
            NSLog(@"Error retrieving leaderboard scores: %@",
                error.localizedFailureReason);
            return;
        }

        NSMutableArray *array = [NSMutableArray array];
        for (GKScore *score in scores)
            [array addObject:score.playerID];

        // Convert the array of identifiers into an array
        // of fully populated GKPlayer objects
        [GKPlayer loadPlayersForIdentifiers:array
            withCompletionHandler:^(NSArray *players, NSError *error){
            if (error)
            {
                // Report only with player ids
                for (GKScore *score in scores)
                    NSLog(@"[%2d] %@: %@ (%@)",
                        score.rank, score.playerID,
                        score.formattedValue, score.date);
                return;
            }
            for (int i = 0; i < scores.count; i++)
            {
                // Report with actual player names
                GKPlayer *aPlayer = [players objectAtIndex:i];
                GKScore *score = [scores objectAtIndex:i];
                NSLog(@"[%2d] %@: %@ (%@)",
                    score.rank, aPlayer.alias,
                    score.formattedValue, score.date);
            }
        }];
    }];
}
```

Leaderboard View Controller

Normally, you do not access leaderboard data directly. GameKit provides a much friendlier already built view controller to show to your user (see Figure 24-5). The following methods demonstrate how you can create, present, and dismiss the leaderboard modally. Make sure your calling class declares the `GKLeaderboardViewControllerDelegate` protocol.

From the Library of Eric Berna

```
#define GKCATEGORY    @"com.sadun.cookbook.topPoints"

- (void)leaderboardViewControllerDidFinish:
    (GKLeaderboardViewController *)viewController
{
    [self dismissModalViewControllerAnimated:YES];
}

- (void) showLeaderboard
{
    GKLeaderboardViewController *leaderboardController =
        [[GKLeaderboardViewController alloc] init];

    leaderboardController.category = GKCATEGORY;
    if (leaderboardController)
    {
        leaderboardController.leaderboardDelegate = self;
        [self presentModalViewController: leaderboardController animated: YES];
    }
}
```



Figure 24-5 GameKit offers preconfigured leaderboard view controllers that you can easily access from your application.

From the Library of Eric Berna

Submitting Scores

Submit scores by creating new `GKScore` instances and setting their value. You can specify the category directly, as the following method does, or you can just call `init` and let Game Center use the default leaderboard instead:

```
- (void) submitScore: (id) sender
{
    NSNumber *userScore = [self requestScore];
    if (!userScore) return;

    GKScore *score = [[GKScore alloc] initWithCategory:GKCATEGORY];
    score.value = userScore.intValue;
    [score reportScoreWithCompletionHandler:^(NSError *error){
        if (error)
        {
            NSLog(@"Error submitting score to game center: %@",
                error.localizedFailureReason);
            return;
        }

        NSLog(@"Success. Score submitted.");
    }];
}
```

Unlocking Achievements

Each achievement can be unlocked and/or reset by your application as needed. Because achievements can be partially attained, you can assign a `percentComplete` value to each achievement. These will be displayed as partial achievements (with the pre-achieved text description) in the GameKit achievement view controller.

To create and present this controller (see Figure 24-6), make sure your primary class declares both `GKAchievementViewControllerDelegate` and `UINavigationControllerDelegate` protocols. The following methods present and dismiss the controller:

```
- (void) achievementViewControllerDidFinish:
    (GKAchievementViewController *)viewController
{
    [self dismissModalViewControllerAnimated:YES];
}

- (void) showAchievements
{
    GKAchievementViewController *achievementController =
        [[GKAchievementViewController alloc] init];
    if (achievementController)
    {
```

From the Library of Eric Berna

```
        achievementController.achievementDelegate = self;
        [self presentModalViewController:achievementController
            animated:YES];
    }
}
```



Figure 24-6 GameKit's prebuilt Achievement Controller allows your user to view each item that he or she has unlocked.

> **Note**
>
> You will likely want to create a special-purpose class to handle achievement semantics rather than spreading achievement code throughout your game. It's easy to intermingle Game Center code with application semantics and end up with a giant jumble. Keep good coding practices and MVC design patterns in mind when working with Game Center elements, just as you would with other application development tasks.

Checking Achievements

The following method calls home to Game Center, typically at the start of an application run, to see which achievements have been activated, and at what level of completion. This example method replaces an Unlock button, which unlocks the achievement with a Reset button if the achievement has already been unlocked.

From the Library of Eric Berna

```
- (void) checkAchievement
{
    // Default to unlock unless it's already unlocked
    self.navigationItem.rightBarButtonItem =
        BARBUTTON(@"Unlock", @selector(unlockAchievement));

    [GKAchievement loadAchievementsWithCompletionHandler:
        ^(NSArray *achievements, NSError *error)
    {
        if (error)
        {
            NSLog(@"Error loading achievements: %@",
                error.localizedFailureReason);
            return;
        }

        for (GKAchievement *achievement in achievements)
        {
            NSLog(@"Achievement: %@ : %f",
                achievement.identifier, achievement.percentComplete);
            if ([achievement.identifier isEqualToString:GKBEGINNER])
                self.navigationItem.rightBarButtonItem =
                    BARBUTTON(@"Reset", @selector(resetAchievements));
        }
    }];
}
```

Reporting Achievements (Unlocking)

To unlock an achievement, you report it to Game Center. If the report errors out and is not successfully updated at Game Center, make sure you set up some fallback for a future repeat attempt so the user does not lose his or her progress. In the following method, the `showsCompletionBanner` allows the user to be rewarded with a visual update, as shown in Figure 24-7:

```
- (void) unlockAchievement
{
    GKAchievement *achievement =
        [[GKAchievement alloc] initWithIdentifier: GKBEGINNER];
    if (achievement)
    {
        achievement.percentComplete = 100.0f;
        achievement.showsCompletionBanner = YES;
        [achievement reportAchievementWithCompletionHandler:^(NSError *error)
         {
             if (error)
```

From the Library of Eric Berna

```
            {
                NSLog(@"Error reporting achievement: %@",
                    error.localizedFailureReason);

                // Make sure to try again later in real
                // world deployment!!!

                return;
            }


              // Achievement is now unlocked

        }];
    }
}
```

Each achievement report represents a network access at a minimum. Although reporting achievements is not especially expensive, it's not something you'll want to overuse in your applications. You don't want requests stumbling over each other, and you don't really need to update an achievement many times a second as the user makes a tiny percentage of progress toward some goal. Instead, think about where achievement updates make sense rather than reflexively sticking them into code wherever the user has any state change.

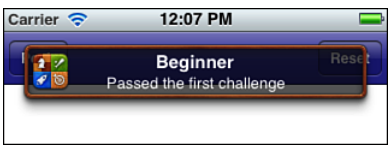


Figure 24-7 Completion banners let users know when they have unlocked achievements.

Resetting Achievements

At times, you may want to allow users to reset their achievements and start the game from scratch. They can re-earn each achievement and re-live the fun a second time around, even with the same Game Center ID.

The following method shows a typical way you'd approach this. It requests the reset and provides a completion handler to respond to the user about the success or failure of the request. Obviously, you'll want to confirm resets with the user—and you may want to provide a way to undo that reset by resubmitting achievements if needed.

```
- (void) resetAchievements
{
    [GKAchievement resetAchievementsWithCompletionHandler:
```

From the Library of Eric Berna

```
        ^(NSError *error)
    {
        if (error)
        {
            NSLog(@"Error resetting achievements: %@",
                error.localizedFailureReason);
            return;
        }

        // Achievements are now reset

    }];
}
```

Matchmaking: Multiplayer GameKit Matches

GameKit enables your users to solicit matches with other players, whether those players are friends or anonymous other persons. The matchmaker view controller handles both specific invitations and general random game matches. To get started with a matchmaking session, create a new match request. Specify how many players are required for gameplay and how many players can be handled total. For Game Center gameplay, you can create matches for two to four players. Hosted matches on your own servers can allow up to 16 players at once. This chapter does not cover hosted matches outside of Game Center. The following snippet creates a request for a basic two-player game:

```
- (void) startMatch
{
    // Clean up any previous game
    sendingView.text = @"";
    receivingView.text = @"";

    // This is not a hosted match, which allows up to 16 players
    GKMatchRequest *request = [[GKMatchRequest alloc] init];
    request.minPlayers = 2; // Between 2 and 4
    request.maxPlayers = 2; // Between 2 and 4

    GKMatchmakerViewController *mmvc =
        [[GKMatchmakerViewController alloc]
            initWithMatchRequest:request];
    mmvc.matchmakerDelegate = self;
    mmvc.hosted = NO;

    [self presentModalViewController:mmvc animated:YES];
}
```

From the Library of Eric Berna

Make sure your primary class declares the `GKMatchmakerViewControllerDelegate` protocol. Once you present the controller, it's up to the user to choose whether to invite friends or use Game Center's auto-match.

Matchmaker Required Methods

The delegate protocol requires the following two methods. These deal with the user cancelling the controller and with a failure to connect to Game Center. Here's how you might implement those, making sure that both methods dismiss the modal view controller:

```
 - (void) matchmakerViewControllerWasCancelled:
    (GKMatchmakerViewController *)viewController
{
    [self dismissModalViewControllerAnimated:YES];
}

- (void) matchmakerViewController:(GKMatchmakerViewController *)viewController
    didFailWithError:(NSError *)error
{
    [self dismissModalViewControllerAnimated:YES];
    NSLog(@"Error creating match: %@",
        error.localizedFailureReason);
}
```

Starting a Match

In addition to the required callback methods, you must implement the (optional) `matchmakerViewController:didFindMatch:` callback in order to start a game. (This callback is for Game Center-hosted games. For your own server-hosted games, implement the players-found callback instead.) The match-found delegate method is called when there's been a successful match made between your application's current player and an external player on another device.

Upon finding a match, start by checking to see if you've already started playing—sometimes, race conditions mean that you've already set up another match first. Otherwise, dismiss the matchmaker controller, save the match to a local instance variable, and set the match's delegate. You'll need to hold onto the match so you can use that object to send data to other players.

The delegate declares the `GKMatchDelegate` protocol. It receives data and status updates from the other parties in the game. It handles the actual gameplay once the match has been made.

