Erica Sadun

Third Edition

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

# The iOS 5 Developer's Cookbook

## Core Concepts and Essential Recipes for iOS Programmers

Developer's Library

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

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

—Tim Burks, iPhone Software Developer, TootSweet Software

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

—Jacqui Cheng, Associate Editor, *Ars Technica*

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

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

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

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

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

—John Zorko, Sr. Software Engineer, Mobile Devices

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

—John Muchow, 3 Sixty Software, LLC; founder, iPhoneDeveloperTips.com

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

—Roberto Gamboni, Software Engineer, AT&T Interactive

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

—Jeremy McNally, Developer, entp

# The iOS 5 Developer's Cookbook:

## Core Concepts and Essential Recipes for iOS Programmers

---

Third Edition

Erica Sadun

Addison-Wesley

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

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Sadun, Erica.
The iOS 5 developer's cookbook : core concepts and essential recipes for iOS programmers / Erica Sadun. — 3rd ed.
p. cm.
Rev. ed. of: iPhone developer's cookbook. 2009.
ISBN 978-0-321-75426-4 (pbk. : alk. paper)
1. iPhone (Smartphone)—Programming. 2. Computer software—Development. 3. Mobile computing. I. Sadun, Erica. iPhone developer's cookbook. II. Title.
QA76.8.I64S33 2011
004.16'7—dc23

2011036427



ISBN-13: 978-0-321-75426-4
ISBN-10: 0-321-75426-3

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

First printing November 2011

**Editor-in-Chief**
Mark Taub

**Senior Acquisitions Editor**
Chuck Toporek

**Senior Development Editor**
Chris Zahn

**Managing Editor**
Kristy Hart

**Project Editor**
Anne Goebel

**Copy Editor**
Bart Reed

**Indexer**
Erika Millen

**Proofreader**
Linda Seifert

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

**Editorial Assistant**
Olivia Basegio

**Cover Designer**
Gary Adair

**Composition**
Nonie Ratcliff

❖

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

❖

# Contents at a Glance

# Contents

# Acknowledgments

This book would not exist without the efforts of Chuck Toporek (my editor and whip-cracker), Chris Zahn (the awesomely talented development editor), and Olivia Basegio (the faithful and rocking editorial assistant who kept things rolling behind the scenes). Also, a big thank you to the entire Addison-Wesley/Pearson production team, specifically Kristy Hart, Anne Goebel, Bart Reed, Linda Seifert, Erika Millen, Nonie Ratcliff, and Gary Adair. Thanks also to the crew at Safari for getting my book up in Rough Cuts and for quickly fixing things when technical glitches occurred.

Thanks go as well to Neil Salkind, my agent of many years, to the tech reviewers (Jon Bauer, Joachim Bean, Tim Burks, and Matt Martel) who helped keep this book in the realm of sanity rather than wishful thinking, and to all my colleagues, both present and former, at TUAW, Ars Technica, and the Digital Media/Inside iPhone blog.

I am deeply indebted to the wide community of iOS developers, including Tim Isted, Joachim Bean, Aaron Basil, Roberto Gamboni, John Muchow, Scott Mikolaitis, Alex Schaefer, Nick Penree, James Cuff, Jay Freeman, Mark Montecalvo, August Joki, Max Weisel, Optimo, Kevin Brosius, Planetbeing, Pytey, Michael Brennan, Daniel Gard, Michael Jones, Roxfan, MuscleNerd, np101137, UnterPerro, Jonathan Watmough, Youssef Francis, Bryan Henry, William DeMuro, Jeremy Sinclair, Arshad Tayyeb, Daniel Peebles, ChronicProductions, Greg Hartstein, Emanuele Vulcano, Sean Heber, Josh Bleecher Snyder, Eric Chamberlain, Steven Troughton-Smith, Dustin Howett, Dick Applebaum, Kevin Ballard, Hamish Allan, Kevin McAllister, Jay Abbott, Tim Grant Davies, Chris Greening, Landon Fuller, Wil Macaulay, Stefan Hafeneger, Scott Yelich, chrallelinder, John Varghese, Andrea Fanfani, J. Roman, jtbandes, Artissimo, Aaron Alexander, Christopher Campbell Jensen, rincewind42, Nico Ameghino, Jon Moody, Julián Romero, Scott Lawrence, Evan K. Stone, Kenny Chan Ching-King, Matthias Ringwald, Jeff Tentschert, Marco Fanciulli, Neil Taylor, Sjoerd van Geffen, Absentia, Nownot, Emerson Malca, Matt Brown, Chris Foresman, Aron Trimble, Paul Griffin, Paul Robichaux, Nicolas Haunold, Anatol Ulrich (hypnocode GmbH), Kristian Glass, Remy Demarest, Yanik Magnan, ashikase, Shane Zatezalo, Tito Ciuro, Jonah Williams of Carbon Five, Joshua Weinberg, biappi, Eric Mock, Jay Spencer, and everyone at the iPhone developer channels at irc.saurik.com and irc.freenode.net, among many others too numerous to name individually. Their techniques, suggestions, and feedback helped make this book possible. If I have overlooked anyone who helped contribute, please accept my apologies for the oversight.

