Erica Sadun

Second Edition

The iPhone™ Developer's Cookbook

Building Applications with the iPhone 3.0 SDK

Developer's Library

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

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

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

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

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

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data:

Sadun, Erica.

The iPhone developer's cookbook : building applications with the iPhone 3.0 SDK / Erica Sadun. — 2nd ed.

p. cm.

Includes index.

ISBN 978-0-321-65957-6 (pbk. : alk. paper) 1. iPhone (Smartphone)—Programming. 2. Computer software—Development. 3. Mobile computing. I. Title.

QA76.8.I64S33 2010

004.167—dc22

2009042382

Copyright © 2010 Pearson Education, Inc.

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

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

ISBN-13: 978-0-321-65957-6
ISBN-10: 0-321-65957-0

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

First printing December 2009

Editor-in-Chief
Karen Gettman

Senior Acquisitions Editor
Chuck Toporek

Senior Development Editor
Chris Zahn

Managing Editor
Kristy Hart

Project Editor
Anne Goebel

Copy Editor
Geneil Breeze

Senior Indexer
Cheryl Lenser

Proofreader
Sheri Cain

Technical Reviewers
Joachim Bean,
Aaron Basil,
Tim Isted,
Mr. X,
Tim Burks,
Daniel Pasco,
Alex C. Schaefer,
John Muchow
(3 Sixty Software,
LLC Founder,
iPhoneDeveloper-
Tips.com),
Roberto Gamboni

Editorial Assistant
Romny French

Cover Designer
Gary Adair

Composition
Jake McFarland

Preface

Few platforms match the iPhone's unique developer technologies. The iPhone combines OS X-based mobile computing with an innovative multitouch screen, location awareness, an onboard accelerometer, and more. When Apple first introduced the iPhone SDK beta in March 2008, developers responded in droves, bringing Apple's servers to its knees. In less than a week, developers downloaded the iPhone SDK more than 100,000 times.

Since then, more than 50,000 applications have been delivered to the App Store for an audience that now exceeds 30 million iPhones and more than 20 million iPod touches. As the iPhone ecosystem continues to grow, *The iPhone Developer's Cookbook* will continue to evolve as an accessible resource for those new to iPhone programming.

What's New in This Edition?

If you purchased the first edition of this book, you might ask yourself, *Why do I need to buy the new edition, too?* The answer is pretty simple: Just compare the size of the two books. This new edition is more than 200% larger than the original edition. That's right, we've packed on almost 500 pages of new material so we could cover everything that's new to the iPhone 3.0 SDK, as well as expand on some of the topics covered in the first edition.

Some things you'll find new to this edition include chapters or coverage on

- How to use Xcode and Interface Builder
- An Objective-C jump-start tutorial
- Core Data for the iPhone
- MapKit and Core Location
- Using GameKit beyond games to add chat and Bonjour networking
- Advanced motion detection including shake-to-undo support
- The new search display controller class, along with custom table headers and footers
- Apple's new device capabilities specifications
- In-App purchasing with StoreKit
- Push notification, both from the client and server side
- Searching for and playing media from the onboard iPod library

- Video capture and editing, plus the new AV audio player and recorder classes
- How to leverage the Accessibility framework, including VoiceOver, in your app
- And much, much more!

You'll also notice that we've taken your feedback to heart. When the first edition came out, there was some confusion about who the target audience was for this book. Was it for new developers or experienced developers? Well, we've taken care of that, too. While this book is for experienced iPhone and Mac developers already familiar with Objective-C, Xcode, and the Cocoa frameworks, this new edition includes an "Objective-C Boot Camp" (see Chapter 3), and coverage of Xcode and Interface Builder, to help developers who have experience working in other languages (or on other platforms) quickly get oriented into the Mac/iPhone world.

While it is true that one book can't be everything to everyone, we're certainly giving it a shot in this new edition. We hope you like the changes you see throughout this bigger book, and if you do, be sure to post a review on Amazon or send me a note (erica@ericasadun.com).

Audience for This Book

This book is written for experienced developers who want to build apps for the iPhone and iPod touch. You should already be familiar with Objective-C, the Cocoa frameworks, and the Xcode Tools. That said, if you're new to the platform, this new edition of *The iPhone Developer's Cookbook* includes a quick-and-dirty introduction to Objective-C, along with an intro to the Xcode Tools, to help you quickly get up to speed.