```
- (void) matchmakerViewController:
    (GKMatchmakerViewController *)viewController
    didFindMatch:(GKMatch *)aMatch
{
```

From the Library of Eric Berna

```
    // Already playing. Ignore.
    if (matchStarted)
        return;

    if (viewController)
    {
        [self dismissModalViewControllerAnimated:YES];
        match = aMatch;
        match.delegate = self;
        self.navigationItem.rightBarButtonItem = nil;
    }

    // Normal matches now wait for player connection

    // Invited connections may be ready to go now. If so, begin
    if (!matchStarted && !match.expectedPlayerCount)
    {
        // Start game!
        [self activateGameGUI];
    }
}
```

Starting the Game

After a match has been made, you should wait until the match's `expectedPlayerCount` goes down to zero before you start game play. Each time a player connects, the number decreases by one. When it hits zero, all players are connected and your game can start the actual match.

There are two points, programmatically, at which the count may hit zero, at which your game can get going. It may take place in the matchmaker's match-found callback, or it may take place in the match delegate's `match:player:didChangeState:` callback. Your code should check in both locations. Typically, invited games go live in the matchmaker callback and auto-match games go live in the player state callback, when player state changes to connected. This trips up many developers when they first implement matching in their apps. The sample code that accompanies this chapter demonstrates both situations.

Creating an Invitation Handler

Your application can decide in advance how it will automatically handle invitation requests. By establishing an invitation handler, you create a code block that will be executed whenever the shared matchmaker object receives an invitation. This `inviteHandler` property of the shared matchmaker should be set as soon as your application is launched, either in the application delegate (typically) or in the primary view controller as it loads.

From the Library of Eric Berna

This handler takes two arguments, an invitation and an array of players. When your game receives a direct invitation, the `invitation` parameter is used, and set to some non-nil value. When you detect an invitation, the match has already been established; you should not create a new match request. Instead, just present a matchmaker view controller pre-loaded with the invitation details. Your user then waits for the host to start the game. The view controller offers a *Cancel* option, which allows the user to dismiss the view controller and cancel the invitation. Only the host will see a *Play Now* button.

When the `playersToInvite` parameter is non-nil, it indicates that this match was launched on-device at the Game Center. This is equivalent to requesting a match from inside the app so the application should build and present the standard matchmaker controller, as it does in the following method:

```
- (void) addInvitationHandler
{
    [GKMatchmaker sharedMatchmaker].inviteHandler =
        ^(GKInvite *invitation, NSArray *playersToInvite)
    {
        // Clean up any in-progress game and change the focus
        // to handling the invitation.
        [self finishMatch];

        if (invitation)
        {
            GKMatchmakerViewController *mmvc =
                [[GKMatchmakerViewController alloc]
                    initWithInvite:invitation];
            mmvc.matchmakerDelegate = self;
            [self presentModalViewController:mmvc animated:YES];
        }
        else if (playersToInvite)
        {
            GKMatchRequest *request = [[GKMatchRequest alloc] init];
            request.minPlayers = 2;
            request.maxPlayers = 2; // 2-player matches for this example
            request.playersToInvite = playersToInvite;
            GKMatchmakerViewController *mmvc =
                [[GKMatchmakerViewController alloc]
                    initWithMatchRequest:request];
            mmvc.matchmakerDelegate = self;
            [self presentModalViewController:mmvc animated:YES];
        }
    };
}
```

Note

The `finishMatch` method called here may be as simple as sending `disconnect` to the current match object and resetting the game GUI to its initial state.

From the Library of Eric Berna

Managing Match State

After establishing a match, it is the match delegate's role to manage changes in the state of the match. The two primary fail states include a failed match (connection to Game Center) and a failed player connection (connection from Game Center to another player). For a failed match, you'll want to alert the user, clean up your ongoing gameplay and revert to a pre-match "ready to play" state, as is shown in the following method:

```
- (void)match:(GKMatch *) aMatch didFailWithError:(NSError *)error
{
    NSLog(@"Match failed: %@",
        playerID, error.localizedFailureReason);
    [self cleanupGUI];

    // Alert the user. I include the error here because it can
    // provide useful context. You may choose otherwise.
    UIAlertView *alert = [[UIAlertView alloc]
        initWithTitle:@"Lost Game Center connection"
        message:error.localizedFailureReason
        delegate:nil cancelButtonTitle:nil
        otherButtonTitles:@"Okay", nil];
    [alert show];
}
```

For lost players, you can either treat the game as ended, as is done in the following method, or you might want to alert the user and wait for the other player to re-connect.

```
- (void)match:(GKMatch *) aMatch
    connectionWithPlayerFailed:(NSString *)playerID
    withError:(NSError *)error
{
    NSLog(@"Connection failed with player %@: %@",
        playerID, error.localizedFailureReason);
    [self cleanupGUI];
}
```

To simplify re-connection, implement the optional should-reinvite delegate method. When this method returns `YES`, Game Center will automatically re-invite the player whose connection was lost.

```
- (BOOL)match:(GKMatch *) aMatch
    shouldReinvitePlayer:(NSString *)playerID
{
    return YES;
}
```

From the Library of Eric Berna

Player State Changes

Although disconnect notices are generated when a player's device goes to sleep or they lose their Internet connection, players can change state in a more orderly fashion as well. For example, a player might tap a Quit button, with the application sending `disconnect` to the match object. This sends a proper state change before disconnecting and does not invoke the reinvite query that a lost connection does.

Your application will also receive state changes when players first connect. This is where you should check the current match's expected player count, launching the game when the count goes down to zero.

The following method handles state callbacks for both connect and disconnect events, showing an example of how you might react to both of these state changes:

```
- (void)match:(GKMatch *) aMatch
    player: (NSString *) playerID
    didChangeState: (GKPlayerConnectionState) state
{
    if (state == GKPlayerStateDisconnected)
    {
        NSLog(@"Player %@ disconnected", playerID);
        [match disconnect];
        [self cleanupGUI];
    }
    else if (state == GKPlayerStateConnected)
    {
        NSLog(@"Player %@ has connected", playerID);

        if (!matchStarted && !match.expectedPlayerCount)
        {
            // start game here
            [self activateGameGUI];
        }
    }
    else
    {
        NSLog(@"Player state changed to unknown");
    }
}
```

Retrieving Player Names

Matches use player identifiers, not player names. Identifiers are basically numerical indicators of a particular Game Center player in the form of a string. You can retrieve an array of `GKPlayer` objects, with their human-readable `alias` name property, on demand. This operation runs asynchronously, so if you want to display the "name" of your new opponent, you'll have to wait for the completion block to execute. I recommend storing the

From the Library of Eric Berna

opponent (or opponents) locally, as demonstrated here, so you do not have to run this operation more than once:

```
[GKPlayer loadPlayersForIdentifiers:
       [NSArray arrayWithObject:playerID]
   withCompletionHandler:^(NSArray *players, NSError *error)
{
    [self activateGameGUI];
    if (error) return;

    // Store the opponent object (2-player game)
    opponent = [players lastObject];

    // Announce the match
    NSString *matchString = [NSString stringWithFormat:
        @"Commencing Match with %@", opponent.alias];
    UIAlertView *alert = [[UIAlertView alloc]
        initWithTitle:matchString message:nil delegate:nil
        cancelButtonTitle:nil otherButtonTitles:@"Okay", nil];
    [alert show];
}];
```

Game Play

Games primarily involve sending data back and forth through the `GKMatch` object and updating the local GUI to reflect any remote changes. Generally, you'll want to send tiny bits of data very quickly for real time updates. You can bend or break this rule as needed, especially if you don't need your updates to arrive quickly. Matches use two kinds of data transmission: reliable and unreliable.

If your data must arrive coherently and cannot be lost, use reliable transmission. Reliable transmissions are delivered in the order they are sent, are guaranteed to arrive, and will be resent until they are fully transmitted. Think "TCP" transmission when working with reliable connections.

Unreliable data is sent once and may be received out-of-order by recipients. Use this approach for data that arrives in small packets, but that must use near-instantaneous transmission to be useful, such as in a live shooting action game. Think "UDP" transmission for unreliable connections.

Here's an example of a simple texting application that sends string data as it is typed into a sending text view. It uses reliable transmission ensuring that every character is transmitted in order as it is typed. As the application receives data, it converts that information back into a string and shows it on the receiving text view.

```
- (void)textViewDidChange:(UITextView *)textView
{
    NSError *error;
    NSData *dataToSend =
```

From the Library of Eric Berna

```
        [sendingView.text dataUsingEncoding:NSUTF8StringEncoding];
    BOOL success = [match sendDataToAllPlayers:dataToSend
        withDataMode:GKMatchSendDataReliable error:&error];
    if (!success)
        NSLog(@"Error sending match data: %@", error.localizedFailureReason);
}

- (void)match:(GKMatch *) aMatch didReceiveData:(NSData *)data
    fromPlayer:(NSString *)playerID
{
    NSString *received = [[NSString alloc]
        initWithData:data encoding:NSUTF8StringEncoding];
    receivingView.text = received;
}
```

Serializing Data

When sending data for complex gameplay, you may want to send information that's more structured than simple strings. JSON serialization provides an easy-to-use solution for compact structured data objects, both for dictionaries with key/value pairs and arrays with information sent in a key-less pre-ordained order. For example, imagine you have an application that "rolls" a local die and then transmits its value. Rolling a number might be just one action among many a game might use. Creating a key/value dictionary pair allows your game to differentiate between rolls, moving pieces, selecting cards, and other game-specific commands.