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

## About the Author

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

# Preface

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

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

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

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

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

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

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

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

—Erica Sadun, November 2011

## What You'll Need

It goes without saying that, if you're planning to build iOS applications, you're going to need at least one of those iOS devices to test out your application, preferably a 3GS or later, a third-gen iPod touch or later, or any iPad. The following list covers the basics of what you need to begin:

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

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

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

## Your Roadmap to Mac/iOS Development

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

Once you know C and how to work with a compiler (something you'll learn in that basic C course), the rest should be easy. From there, you'll hop right on to Objective-C and learn how to program with that alongside the Cocoa frameworks. To help you along the way, my editor Chuck Toporek and I put together the flowchart shown in Figure P-1 to point you at some books of interest.

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

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

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

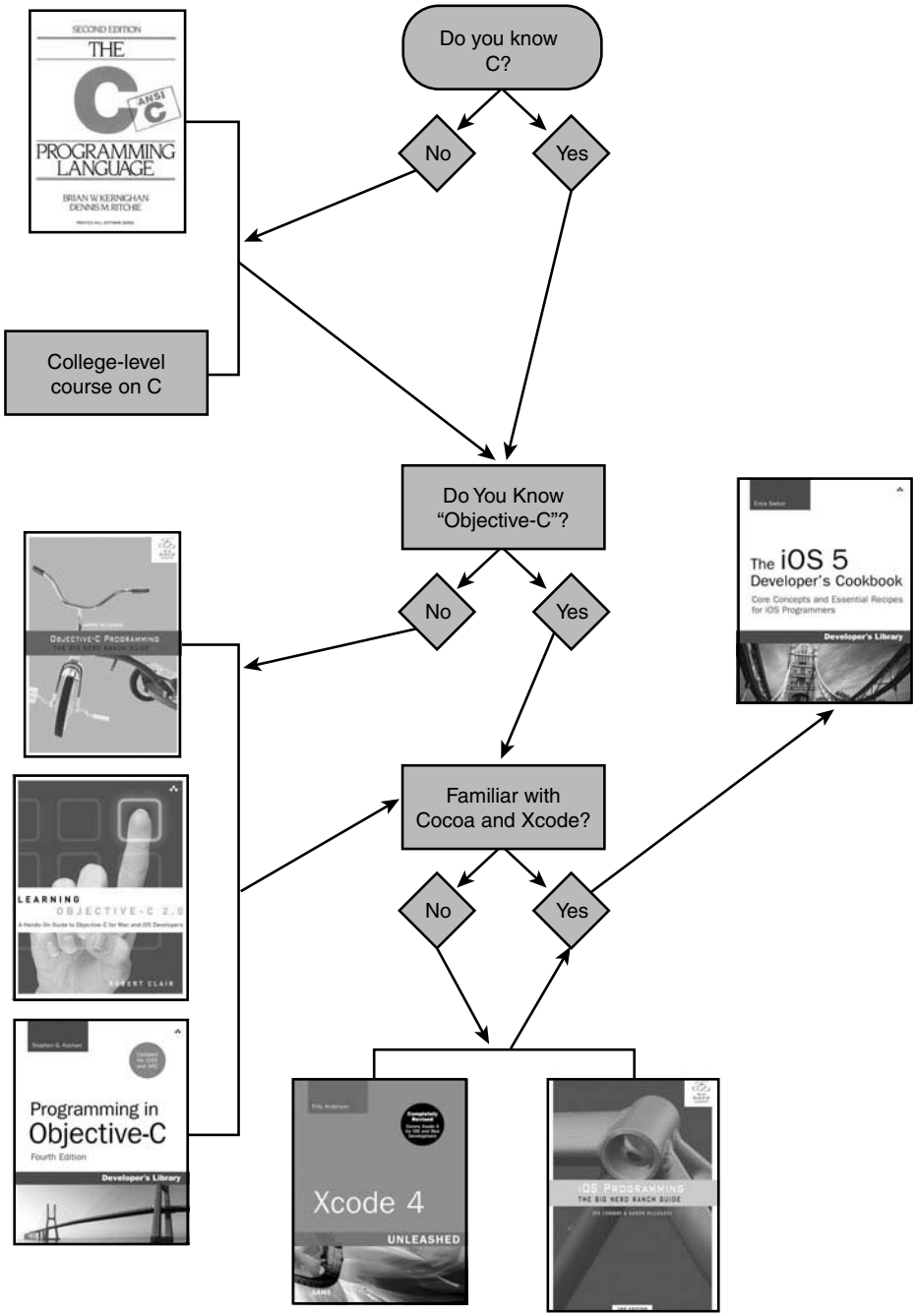


Figure P-1 What it takes to be an iOS programmer.