New to the Mac or iPhone?

If you have some C experience, or have spent some time with another object-oriented language such as C++ or Java, we included a section in this Preface to help guide you down the road to being a Mac developer. Be sure to read the section "Your Roadmap to Mac/iPhone Development," later in this Preface.

Although each programmer brings different goals and experiences to the table, most iPhone developers end up solving similar tasks in their development work:

- "How do I build a table?"
- "How do I create a secure Keychain entry?"
- "How do I search the Address Book?"
- "How do I move between views?"
- "How do I use Core Location and the iPhone 3GS's magnetometer?"

And so on. If you've asked yourself these questions, then this book is for you. Complete with clear, fully documented examples, *The iPhone Developer's Cookbook* will get you up

to speed and working with the iPhone SDK in no time. Best of all, all of the code recipes in the book have been tested—and put to the test in real-world applications—offering you ready-to-use solutions for the apps you're building today.

What You'll Need

It goes without saying that, if you're planning to build apps for the iPhone or iPod touch, you're going to need at least one of those devices to test out your application. The following list covers the basics of what you need to begin programming for the iPhone or iPod touch:

- **Apple's iPhone SDK**—The latest version of the iPhone SDK can be downloaded from Apple's iPhone Dev Center (http://developer.apple.com/iphone). You must join Apple's (free) developer program before you download; however, if you plan to sell apps through the App Store, you will need to become a paid iPhone developer, which costs $99/year for individuals and $299/year for enterprise (i.e., 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 Discounts

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 if 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 Leopard or Snow Leopard**—Snow Leopard is recommended, as it offers access to Xcode 3.2 with its many new features like "Build and Analyze." 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 iPhone or iPod touch**—Although the iPhone SDK and Xcode include a simulator for you to test your applications in, you really do need to have an actual iPhone 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 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 3, "Objective-C Boot Camp," helps you get up to speed.

> **Note**
>
> Although the SDK supports development for the iPhone and iPod touch, as well as possible yet-to-be-announced platforms, this book refers to the target platform as iPhone for the sake of simplicity. When developing for the iPod touch, most of the examples in this book are applicable; however, certain features such as telephony and onboard speakers are not applicable to the iPod touch.

Your Roadmap to Mac/iPhone 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. There is, indeed, a lot you need to know to develop for the Mac and iPhone 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. While 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, I've 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 3 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, *Object-Oriented Programming with Objective-C 2.0*,[1] or you can opt to buy a book such as Stephen Kochan's *Programming in Objective-C 2.0* (Addison-Wesley, 2009).

[1] See http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/OOP_ObjC/OOP_ObjC.pdf.