```
// Roll the die
unsigned int roll = random() % 6;
localRoll = [NSNumber numberWithUnsignedInt:roll];

// Create an info dictionary for the roll
NSDictionary *dictionary =
    [NSDictionary dictionaryWithObject:localRoll forKey:@"Roll"];

// Transmit the roll as serialized JSON data
NSData *json = [NSJSONSerialization
    dataWithJSONObject:dictionary options:0 error:nil];
[match sendDataToAllPlayers:json
    withDataMode:GKMatchSendDataReliable error:nil];
```

On the receiving side, the other player's app can deserialize the data, recover the roll value and apply it locally. More complex actions (for example, if the player has several pieces that can be moved, not just one), require slightly more complicated structures, but the send and receive components remain the same: Convert a JSON-serializable object to data, send it, and translate it back to object form on the receiving end.

From the Library of Eric Berna

```
- (void)match:(GKMatch *) aMatch
    didReceiveData:(NSData *)data
    fromPlayer:(NSString *)playerID
{
    NSDictionary *dict =
        [NSJSONSerialization JSONObjectWithData:data
            options:0 error:nil];
    NSString *key = [[dict allKeys] lastObject];
    if (!key) return;

    if ([key isEqualToString:@"Roll"])
    {
        // take action here, e.g. move n spaces
        // or extract which piece to move
    }
}
```

Top-level JSON-serializable objects must be arrays or dictionaries. Below that, they can contain further arrays, dictionaries, strings, numbers, and `NSNull` instances. Numbers cannot be NaN or infinity. You can test whether your objects are valid JSON instances by calling `isValidJSONObject:`.

Sending Other Forms of Data

Although JSON serialization is simple to implement, it may not provide sufficient flexibility for your needs. If you need to send `NSData` or `NSDate` objects in addition to your dictionaries, arrays, strings, or numbers, you may want to investigate standard property list serialization instead.

Property lists provide a helpful abstract data type. A property list object can point to data (`NSData`), strings (`NSString`), arrays (`NSArray`), dictionaries (`NSDictionary`), dates (`NSDate`), and numbers (`NSNumber`). When working with collection objects (i.e., arrays and dictionaries), all members and keys must be property list objects as well (that is, data, strings, numbers, and dates, as well as embedded arrays and dictionaries).

While that seems limiting, you can transform most structures and objects to and from strings. For example, you can use the built-in `NSStringFromCGPoint()` or `NSStringFromClass()` functions, or you can create your own. The following pair of methods extends the `UIColor` class, providing functionality needed to send color information across a GameKit connection as strings:

```
@implementation UIColor (utilities)
- (NSString *) stringFromColor
{
    // Recover the color space and store RGB or monochrome color
    const CGFloat *c = CGColorGetComponents(self.CGColor);
    CGColorSpaceModel csm =
        CGColorSpaceGetModel(CGColorGetColorSpace(self.CGColor));
    return (csm == kCGColorSpaceModelRGB) ?
        [NSString stringWithFormat:@"%0.2f %0.2f %0.2f %0.2f",
```

From the Library of Eric Berna

```
            c[0], c[1], c[2], c[3]] :
        [NSString stringWithFormat:@"%0.2f %0.2f %0.2f %0.2f",
            c[0], c[0], c[0], c[1]];
}

+ (UIColor *) colorWithString: (NSString *) colorString
{
    // Read a color back from a string
    const CGFloat c[4];
    sscanf([colorString cStringUsingEncoding:NSUTF8StringEncoding],
        "%f %f %f %f", &c[0], &c[1], &c[2], &c[3]);
    return [UIColor colorWithRed:c[0] green:c[1] blue:c[2] alpha:c[3]];
}
@end
```

Once in property list form, you can serialize your data and send it as a single chunk. On receipt, the deserialized data is ready to use. Listing 24-1 shows example `transmit` and `receivedData:` methods that handle serialization and deserialization. This code comes from a sample that stores various kinds of game state in an `NSDictionary` object.

> **Note**
>
> You can use the `NSKeyedArchiver` and `NSKeyedUnarchiver` classes, as well as the `NSPropertyListSerialization` class and `CFPropertyListCreateFromXMLData()` function demonstrated here.

Listing 24-1 **Serializing and Deserializing Property Lists**

```
- (void) transmit
{
    NSString *errorString;

    // Send a copy of the local points to the peer
    // by serializing the property list into data
    NSData *plistdata = [NSPropertyListSerialization
         dataFromPropertyList:pointData
         format:NSPropertyListXMLFormat_v1_0
         errorDescription:&errorString];

    if (plistdata)
        [match sendDataToAllPlayers: plistdata
            withDataMode:GKMatchSendDataReliable error:nil];
    else
         NSLog(@"Error serializing property list: %@", errorString);
}

- (void)match:(GKMatch *) aMatch
```

From the Library of Eric Berna

```
    didReceiveData:(NSData *)data
    fromPlayer:(NSString *)playerID
{
    // Deserialize the data back into a property list
    CFStringRef errorString;
    CFPropertyListRef plist =
        CFPropertyListCreateFromXMLData(kCFAllocatorDefault,
        (CFDataRef)data, kCFPropertyListMutableContainers,
        &errorString);

    if (!plist)
    {
        NSLog(@"Error deserializating data: %@", errorString);
        return;
    }

    // Do something with the received data
}
```

Synchronizing Data

Although Apple provides a turn-by-turn gaming option via Game Center (see the following section), you may want to develop games that perform similar features in a more immediate atmosphere, where gamers are not free to leave and return to take their turn. You can implement a turn-by-turn style game using direct matches by synchronizing data between each player. For example, one of the most common tasks for turn-based games is to choose who goes first. You can create your own roll-for-first system by generating a random number and comparing it to your opponent's.

Because each application may be slightly out of sync with the other, never assume that your roll will occur before or after the other player's. Instead, store both values locally and wait until both items are available before checking for a winner. That means you'll want to call your startup check both in the received-data method and in the method that performs the roll on your side.

The following sample method demonstrates how to do this. It always checks to see if it's resolved a winner. If so, the method returns as it may be called more than once after a winner has been selected. Next, it checks for the existence of both a local and remote roll. Only when those two are available does it move on to the check.

In the case of a tie, it resets both values, and starts a new roll and check. Otherwise, it selects the winner (higher value wins here) and starts the game play either deferring to the other player (`opponentGoesFirst`) or not.

```
- (void) checkStartupWinner
{
    // Already resolved the startup winner?
    if (startupResolved)
```

From the Library of Eric Berna

```
        return;

    // Need data from both remote and local rolls in order to
    // determine a winner
    if (!remoteRoll || !localRoll)
    {
        [self performSelector:@selector(checkStartupWinner)
            withObject:nil afterDelay:1.0f];
        return;
    }

    // Both rolls are in. Decide a winner
    NSLog(@"Remote roll: %@, local roll: %@", remoteRoll, localRoll);
    unsigned int local = localRoll.unsignedIntValue;
    unsigned int remote = remoteRoll.unsignedIntValue;

    // Handle a tie conflict by resetting and trying again
    if (local == remote)
    {
        NSLog(@"TIE!");
        remoteRoll = nil;
        localRoll = nil;
        [self sendRoll:GKROLLFORFIRST];
        return;
    }

    startupResolved = YES;

    if (remote > local)
    {
        // they go first.
        opponentGoesFirst = YES;
}
    else
    {
        // you go first
        opponentGoesFirst = NO;
    }

    // reset both rolls
    localRoll = nil;
    remoteRoll = nil;
}
```

From the Library of Eric Berna

Turn-by-Turn Matches

Turn-based matches allow your applications to hook into a Game Center feature that moderates games based on one-at-a-time gameplay, such as checkers, gin rummy, scrabble, or bridge. It introduces store-and-forward game state, allowing each player to make a move and then pass that updated state back to the Game Center store.

The turn-based matchmaker allows users to connect both anonymously and by invitation, just as the standard game matchmaker does. What differs is how you establish your game and how game control moves from person to person to person by passing a virtual "baton." The person holding that baton has control of the game until he or she makes a move.

From there, the baton is passed to the next player—it may be the next player in order or someone based on another rule. For example, after each full round it might pass to the person with the highest score or the lowest. Or a player might choose whom to discard an item to, passing game control at the same time.

How you implement that control is up to you and the requirements of your game. How you establish a match and implement turn-by-turn follows a few basic patterns.

This chapter was written during the iOS 5 beta period and reflects the technology during that time. Be aware that some features varied in completeness and reliability as this material was developed; be prepared to adapt your methods according to the final release API.

Starting a Match

The process to initiate a turn-by-turn match mirrors the approach for standard games. You create the same kind of standard match request, specifying the minimum and maximum number of players.

```
- (GKMatchRequest *) matchRequest
{
    GKMatchRequest *request = [[GKMatchRequest alloc] init];
    request.minPlayers = 2; // Between 2 and 4
    request.maxPlayers = 2; // Betseen 2 and 4
    return request;
}
```

Use that request to initialize a `GKTurnBasedMatchmakerViewController` and present it. Your primary class must implement the `GKTurnBasedMatchmakerViewControllerDelegate` protocol. Assign it as the delegate for your view controller and set the `showExistingMatches` property to `YES`. This allows your user to select from a new match as well as presenting existing matches.

```
- (void) startMatch
{
    GKMatchRequest *request = [self matchRequest];
```

From the Library of Eric Berna

```
    GKTurnBasedMatchmakerViewController *viewController =
        [[GKTurnBasedMatchmakerViewController alloc]
            initWithMatchRequest:request];
    viewController.turnBasedMatchmakerDelegate = self;
    viewController.showExistingMatches = YES;

    self.navigationItem.rightBarButtonItem = nil;
    [self presentModalViewController:viewController animated:YES];
}
```

As with many Game Center features, your code needs to handle both in-game match requests and external invitation requests. Your primary class will need to declare and implement the `GKTurnBasedEventHandlerDelegate` as well. One of this protocol's callbacks, for handling invitations, represents the second entry-point for working with the `GKTurnBasedMatchmakerViewController`.

To subscribe to turn-based events, you must set your class as the event handler's delegate.

```
[GKTurnBasedEventHandler sharedTurnBasedEventHandler].delegate
    = self;
```

With the class established as the delegate, you may receive invitation callbacks. Implement the `handleInviteFromGameCenter:` method to catch invites and respond to them. As before, the view controller is initialized with a standard match request, but you set the request's `playersToInvite` property to the array passed to you. Further, you set the view controller's `showExistingMatches` property to `NO`. When handling an invitation, you should only work with the match you've been invited to.