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

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

# How This Book Is Organized

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

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

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

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

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

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

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

## About the Sample Code

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

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

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

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

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

### Getting the Sample Code

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

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

### Getting Git

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

### Getting GitHub

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

## Contacting the Author

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

## Endnotes

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

2 Big Nerd Ranch: http://www.bignerdranch.com.

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

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

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

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

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

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

# 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.


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

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

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

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

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

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

## Using Navigation Controllers and Stacks

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

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

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

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

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

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

## The Navigation Item Class

The objects that populate the navigation bar are put into place using the `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";
```

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

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

## Modal Presentation

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

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

You can present a modal dialog in any of four ways, controlled by the `modalTransitionStyle` property of the presented view controller. The standard, `UIModalTransitionStyleCoverVertical`, slides the modal view up and over the current view controller. When dismissed it slides back down. `UIModalTransitionStyleFlipHorizontal` performs a back-to-front flip from right to left. It looks as if you're revealing the back side of the currently presented view. When dismissed, it flips back left to right. `UIModalTransitionStyleCrossDissolve` fades the new view in over the previous one. On dismissal, it fades back to the original view. Use `UIModalTransitionStylePartialCurl` to curl up content (in the way the Maps application does) to reveal a modal settings view "underneath" the primary view controller.

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

## Recipe: Building a Simple Two-Item Menu

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

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

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

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

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

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

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

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

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

```
}

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

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

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

# Recipe: Adding a Segmented Control

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

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

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

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


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

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

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

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

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

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

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

**Get This Recipe's Code**

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

# Recipe: Navigating Between View Controllers

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

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

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

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

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

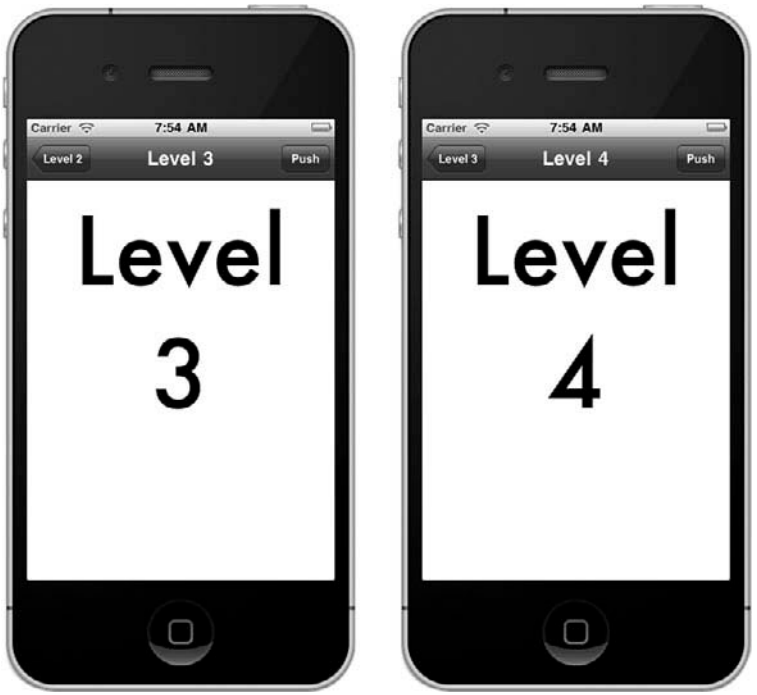


Figure 5-3 The navigation controller automatically creates properly labeled back buttons. After the Level 4 button is selected in the left interface, the navigation controller pushes the Level 4 view controller and creates the Level 3 back button in the right interface.

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

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

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

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

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

// Increment and push a controller onto the stack
- (void) 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", ));
}

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

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

```
    return self;
}

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

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

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

# Recipe: Presenting a Custom Modal Information View

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

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

```
[self 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.

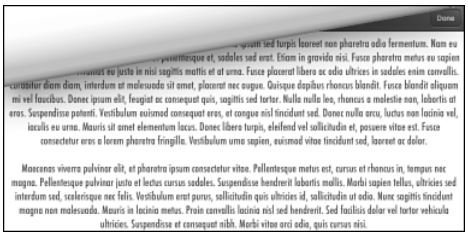


Figure 5-4 This modal view is built using `UIViewController` with a `UINavigationBar`.