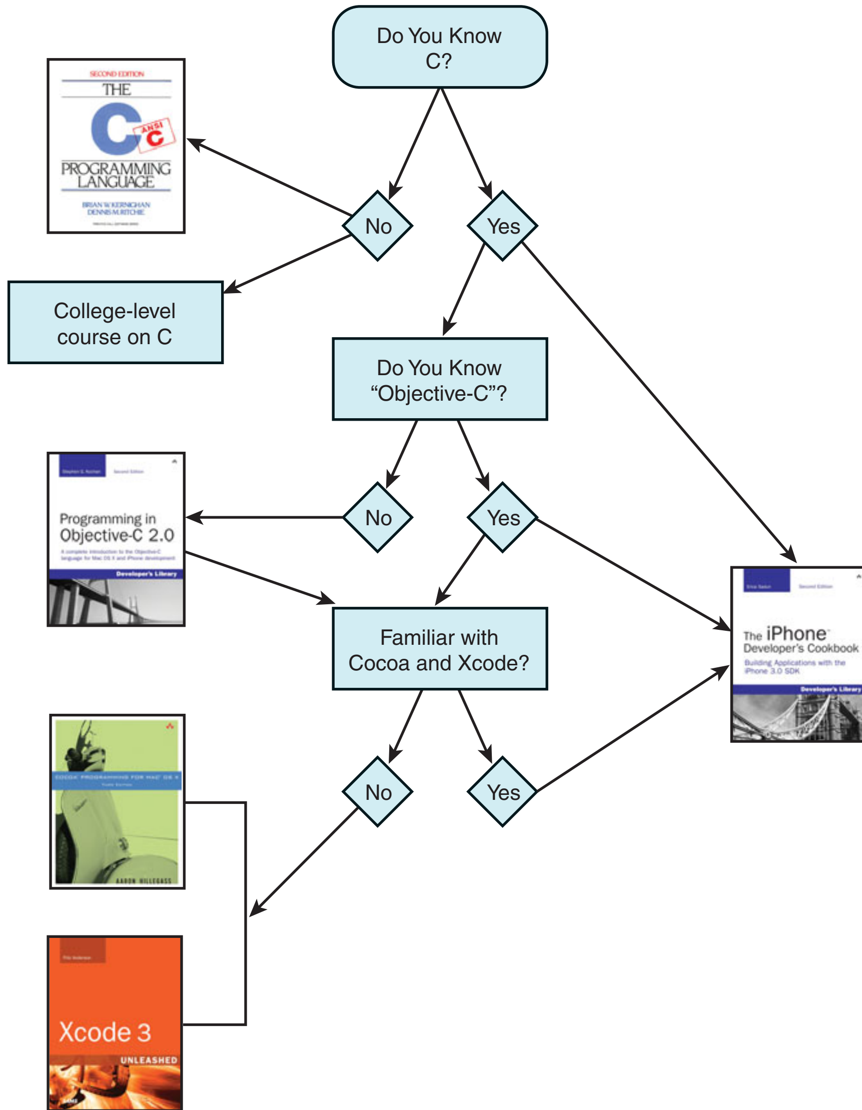

Figure P-1 What it takes to be an iPhone programmer.

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,[2] or if you prefer books, you can learn from the best. Aaron Hillegass, founder of the Big Nerd Ranch in Atlanta,[3] is the author of *Cocoa Programming for Mac OS X,* now in its third 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 3 Unleashed* from Sams Publishing. While the current edition doesn't cover iPhone-specific features of Xcode (which were introduced with Xcode 3.1), the book will give you a solid grounding in how to use Xcode as your development environment.

> **Note**
>
> There are plenty of other books from other publishers on the market, including the best-selling *Beginning iPhone 3 Development*, by Dave Marks and Jeff LaMarche (Apress, 2009), so don't just limit yourself to one book or publisher.

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 or C4, 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 iPhone developers face: laying out interface elements, responding to users, accessing local data sources, and connecting to the Internet. Each chapter groups related tasks together, 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 iPhone 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.

[2] 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).

[3] Big Nerd Ranch: http://www.bignerdranch.com.

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

- **Chapter 1, "Introducing the iPhone SDK"**—Chapter 1 introduces the iPhone SDK and explores the iPhone as a delivery platform, limitations and all. It explains the breakdown of the standard iPhone application and helps you get started with the iPhone Developer Portal.
- **Chapter 2, "Building Your First Project"**—Chapter 2 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 App Store.
- **Chapter 3, "Objective-C Boot Camp"**—If you're new to Objective-C as well as to the iPhone, you'll appreciate this basic skills chapter. Objective-C is the standard programming language for both the iPhone 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 3 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 4, "Designing Interfaces"**—Chapter 4 introduces the iPhone'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 like 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 iPhone 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 iPhone application screens.
- **Chapter 6, "Assembling Views and Animations"**—Chapter 6 introduces iPhone views, objects that live on your screen. You see how to lay out, create, and order your views to create backbones for your iPhone applications. You read about view hierarchies, geometries, and animations, features that bring your iPhone 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 iPhone 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 iPhone's built-in camera.

- **Chapter 8, "Gestures and Touches"**—On the iPhone, 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.
- **Chapter 9, "Building and Using Controls"**—Control classes provide the basis for many of the iPhone's interactive elements, including buttons, text fields, 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 a custom touch wheel.
- **Chapter 10, "Alerting Users"**—The iPhone offers many ways to provide users with a heads-up, from pop-up dialogs and progress bars to audio pings and status bar updates. Chapter 10 shows how to build these indications into your applications and expand your user-alert vocabulary. It introduces standard ways of working with these pop-up classes and offers solutions that allow you to craft more linear programs without explicit callbacks.
- **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, "Making Connections with GameKit and Bonjour"**—GameKit is Apple's new ad hoc networking solution for peer-to-peer connectivity. It's built on a technology called Bonjour that offers simple, no-configuration communications between devices. Chapter 12 introduces GameKit, allowing you to build games and utilities that move information back and forth between iPhones or between an iPhone and a desktop system. This chapter covers standard GameKit, introduces GameKit Voice for walkie-talkie-style voice chats, and offers some basic Bonjour programming that extends beyond GameKit limitations, allowing you to expand your iPhone communications to the desktop.
- **Chapter 13, "Networking"**—As an Internet-connected device, the iPhone 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 13 surveys common techniques for network computing and offering recipes that simplify day-to-day tasks. You read about