```
- (void)handleInviteFromGameCenter:
    (NSArray *)playersToInvite
{
    GKMatchRequest *request = [self matchRequest];
    request.playersToInvite = playersToInvite;
    GKTurnBasedMatchmakerViewController *viewController =
        [[GKTurnBasedMatchmakerViewController alloc]
            initWithMatchRequest:request];
    viewController.showExistingMatches = NO;
    viewController.turnBasedMatchmakerDelegate = self;
    [self presentModalViewController:viewController animated:YES];
}
```

View Controller Delegate Methods

The turn-by-turn view controller's delegate must implement four key methods. As with standard game center controllers, you should respond to the user tapping the *Cancel* button and to any connection fail situations. The following two methods implement basic responses to these situations. They dismiss the controller and then restore the interface to a state where the user can try again to establish a match.

From the Library of Eric Berna

```
// Game Center Fail
- (void)turnBasedMatchmakerViewController:
    (GKTurnBasedMatchmakerViewController *)viewController
    didFailWithError:(NSError *)error
{
    [self dismissModalViewControllerAnimated:YES];
    showAlert(@"Error creating match: %@", error.localizedFailureReason);
    [self initInterfaceForMatch];
}

// User cancel
- (void)turnBasedMatchmakerViewControllerWasCancelled:
    (GKTurnBasedMatchmakerViewController *)viewController
{
    [self dismissModalViewControllerAnimated:YES];
    [self initInterfaceForMatch];
}
```

Handling Quits

The third method allows your application to respond to quit or forfeit situations. There are two elements in this method that you will have to think about further as you move forward.

```
// Handle Quit/Forfeit
- (void)turnBasedMatchmakerViewController:
    (GKTurnBasedMatchmakerViewController *)viewController
    playerQuitForMatch:(GKTurnBasedMatch *)aMatch
{
    NSLog(@"Handle match quit");

    // Respond locally to quit

    // Pass the quit event on to the next person
    [match participantQuitInTurnWithOutcome:
            GKTurnBasedMatchOutcomeQuit
        nextParticipant: [self nextParticipant]
        matchData: match.matchData
        completionHandler:^(NSError *error)
     {
         if (error)
             NSLog(@"Error while quitting match: %@",
                 error.localizedFailureReason);
     }];
}
```

From the Library of Eric Berna

First is the `nextParticipant:` parameter. Upon receiving a quit event, you need to respond locally to any changes caused by the loss of one of the participants. Then, you pass the quit on to the next person in line. The order of players in the match's participant array is always fixed, even as control can be passed out of order. You can use that information to choose who the next person to deal with is.

The simplest way to find the next person is to retrieve the index of the current participant, add one, and take a modulo. This ensures that the `nextParticipant` method always returns the next in line.

```
- (GKTurnBasedParticipant *) nextParticipant
{
    NSUInteger current = [match.participants
        indexOfObject:match.currentParticipant];
    NSUInteger count = match.participants.count;
    NSUInteger nextIndex = (current + 1) % count;
    GKTurnBasedParticipant *participant =
        [match.participants objectAtIndex:nextIndex];
    return participant;
}
```

The problem with this approach is that the next participant in line may or may not still be around to receive that update. You can tweak the `nextParticipant` code to check whether a person has an active status before selecting them as the target. Here's a rough idea of how that method might work, although you need to tweak this as needed and add a check for nil on this method's consumer:

```
- (GKTurnBasedParticipant *) nextParticipant
{
    NSUInteger current = [match.participants
        indexOfObject:match.currentParticipant];
    NSUInteger count = match.participants.count;
    NSUInteger nextIndex = (current + 1) % count;

    for (int i = 0; i <= count; i++)
    {
        GKTurnBasedParticipant *participant =
            [match.participants objectAtIndex:nextIndex];
        if (participant.status == GKTurnBasedParticipantStatusActive)
            return participant;
        nextIndex = (nextIndex + 1) % count;
    }

    NSLog(@"Unable to find an active next participant");
    return nil;
}
```

From the Library of Eric Berna

The other parameter used in the quit/forfeiture callback is the `matchData:` one. You must retrieve that data from Game Center before you ever update it or pass it on as a parameter. Once you have loaded the data, you can retrieve it via the `matchData` parameter (e.g., `match.matchData`).

```
[match loadMatchDataWithCompletionHandler:
    ^(NSData *matchData, NSError *error)
{
    if (error)
    {
        // handle error here
        return;
    }

    // match.matchData is now valid
}];
```

Match Found

The fourth view controller callback allows your game to respond to successful matchmaking. Here is where you store the local match, dismiss the view controller, and set the event handler delegate (if you have not done so already, which you probably have). From there, you move on to actual game play.

```
// User selected match
- (void)turnBasedMatchmakerViewController:
        (GKTurnBasedMatchmakerViewController *)viewController
    didFindMatch:(GKTurnBasedMatch *)aMatch
{
    NSLog(@"Did find match");
    match = aMatch;

    if (viewController)
        [self dismissModalViewControllerAnimated:YES];

    [GKTurnBasedEventHandler sharedTurnBasedEventHandler].delegate = self;

    NSString *currentPlayerID = match.currentParticipant.playerID;
    if (currentPlayerID && [currentPlayerID isEqualToString:player.playerID])
    {
        [self myTurn];
    }
    else
    {
        self.title = @"Waiting";
    }
```

From the Library of Eric Berna

```
    [self showParticipants];
}
```

When your application hits this method, there are a variety of scenarios that you should anticipate. First, the application might be waiting for other players to join. Second, other players may have joined, but it might not be your turn. Always check for the current player, the match's `currentParticipant`, and see if that matches your user. If so—and only if so—enable the interface to allow the user to take a turn.

If it is not your user's turn, make sure the interface reflects this reality. Do not let your user interact with the game state when he or she does not hold the game play "baton." This method represents one of two places where the game play may pass to your user. The other place, which is called more regularly, is part of the event handler's delegate.

```
- (void)handleTurnEventForMatch:(GKTurnBasedMatch *)aMatch
{
    if ([match.matchID isEqualToString:aMatch.matchID])
    {
        match = aMatch;
        [self myTurn];
    }
}
```

Again, this method establishes the GUI for the turn to take place, but does not actually peform any turn action. The user must set the turn in motion once the GUI has been prepared. In the following example method, the GUI is disabled until the match data has finished loading. Once it arrives, the completion handler retrieves the turn data, updates the presentation to match that data, and then enables the interface.

```
- (void) myTurn
{
    self.navigationItem.leftBarButtonItem = nil;
    self.navigationItem.rightBarButtonItem = nil;
    self.title = @"Loading Match Data";

    [match loadMatchDataWithCompletionHandler:
        ^(NSData *matchData, NSError *error) {
        NSString *theText = [[NSString alloc]
            initWithData:match.matchData
            encoding:NSUTF8StringEncoding];
        textView.text = theText;
        self.navigationItem.rightBarButtonItem =
           BARBUTTON(@"Take my turn", @selector(play));
        self.navigationItem.leftBarButtonItem =
           BARBUTTON(@"Quit", @selector(quitMatch));
        self.title = nil;
    }];
}
```

From the Library of Eric Berna

Another method, in this case play, handles the state update and submits the changes to Game Center. The following sample method ends its turn by selecting the next participant and sending the match data. Once it does so successfully, it again updates the game GUI, this time (via `notMyTurn`) disabling user interaction until it is once again the current user's turn.

```
- (void) play
{
    GKTurnBasedParticipant *participant = [self nextParticipant];
    if ([participant.playerID isEqualToString:player.playerID])
    {
        NSLog(@"No one else...");
        showAlert(@"No one to play with yet");
        return;
    }

    // Retrieve the turn data
    NSData *updated = [self getGameData];

    // Update Game Center
    [match endTurnWithNextParticipant:participant
        matchData:[updated dataUsingEncoding:NSUTF8StringEncoding]
        completionHandler:^(NSError *error)
     {
         if (error)
         {
             NSLog(@"Error completing turn: %@",
                 error.localizedFailureReason);
         }
         else
             [self notMyTurn];
     }];
}
```

Your application is responsible for creating a data object that represents the current game state after the user's turn (`getGameData`). How you design that data will be application-specific. It might be as simple as a dictionary representing pieces on a board or as complex as a collection of points, colors, and geometric transforms for a cooperative drawing application.

Ending the Match

Matches end. Someone might win or lose during their turn. If your application detects that the match has ended, you can terminate the ongoing match. To do so, request that the match end for the current user as shown in the following method:

From the Library of Eric Berna

```
- (void) quitMatch
{
    if (![self isCurrentParticipant])
    {
        NSLog(@"You cannot quit. You are not the current participant.");
        return;
    }

    // Orderly match quit
    [match endMatchInTurnWithMatchData:match.matchData
        completionHandler:^(NSError *error)
     {
         if (error)
         {
             NSLog(@"Error while quitting match: %@",
                 error.localizedFailureReason);
             self.title = @"Could not end match";
             // return;
         }

         [self initInterfaceForMatch];
     }];
}
```

As the match ends, the view controller delegate receives the quit-in-turn callback shown earlier in this section. In addition, the event handler receives a callback once the match is over.

```
- (void)handleMatchEnded:(GKTurnBasedMatch *)aMatch
{
    NSLog(@"Match has ended");
    if ([match.matchID isEqualToString:aMatch.matchID])
    {
        [self initInterfaceForMatch];
        self.title = @"Match ended";
        match = nil;
    }
}
```

The completed match remains stored in Game Center until a player deletes it or you remove it manually. The following method disposes of a given match programmatically:

```
- (void) removeMatch: (GKTurnBasedMatch *) aMatch
{
    [aMatch removeWithCompletionHandler:^(NSError * error)
     {
        if (error)
```

From the Library of Eric Berna

```
            NSLog(@"Error removing match: %@", error.localizedFailureReason);
    }];
}
```

Force Quitting Matches

During debugging, there come times that you simply want to clear out all the ongoing matches for a test user. The following method takes a brute-force approach, removing all current matches. This method is not meant for deployment to App Store end users, but provides a handy tool in your debugging arsenal.