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
```

```
{
    [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,
```

```
        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", ));


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

```
    if (IS_IPAD)
    {
        NSArray *presentationChoices =
           [NSArray arrayWithObjects:
               @"Full Screen", @"Page Sheet", @"Form Sheet", nil];
        UISegmentedControl *iPadStyleControl =
            [[UISegmentedControl alloc] 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.

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

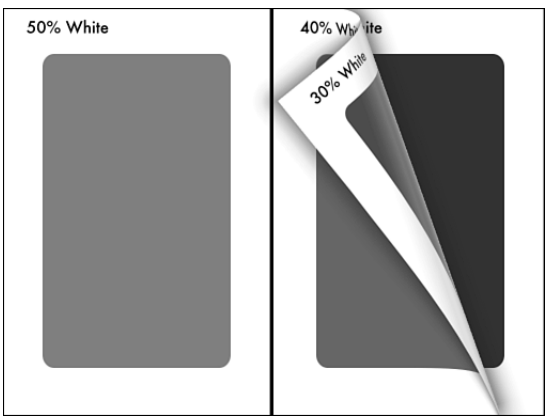


Figure 5-5 The `UIPageViewController` class creates virtual "books" from individual view controllers.

## Wrapping the Implementation

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

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

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

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

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

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

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

## Exploring the Recipe

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
        options:nil];

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

    return bc;
}
@end
```

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

# Recipe: Scrubbing Pages in a Page View Controller

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

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

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

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

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

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

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

- (void) viewDidLoad
{
    [super viewDidLoad];

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

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

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

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

## Recipe: Tab Bars

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

With tab bars, you don't push views the way you do with navigation bars. Instead, you assemble a collection of controllers (they can individually be `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 5-5 (left) shows the interface in its default mode, with the first four items and a More button displayed.

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

```
[[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;
```


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

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

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

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

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

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

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

    return image;
}

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

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

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

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

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

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

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

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

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

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

# Recipe: Remembering Tab State

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# Recipe: Building Split View Controllers

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

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


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

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

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

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

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

    return svc;
}
```

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

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

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

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

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

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

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

@implementation DetailViewController
@synthesize popoverController;

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

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

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

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

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


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

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

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

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

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

```
    return cell;
}

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

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

# Recipe: Creating Universal Split View/Navigation Apps

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

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

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

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

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

```
- (UINavigationController *) navWithColorViewController
{
    ColorViewController *colorViewController =
```

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

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

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

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

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

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

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

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

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

```
        initWithStyle: UITableViewCellStyleDefault
        reuseIdentifier:@"generic"];

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

    cell.accessoryType = IS_IPAD ?
        UITableViewCellAccessoryNone :
        UITableViewCellAccessoryDisclosureIndicator;

    return cell;
}

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

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

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

# Recipe: Custom Containers and Segues

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

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

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

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

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

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

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

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

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

    vcIndex = newIndex;
}
```

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


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

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

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

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

```
@implementation RotatingSegue
@synthesize goesForward;
@synthesize delegate;

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

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

    return imageLayer;
}

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

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

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

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

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

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

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

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

## Transitioning Between View Controllers

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

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

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

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

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

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

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

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

# One More Thing: Interface Builder and Tab Bar Controllers

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

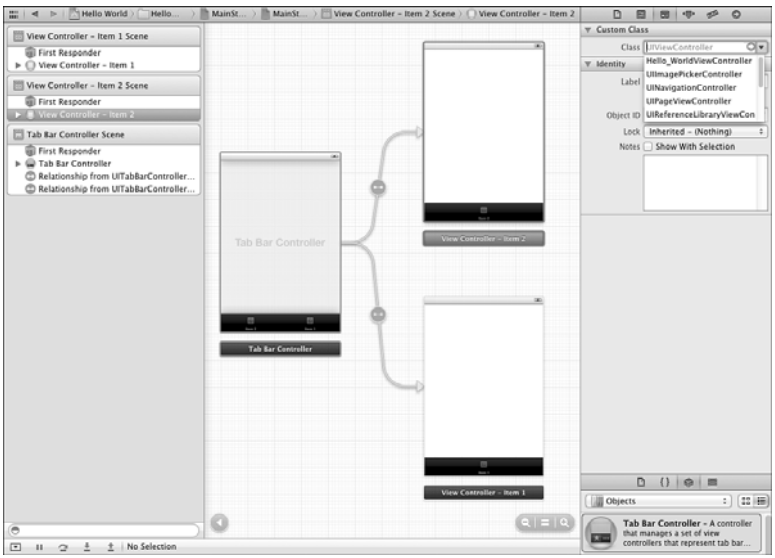


Figure 5-9 Interface Builder storyboards provide tools for laying out tab bar controllers, simplifying laying out what is essentially a logical and not a visual class, compared to what previous versions of Xcode allowed.

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


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

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

# Summary

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

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

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

*This page intentionally left blank*

# Index

## Symbols



## 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

*This page intentionally left blank*