```
- (void) quitAllMatches
{
    [GKTurnBasedMatch loadMatchesWithCompletionHandler:
        ^(NSArray *matches, NSError *error)
     {
         if (error)
         {
             NSLog(@"Error loading matches: %@",
                 error.localizedFailureReason);
         }
         else
         {
             NSLog(@"You were involved in %d matches", matches.count);
             for (GKTurnBasedMatch *aMatch in matches)
             {
                 [aMatch removeWithCompletionHandler:
                     ^(NSError *error)
                  {
                      if (error)
                          NSLog(@"Could not quit from %@: %@",
                              aMatch.matchID,
                              error.localizedFailureReason);
                      else
                          NSLog(@"Killed match %@", aMatch.matchID);
                  }];
             }

             self.title = nil;
             [self initInterfaceForMatch];
             showAlert(@"You quit out of %d matches", matches.count);
         }
     }];
}
```

From the Library of Eric Berna

Game Center Voice

Each game center match can establish voice chats. These work best as push-to-talk walkie-talkies between players. Your application can create a group chat for everyone, limit chat between team members, or create a single one-to-one channel between two players. That's because all chats are named. When players connect into the same-named chat, they can hear and participate in that chat. Your application controls the names, and API-provided hooks allows you to build some high-level GUI controls to manage the chat.

Testing for Chat Availability

Before offering chat to your users, test whether the feature is available. The `GKVoiceChat` class offers a simple check that determines whether Voice over IP is currently permitted on the device. As Apple puts it in its documentation, "Some countries or phone carriers may restrict the availability of voice over IP services."

```
if (![GKVoiceChat isVoIPAllowed])
        return;
```

Establishing a Play and Record Audio Session

To use Game Center Voice, you need to link your project to the `AVFoundation` framework. Then, as the program launches, set the shared `AVFoundation` audio session to use a play-and-record category. The following method updates the shared session to be able to both play and record audio:

```
- (BOOL) establishPlayAndRecordAudioSession
{
    NSLog(@"Establishing Audio Session");
    NSError *error;
    AVAudioSession *audioSession =
        [AVAudioSession sharedInstance];
    BOOL success = [audioSession setCategory:
        AVAudioSessionCategoryPlayAndRecord error:&error];
    if (!success)
    {
        NSLog(@"Error setting session category: %@",
            error.localizedFailureReason);
        return NO;
    }
    else
    {
        success = [audioSession setActive: YES error: &error];
        if (success)
        {
            NSLog(@"Audio session is active");
```

From the Library of Eric Berna

```
            return YES;
        }
        else
        {
            NSLog(@"Error activating audio session: %@",
                error.localizedFailureReason);
            return NO;
        }
    }

    return NO;
}
```

Creating a Voice Chat

Create and hold onto a `GKVoiceChat` instance for each chat you wish to use within your application, typically by assigning it to a `strong` instance variable or an array. Each voice chat channel uses a simple string to identify it. For example, you might create a "GeneralChat" for your match:

```
chat = [match voiceChatWithName:@"GeneralChat"];
```

There are no further security elements in place. All chats are limited to the current match and are controlled by your application. You programmatically provide access to the chats as required.

Starting and Stopping a Chat

To start your chat, create it, set its `active` property to `NO`, optionally initialize its `volume` property, and send it the `start` message. The chat is immediately ready to use, but will not start recording or sending data yet.

```
chat = [match voiceChatWithName:@"GeneralChat"];
chat.active = NO;
chat.volume = 1.0f;
[chat start];
```

To finish a chat, send it `stop`. You can start it again later or, if you reassign the variable that is holding onto it, allow it to be deallocated.

```
[chat stop];
chat = nil;
```

Chat State Monitoring

When you call `start` or `stop` on a chat, Game Center sends out notifications that the local player has joined the chat. There are four possible voice chat states: connected, disconnected, speaking, and silent. Notifications go out for each of these. So, if you have an

From the Library of Eric Berna

active chat and your user pauses to take a breath, `GKVoiceChat` propagates a `GKVoiceChatPlayerSilent` event, followed by `GKVoiceChatPlayerSpeaking`, when he or she starts talking again.

You can use these events to update local icons, highlighting them when a user is speaking, dimming them when they are not, or removing them from view when the user is no longer part of the chat. Add a state update handler to each chat to control the GUI presentation when a player's chat state changes.

```
chat.playerStateUpdateHandler =
    ^(NSString *playerID, GKVoiceChatPlayerState state)
{
    switch (state)
    {
        case GKVoiceChatPlayerSpeaking:
            // Highlight player's picture
            break;
        case GKVoiceChatPlayerSilent:
            // Dim player's picture
            break;
        case GKVoiceChatPlayerConnected:
            // Show player name/picture
            break;
        case GKVoiceChatPlayerDisconnected:
            // Hide player name/picture
           break;
    }
};
```

Implementing a Chat Button

It helps to treat your chats as push-to-talk items, activating them only when a user opts in to speak. For example, you might create a talk button that activates the local microphone as it is pressed, and deactivates it when released. Here's how you could define that button:

```
talkButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
[talkButton setTitle:@"Speak" forState:UIControlStateNormal];
talkButton.frame = CGRectMake(0.0f, 0.0f, 200.0f, 40.0f);
[self.view addSubview:talkButton];

[talkButton addTarget:self action:@selector(startSpeaking)
    forControlEvents:UIControlEventTouchDown];
[talkButton addTarget:self action:@selector(stopSpeaking)
    forControlEvents:UIControlEventTouchUpInside |
    UIControlEventTouchUpOutside];
```

From the Library of Eric Berna

The two callbacks involve nothing more than setting the chat's active property to YES (microphone enabled and live) or NO (microphone disabled). As you've already seen, always initially set this property to NO when you create the chat.

```
- (void) stopSpeaking
{
    chat.active = NO;
}

- (void) startSpeaking
{
    chat.active = YES;
}
```

Controlling Volume

The chat's volume property ranges between 0.0 (muted) and 1.0 (full volume). This controls how loudly the chat plays back on the local device's speakers. In addition, you can control whether you're listening to other players on a player-by-player basis. Mute individual players by calling setMute:forPlayer:. Like the player state update handler, this feature is meant to be integrated into a player-by-player icon GUI, allowing your users to mute others individually.

```
[chat setMute:YES forPlayer: playerID];
```

You can mute all players by iterating through the chat's list of player identifiers. This affects audio for only the selected chat. If the player is participating in more than one chat, you must mute each one or set the volume property for each chat to 0.0.

```
- (void) muteChat: (GKVoiceChat *) aChat
{
    NSLog(@"Muting chat %@", aChat.name);
    for (NSString *playerID in aChat.playerIDs)
        [aChat setMute:YES forPlayer:playerID];
}

- (void) unmuteChat: (GKVoiceChat *) aChat
{
    NSLog(@"Unmuting chat %@", aChat.name);
    for (NSString *playerID in aChat.playerIDs)
        [aChat setMute:NO forPlayer:playerID];
}
```

From the Library of Eric Berna

GameKit Peer Services

GameKit provides peer-to-peer connectivity between iOS devices, outside of the Game Center milieu. The framework helps you create interconnected applications that exchange live data in real time. In its default implementation, GameKit works by creating and managing an ad-hoc Bluetooth network that lets devices find each other, establish a connection, and transmit data through that connection. GameKit also allows you to find, connect, and transmit to devices on the same Wi-Fi network.

Using Bluetooth and Wi-Fi provides a fast and reliable approach to interdevice communications. Unfortunately, peer-to-peer communication using GameKit is neither fast nor reliable. Trying to use GameKit in this fashion can prove an extremely frustrating experience for end users. You should expect your support commitments to be tested accordingly if you decide to include this GameKit feature in your applications.

After having deployed a number of apps to App Store that leverage ad-hoc peering, I emphatically recommend that you consider implementing standard Game Center matches instead.

Peering is slow to establish, frustrating to use, and can become more and more unreliable over time should users keep trying to connect after failed attempts. (Users will not know to reset the stack by rebooting their devices.)

What's more, the standard use scenario requires that users own more than one device and use them in close quarters. I have personally found that no matter how nifty the peer-to-peer idea is, business-wise, it's a poor investment of time. Using Game Center is far simpler and more rugged and provides a greater potential user-base for those features.

With that said, here is a quick introduction to the peer-to-peer aspects of GameKit. This section is abbreviated from this book's last edition. If you truly want to explore peer-to-peer development, you may want to go back to the original discussion there. It was written when the technology was just first being introduced, before years of being worn down by end-user feedback.

> **Note**
>
> You can increase GK connection reliability by using server/client (`GKSessionModeServer/GKSessionModeClient`) connections instead of peering (`GKSessionModePeer`). The downside is that your connection metaphor won't be as simple. You'll have to update your GUI to reflect each task, such as "Start a session" and "Connect to a session started on another device."

GameKit Bluetooth Limitations

For Bluetooth, you need proximity. Connections are limited to about 10 meters (30 feet). So, think of your audience including people riding together on a train or in a car, in a

From the Library of Eric Berna

convention-hall's meeting room, or working in the same office. Within that range, your application can establish a peer-to-peer connection.

GameKit offers excellent performance for short, tight blips of information. Apple recommends that GameKit transmissions be limited to 1,000 bytes and under. Although GameKit can handle larger blobs, up to 95 Kilobytes at a time, it's not meant for use as general device-to-device data transfer. Try to send too much data at once, and you will receive transmission errors.

If you must transfer large files, you need to break those files into manageable chunks. Make sure that you use standard handshaking and packet checksumming techniques to ensure the reliability of your data.

Device Limitations

GameKit's Bluetooth networking is not for every iOS device. It works with the 3G iPhone and later models, with the second generation iPod touch and later models, and all iPads. Plus, GameKit peering is poorly supported on the Simulator. The `peer-peer` key in the Info.plist `UIRequiredDeviceCapabilities` entry lets you require support for peer-to-peer connectivity over Bluetooth.

Bonjour Sessions

GameKit's peer-to-peer connections are built using Bonjour networking. Bonjour, which is Apple's trade name for zero-configuration networking, allows devices to advertise and discover network services. Built into Mac OS X since version 10.2, Bonjour offers these features without calling attention to itself. For example, Bonjour powers the features that let users find shared music for iTunes or connect to wireless printers without requiring custom configuration. These services automatically appear when they become available and disappear when they're not. It's a powerful OS feature.

GameKit provides that same Bonjour power without having to build the often complicated Bonjour callbacks for registering and detecting services. With GameKit, you request a connection using a "peer picker" controller and then manage a "session" once the connection has been established.

GameKit's session objects provide a single focus point for data transfer management. Each session uses a unique name, which you choose, to advertise itself. When an application looks for another device to connect to, it uses this name to identify compatible services.

If you use a Bonjour browsing service to look for that name, you'll fail. Apple encodes the service name using a SHA-1 hash. For example, a service called "MacBTClient Sample" becomes the "_11d7n7p5tob54j._udp." Bonjour service. GameKit automatically transforms the name you supply so it knows how to find matching services.

Unfortunately, there's no Mac OS X or Windows API available to let you build services from a desktop system that would let you hook into GameKit. Apple's name encryption and unpublished internal protocol more or less guarantees that standard Bonjour communications will not work with iOS-based GameKit applications. You can bypass these limitations by working directly with Bonjour.

From the Library of Eric Berna

Servers, Clients, Peers

GameKit offers three session modes: Applications can act as servers, clients, and peers. Servers advertise a service and initialize a session, allowing clients to search for and connect to them. This is the kind of behavior that a smart printer uses, letting clients find and use its capabilities. It's handy for devices that provide a fixed functionality, but it's not the best choice for most iOS applications, especially games.

Peers work as both servers and clients. They advertise and search simultaneously. Once a peer selects a service, its client/server role is hidden both from the user and the developer. This makes the peer approach easy to develop for iOS. You don't have to build separate client and server applications. One peer-based application does all the work.

The Peer Connection Process

The peer picking process is handled by a class called `GKPeerPickerController`. It provides a built-in series of interactive alert dialogs that automate the task of advertising device availability and selecting a peer. Using this class is not mandatory. You can bypass it and create a custom class to search for and connect to peers.

For simple connections, however, the `GKPeerPickerController` class offers a ready-to-use interface that sidesteps the need for detecting and negotiating with peers. To use the peer picker, you allocate it, set a delegate (which must implement the `GKPeerPickerControllerDelegate` protocol), and show it.

Displaying the Peer Picker

The following code allocates and shows a new peer picker controller, setting its connection style to nearby. This skips an optional interaction step where a user selects between Online and Nearby modes. When presented, it shows the interface in Figure 24-8. You do not have to use a peer picker to establish GameKit sessions. The iOS SDK lets you create your own custom interfaces to work with the underlying GameKit connections.

```
// Create and present a new peer picker
GKPeerPickerController *picker = [[GKPeerPickerController alloc]
    init];
picker.delegate = self;
picker.connectionTypesMask = GKPeerPickerConnectionTypeNearby;
[picker show];
```

From the Library of Eric Berna



Figure 24-8 This is the first screen presented to the user for peer-to-peer Bluetooth/Wi-Fi connections.

When your mask also includes the online type (`GKPeerPickerConnectionTypeOnline`), the picker first asks the user which kind of connection to use before moving on to either the nearby connection interface or to a custom online interface that you build yourself.

Pressing Cancel

Users may cancel out of the peer picker alert. When they do so, the delegate receives a `peerPickerControllerDidCancel:` callback. If you display a Connect button in your application, make sure to restore it at this point so the user can try again.

```
- (void) peerPickerControllerDidCancel: (GKPeerPickerController *)picker
{
    [self setupConnectButton];
}
```

Creating the Session Object

As the picker delegate, you must supply a session object on request. Sessions, which provide an abstract class that creates and manages a data socket between devices, belong to the `GKSession` class and must be initialized with a session identifier. This is the unique string used to create the Bonjour service and link together two iOS devices (peers) both advertising the same service. By setting the display name to `nil`, the session uses the built-in device name.

From the Library of Eric Berna

```
- (GKSession *)peerPickerController:(GKPeerPickerController *)picker
      sessionForConnectionType:(GKPeerPickerConnectionType)type
{
    // Create a new session if one does not already exist
    if (!self.session) {
        self.session = [[GKSession alloc] initWithSessionID:
            (self.sessionID ? self.sessionID : @"Sample Session")
            displayName:nil sessionMode:GKSessionModePeer];

        self.session.delegate = self;
    }
    return self.session;
}
```

Although this is an optional method, you'll usually want to implement it so you can set your session ID and mode. Upon detecting another iOS device with the same advertised service ID, the peer picker displays the peer as a compatible match, as shown in Figure 24-9.



Figure 24-9 The peer picker lists all devices that can act as peers.

Waiting for the peer picker list can take a few seconds or up to a few minutes. During development, you usually need to allow your Bonjour network stack to clear out any previous sessions when you iterate on the code. That's what typically causes some longer delays. Apple recommends always debugging from a clean restart. If debugging delays get frustrating enough, make sure to reboot.

From the Library of Eric Berna

In normal use, connection delays usually hover around 45 seconds at a maximum—if it connects at all. Warn your users to be patient. In Figure 24-9, Binky is the device name for a second iOS device running the same application. When the user taps the name Binky, this device automatically goes into client mode, and Binky goes into server mode.

Client and Server Modes

When a device changes into client mode, it stops advertising its service. The device-choice dialog shown previously in Figure 24-9 changes on the server unit. The client's peer name dims to dark gray and the words "is not available" appear underneath. A few seconds later (and this can actually run up to a minute, so again warn your users about delays), both units update their peer picker display.

Figure 24-10 shows the server and client peer pickers during this process. The client waits as the server receives the connection request (left). On the server, the host user must accept or decline the connection (middle). Should they decline, an updated peer picker notifies the client (right). If they accept, both delegates receive a new callback.

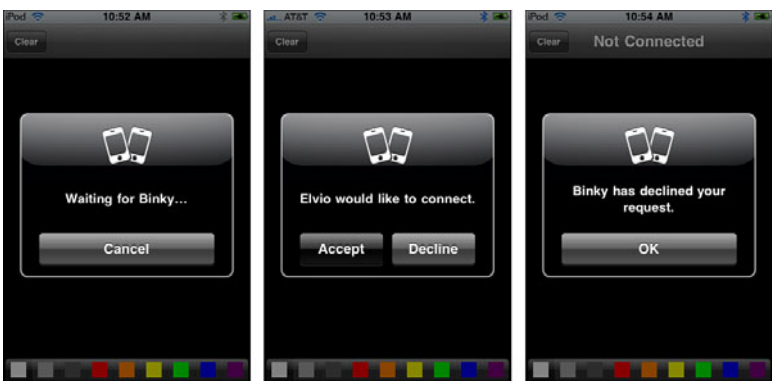


Figure 24-10 Upon choosing a partner, the client goes into wait mode (left) as the server decides whether to accept or decline the connection (middle). Should the server decline, the client receives a notice to that effect (right).

The delegate callback lets the new peers dismiss the peer picker and to set their data received handler.

```
- (void)peerPickerController:(GKPeerPickerController *)picker
    didConnectPeer:(NSString *)peerID
    toSession: (GKSession *) session
{
    // Dismiss the picker, then set the data handler
    [picker dismiss];
    [self.session setDataReceiveHandler:self withContext:nil];
}
```

From the Library of Eric Berna

Sending and Receiving Data

The data handler (in this case, `self`) must implement the `receiveData:fromPeer:inSession:context:` method. The data sent to this method uses an `NSData` object; there are no hooks or handles for partial data receipt and processing. As the data arrives as a single chunk, keep your data bursts short (under 1,000 bytes) and to the point for highly interactive applications.

```
- (void) receiveData:(NSData *)data fromPeer:(NSString *)peer
    inSession: (GKSession *)session context:(void *)context
{
    // handle data here
}
```

Send data via the session object. You can send in reliable mode or unreliable mode. Reliable mode uses error checking and retrying until the data is properly sent. All items are guaranteed to arrive in the order they are sent, using TCP transmission. With unreliable mode, data is sent once using UDP transmission, with no retry, Data may arrive out of order. Use reliable mode (`GKSendDataReliable`) when you must guarantee correct delivery and unreliable mode for short bursts of data that must arrive nearly instantaneously.

```
- (void) sendDataToPeers: (NSData *) data
{
    // Send the data, checking for success or failure
    NSError *error;
    BOOL didSend = [self.session sendDataToAllPeers:data
        withDataMode:GKSendDataReliable error:&error];
    if (!didSend)
        NSLog(@"Error sending data to peers: %@",
            Error.localizedFailureReason);
}
```

One error you'll encounter here results from queuing too much data in reliable mode. This produces a "buffer full" error.

State Changes

The following session delegate callback lets you know when a peer's state has changed. The two states you want to look for are connected, that is, when the connection finally happens after the peer picker has been dismissed, and disconnected, when the other user quits the application, manually disconnects, or moves out of range.

```
- (void)session:(GKSession *)session peer:(NSString *)peerID
    didChangeState:(GKPeerConnectionState)state
{
    /* STATES: GKPeerStateAvailable, GKPeerStateUnavailable,
      GKPeerStateConnected, GKPeerStateDisconnected,
      GKPeerStateConnecting */
```

From the Library of Eric Berna

```
    if (state == GKPeerStateConnected)
    {
        // handle connected state
    }

    if (state == GKPeerStateDisconnected)
    {
        // handle disconnection
    }
}
```

To force a session to disconnect, use the `disconnectFromAllPeers` method.

```
- (void) disconnect
{
    // Disconnect and then reset the session property
    [session disconnectFromAllPeers];
    session = nil;
}
```

Creating a GameKit Helper

You will find a simplified helper class in the sample code that accompanies this chapter. This class hides most of the GameKit details connection and data transfer details, while providing a demonstration of how to use these features. More importantly, it breaks down how you might look at the GameKit process, with its two key details: connection and data.

Peer-to-Peer Voice Chat

GameKit's In-Game Voice service lets applications create a walkie-talkie–style voice channel connecting two devices together. The voice additions, provided by the `GKVoiceChatService` class, sit outside normal GameKit. Chat services connect into the iPhone's audio playback and recording system, so voice chat can listen to and play back audio. Voice Chat then sends its data through GameKit and plays back the data it receives from GameKit.

Unfortunately, you cannot use the peer-to-peer Voice Chat service over a connection other than Bluetooth. `GKVoice` expects a `GKSession` with `GKPeers` in order to transmit its data. If you need to use voice transmission for another connection style, you'll have to write that layer yourself.

Implementing Voice Chat

When working with voice, there's no difference in the way you get started. You display a peer picker and negotiate the connection, as you would normally do with GameKit. The difference arrives once the peer connects. You need to establish the voice chat and redirect the data to and from that service.

From the Library of Eric Berna

Upon connecting to the new peer, set up the voice chat basics. Activate a play-and-record audio session (as shown earlier in this chapter), set the default chat service client, and start a new voice chat with that peer. By setting the `client` property, you ensure that your class receives the voice chat callbacks needed for negotiating data.

```
// Set the voice chat client and start voice chat
[GKVoiceChatService defaultVoiceChatService].client = self;
if (![[GKVoiceChatService defaultVoiceChatService]
    startVoiceChatWithParticipantID: peerID error: &error])
{
    NSLog(@"Error starting voice chat");
    return;
}
```

Your primary class must declare the `GKVoiceChatClient` protocol to do this. When the chat service gathers data through the microphone, it triggers the `voiceChatService:sendData:toParticipantID:` callback. Here, you can redirect voice data to your normal GameKit session. For a voice-only connection, just send along the data. When your application handles both voice and other data, build a dictionary and tag the data with a key, such as `@"voice"`, or when your class receives data through the normal `receiveData:fromPeer:inSession:context:` callback, the same approaches apply. For voice only, use `receivedData:fromParticipantID:` to send the data off to the chat service. Voice Chat allows you to mix game audio with in-game voice. For voice-data hybrid applications, deserialize the data, determine whether the packet included voice or regular data, and redirect that data to the appropriate recipient.

```
- (void)voiceChatService:(GKVoiceChatService *)voiceChatService
    sendData:(NSData *)data
    toParticipantID:(NSString *)participantID
{
    // Send the next burst of data to peers
    [self.session sendData: data toPeers:[NSArray arrayWithObject:
        participantID] withDataMode: GKSendDataReliable error: nil];
}

- (void) receiveData:(NSData *)data
    fromPeer:(NSString *)peer
    inSession: (GKSession *)session context:(void *)context
{
    // Redirect any voice data to the voice chat service
    [[GKVoiceChatService defaultVoiceChatService]
        receivedData:data fromParticipantID:peer];
}
```

From the Library of Eric Berna

Creating an "Online" GameKit Connection

In the GameKit world, "online" currently means any valid connection style other than Game Center or Bluetooth. You might use a local WLAN network to connect to another device on the same network or connect through WWAN (i.e., the cellular service) or Wi-Fi to a remote Internet-based host. GameKit takes you only so far, as shown in Figure 24-11. By selecting Online, your user depends on you to create a custom connection to another device or service.

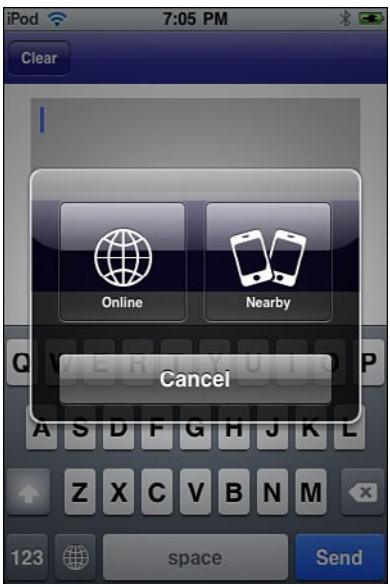


Figure 24-11 The Online GameKit connection means "bring your own networking."

You create this two-item dialog by supplying the online option to the peer picker mask. In all other ways, there's no change in how you create and present a standard GameKit peer picker controller.

```
- (void) startConnection
{
    if (!self.isConnected)
    {
        GKPeerPickerController *picker = [[GKPeerPickerController
            alloc] init];
        picker.delegate = self;
        picker.connectionTypesMask =
            GKPeerPickerConnectionTypeNearby |
            GKPeerPickerConnectionTypeOnline;
```

From the Library of Eric Berna

```
        [picker show];
        if (self.viewController)
            self.viewController.navigationItem.rightBarButtonItem =
                nil;
    }
}
```

Catch the user selection in the `peerPickerController:didSelectConnectionType:` callback. You can assume that if the user selected *Nearby* that all the handshaking dialogs are taken care of for you. Should the user select *Online*, however, it's up to you to move things to the next step. You need to dismiss the picker and display the next stage of the connection task. Here, control passes away from the peer picker to a custom class. It's up to you to produce that class. The following example method uses `BonjourHelper`, which was introduced in the previous edition of this book. Whatever class you use, its role is to begin an online connection, choose a peer to share data with, and serve that data directly to that peer.

```
- (void)peerPickerController:(GKPeerPickerController *)picker
    didSelectConnectionType:(GKPeerPickerConnectionType)type
{
    if(type == GKPeerPickerConnectionTypeOnline)
    {
        [picker dismiss];

        // Establish your own custom connection class here
        [BonjourHelper sharedInstance].sessionID = self.sessionID;
        [BonjourHelper sharedInstance].viewController =
            self.viewController;
        [BonjourHelper sharedInstance].dataDelegate =
            self.dataDelegate;
        [BonjourHelper connect];
    }
}
```

Summary

GameKit offers an exciting new player in the iPhone development arena. Its easy-to-use Game Center connections make it simple for you to deliver applications that communicate between remote iOS devices for satisfying information transfer and game play. In this chapter, you saw how to build those connections and produce real-time data transfers that allow games and other applications to coordinate information between separate devices. Here are a few last-minute thoughts on these technologies:

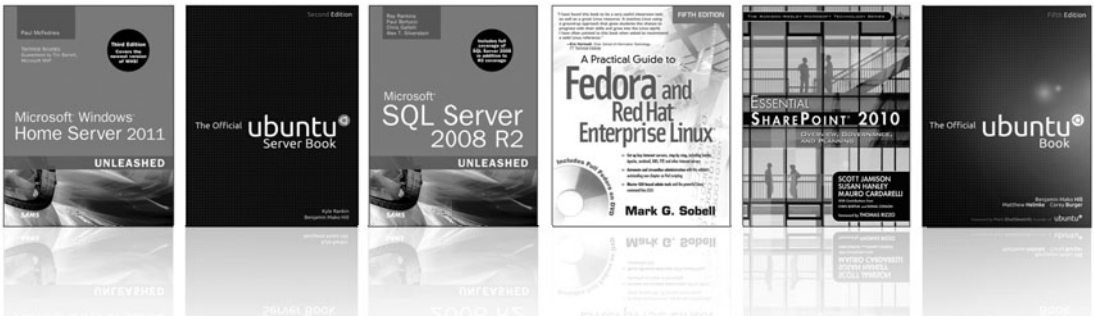
From the Library of Eric Berna

- Apple has not delivered GameKit for Macintosh. Although I keep hoping that it remains on its to-do list, you may want to explore alternative networking technologies to move data from iOS devices to desktop systems. Chapter 15, "Networking," introduced some of those approaches.
- Just because the name has the word "Game" doesn't mean that you cannot use GameKit and Game Center to transfer other kinds of information and data between devices. Don't limit your applications because of Apple's game branding of the service. GameKit provides a wonderful data-transfer infrastructure. The game-specific features lie on top of those.
- When it comes to Bluetooth/Wi-Fi peering, GameKit promises more than it really ever delivered. Consider skipping these technologies and using either straight Bonjour development or Game Center instead. Unfortunately time and page limitations prevented me from exploring Bonjour in this chapter. Many online tutorials can help you get started. Plus, Bonjour runs natively on Windows. Do a Google search for `mDNSResponder` for details.
- When working with Voice Chat locally, especially during testing, remember that nearby users may produce sound loops creating feedback distortion unless they use headsets. For peer-to-peer use, consider that people sitting 10 feet apart from each other can easily talk without the use of technology. For remote Game Center Voice Chat, be aware of how easily the technology can be abused and provide local muting options for your users.
- Don't forget to incentivize your users. Leaderboards and achievements provide wonderful ways to motivate and engage game play beyond a single device or a single session of use.
- Turn-by-turn gaming is still a little creaky around the edges, as I am writing this chapter, but it's going to develop into a terrific system once it settles down and gets going properly.

From the Library of Eric Berna

Try Safari Books Online FREE for 15 days

Get online access to Thousands of Books and Videos



Safari Books Online

FREE 15-DAY TRIAL + 15% OFF*
informit.com/safaritrial

Feed your brain

Gain unlimited access to thousands of books and videos about technology, digital media and professional development from O'Reilly Media, Addison-Wesley, Microsoft Press, Cisco Press, McGraw Hill, Wiley, WROX, Prentice Hall, Que, Sams, Apress, Adobe Press and other top publishers.

See it, believe it

Watch hundreds of expert-led instructional videos on today's hottest topics.

WAIT, THERE'S MORE!

Gain a competitive edge

Be first to learn about the newest technologies and subjects with Rough Cuts pre-published manuscripts and new technology overviews in Short Cuts.

Accelerate your project

Copy and paste code, create smart searches that let you know when new books about your favorite topics are available, and customize your library with favorites, highlights, tags, notes, mash-ups and more.

* Available to new subscribers only. Discount applies to the Safari Library and is valid for first 12 consecutive monthly billing cycles. Safari Library is not available in all countries.

Addison Wesley · Adobe Press · ALPHA · Cisco Press · FT Press · IBM Press · Microsoft Press · New Riders · O'REILLY · Peachpit Press · PRENTICE HALL · que · Redbooks · SAMS · SAS Publishing · Wharton School Publishing · WILEY · wrox

From the Library of Eric Berna



Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to **informit.com/register** to sign in or create an account. You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.

About InformIT — THE TRUSTED TECHNOLOGY LEARNING SOURCE

INFORMIT IS HOME TO THE LEADING TECHNOLOGY PUBLISHING IMPRINTS Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que, and Sams. Here you will gain access to quality and trusted content and resources from the authors, creators, innovators, and leaders of technology. Whether you're looking for a book on a new technology, a helpful article, timely newsletters, or access to the Safari Books Online digital library, InformIT has a solution for you.

informIT.com
THE TRUSTED TECHNOLOGY LEARNING SOURCE

Addison-Wesley | Cisco Press | Exam Cram
IBM Press | Que | Prentice Hall | Sams

SAFARI BOOKS ONLINE

From the Library of Eric Berna

informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE

PEARSON

InformIT is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the top brands, authors, and contributors from the tech community.

Addison-Wesley Cisco Press EXAM/CRAM IBM Press. QUE PRENTICE HALL SAMS | Safari

LearnIT at InformIT

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials? Looking for expert opinions, advice, and tips? **InformIT has the solution.**

- Learn about new releases and special promotions by subscribing to a wide variety of newsletters.
  Visit **informit.com/newsletters**.
- Access FREE podcasts from experts at **informit.com/podcasts**.
- Read the latest author articles and sample chapters at **informit.com/articles**.
- Access thousands of books and videos in the Safari Books Online digital library at **safari.informit.com**.
- Get tips from expert blogs at **informit.com/blogs**.

Visit **informit.com/learn** to discover all the ways you can access the hottest technology content.

Are You Part of the IT Crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube, and more! Visit **informit.com/socialconnect**.



informIT.com THE TRUSTED TECHNOLOGY LEARNING SOURCE

PEARSON

Addison-Wesley Cisco Press EXAM/CRAM IBM Press. QUE PRENTICE HALL SAMS | Safari

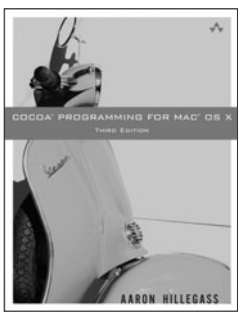
From the Library of Eric Berna

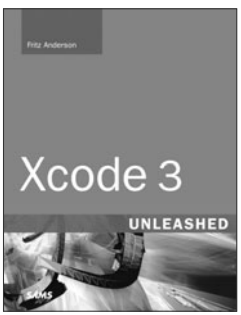
The Titles You Need to Build Killer Apps

informit.com/learnmac

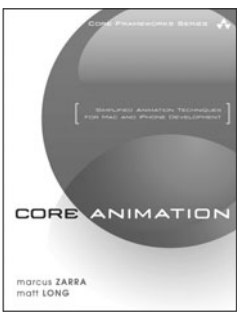
iOS Development is evolving rapidly. So are the resources from Addison-Wesley Professional, Prentice Hall, and Sams.

From getting started with iPhone and iPad application development, to improving the user experience of your applications, to mastering Cocoa, we are updating and publishing the titles you need, when you need them, and in the formats you want!

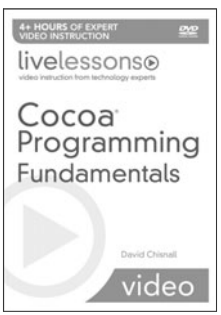


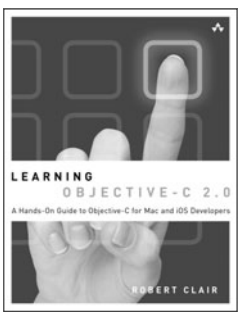


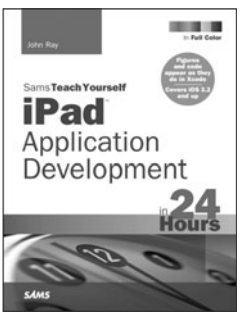




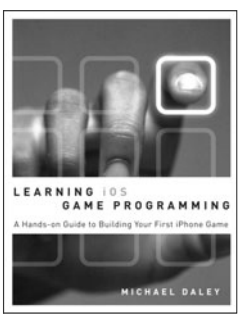


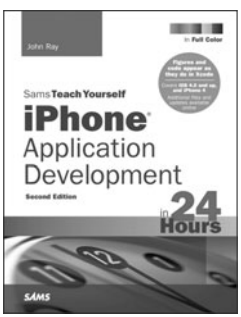


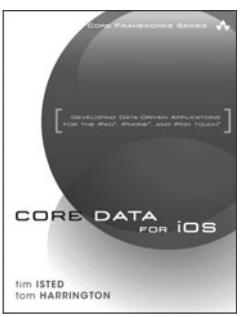












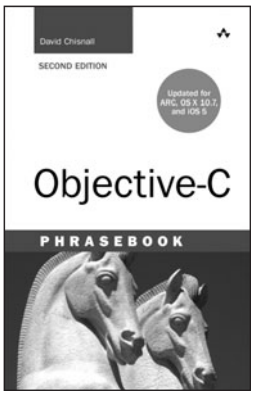
Addison Wesley | PRENTICE HALL | SAMS | informIT | Safari Books Online

Check informit.com/learnmac regularly to see new titles and new editions on Mac and iOS programming.

From the Library of Eric Berna

Developer's Library

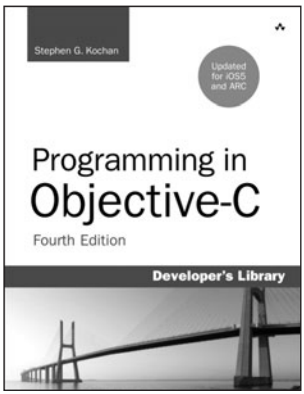
ESSENTIAL REFERENCES FOR PROGRAMMING PROFESSIONALS



Objective-C Phrasebook, Second Edition

David Chisnall

ISBN-13: 978-0-321-81375-6



Programming in Objective-C, Fourth Edition

Stephen G. Kochan

ISBN-13: 978-0-321-81190-5



Cocoa Design Patterns

Erik M. Buck
Donald A. Yacktman

ISBN-13: 978-0-321-53502-3

Other Developer's Library Titles

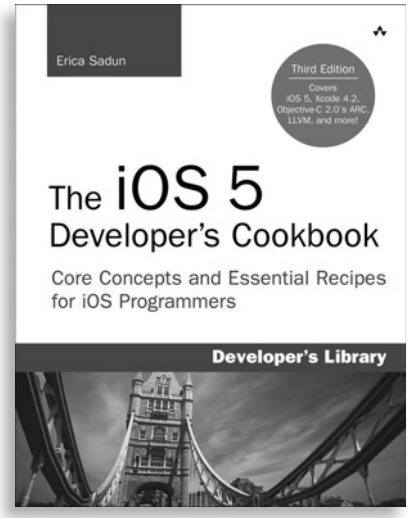
TITLE	AUTHOR	ISBN-13
HTML5 Developer's Cookbook	Chuck Hudson / Tom Leadbetter	978-0-321-76938-1
The Android Developer's Cookbook	James Steele / Nelson To	978-0-321-74123-3
Cocoa Programming Developer's Handbook	David Chisnall	978-0-321-63963-9
Android Wireless Application Development, Second Edition	Shane Conder / Lauren Darcey	978-0-321-74301-5

Developer's Library books are available at most retail and online bookstores. For more information or to order direct visit our online bookstore at **informit.com/store**

Online editions of all Developer's Library titles are available by subscription from Safari Books Online at **safari.informit.com**

Developer's Library
informit.com/devlibrary

From the Library of Eric Berna



FREE Online Edition

Your purchase of ***The iOS 5 Developer's Cookbook, Third Edition*** includes access to a free online edition for 45 days through the Safari Books Online subscription service. Nearly every Addison-Wesley Professional book is available online through Safari Books Online, along with more than 5,000 other technical books and videos from publishers such as Cisco Press, Exam Cram, IBM Press, O'Reilly, Prentice Hall, Que, and Sams.

SAFARI BOOKS ONLINE allows you to search for a specific answer, cut and paste code, download chapters, and stay current with emerging technologies.

Activate your FREE Online Edition at www.informit.com/safarifree

- **STEP 1:** Enter the coupon code: HXKGHFH.
- **STEP 2:** New Safari users, complete the brief registration form. Safari subscribers, just log in.

If you have difficulty registering on Safari or accessing the online edition, please e-mail customer-service@safaribooksonline.com

Safari Books Online

Addison Wesley · Adobe Press · Alpha · Cisco Press · FT Press · IBM Press · lynda.com · Microsoft Press · New Riders · O'Reilly · Peachpit Press · Prentice · Que · Redbooks · Sams · SAS Publishing · Sun microsystems · Wiley

From the Library of Eric Berna



JOIN THE
INFORMIT
AFFILIATE TEAM!

You love our titles and you love to share them with your colleagues and friends...why not earn some $$ doing it!

If you have a website, blog, or even a Facebook page, you can start earning money by putting InformIT links on your page.

Whenever a visitor clicks on these links and makes a purchase on informit.com, you earn commissions* on all sales!

Every sale you bring to our site will earn you a commission. All you have to do is post the links to the titles you want, as many as you want, and we'll take care of the rest.

APPLY AND GET STARTED!

It's quick and easy to apply.
To learn more go to:
http://www.informit.com/affiliates/

*Valid for all books, eBooks and video sales at www.informit.com

Addison Wesley
PRENTICE HALL
SAMS

informIT

From the Library of Eric Berna