network reachability, synchronous and asynchronous downloads, working with the iPhone's secure keychain to meet authentication challenges, and more.

- **Chapter 14, "Device Capabilities"**—Each iPhone device represents a meld of unique, shared, momentary, and persistent properties. These properties include the device's current physical orientation, its model name, 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 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, "Audio, Video, and MediaKit"**—The iPhone is a media master; its built-in iPod features expertly handle both audio and video. The iPhone SDK exposes that functionality to developers. A rich suite of classes simplifies media handling via playback, search, and recording. Chapter 15 introduces recipes that use these classes, presenting media to your users and letting your users interact with that media. You see how to build audio and video players as well as audio and video recorders. You discover how to browse the iPod library and how to choose what items to play.
- **Chapter 16, "Push Notifications"**—When developers need to communicate directly with users, push notifications provide the solution. They deliver messages directly to the iPhone screen via a special Apple service. Push notifications let the iPhone display an alert, play a custom sound, or update an application badge. In this way, off-phone services connect with an iPhone-based client, letting them know about new data or updates. Chapter 16 introduces push notifications. In this chapter, you learn how push notifications work and dive into the details needed to create your own push-based system.
- **Chapter 17, "Using Core Location and MapKit"**—Core Location infuses the iPhone with on-demand geopositioning based on a variety of technologies and sources. MapKit adds interactive in-application mapping allowing users to view and manipulate annotated maps. With Core Location and MapKit, you can develop applications that help users meet up with friends, search for local resources, or provide location-based streams of personal information. Chapter 17 introduces these location-aware frameworks and shows you how you can integrate them into your iPhone applications.
- **Chapter 18, "Connecting to the Address Book"**—The iPhone's Address Book frameworks allow you to programmatically access and manage the contacts database. Chapter 18 introduces the Address Book and demonstrates how to use its frameworks in your applications. You read about accessing information on a contact-by-contact basis, how to modify and update contact information, and how to

use predicates to find just the contact you're interested in. This chapter also covers the GUI classes that provide interactive solutions for picking, viewing, and modifying contacts.

- **Chapter 19, "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 19 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.
- **Chapter 20, "StoreKit: In-App Purchasing"**—New to the 3.0 SDK, StoreKit offers in-app purchasing that integrates into your software. This chapter introduces StoreKit and shows you how to use the StoreKit API to create purchasing options for users. In this chapter, you read about getting started with StoreKit. You learn how set up products at iTunes Connect and localize their descriptions. And you see what it takes to create test users and how to work your way through various development/deployment hurdles. This chapter teaches you how to solicit purchase requests from users and how to hand over those requests to the store for payment. This chapter covers the entire StoreKit picture, from product creation to sales.
- **Chapter 21, "Accessibility and Other iPhone OS Services"**—Applications interact with standard iPhone services in a variety of ways. This chapter explores some of these approaches. Applications can define their interfaces to the iPhone's VoiceOver accessibility handler, creating descriptions of their GUI elements. They can create bundles to work with the built-in Settings applications so that users can access applications defaults using that interface. Applications can also declare public URL schemes allowing other iPhone applications to contact them and request services that they themselves offer. This chapter explores application service interaction. It shows you how you implement these features in your applications. You see how to build these service bridges through code, through Interface Builder, and through supporting files.
- **Appendix A, "Info.plist Keys"**—This appendix gathers together many of the keys available for the iPhone's Info.plist file, the file that describes an application to the iPhone operating system.

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 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 5 or 7 or 9 individual files at once.

Offering a single file concentrates that story, allowing access to that idea in a single chunk.

These samples are not intended as stand-alone 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 on offer. Those samples 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 samples for this book use a single application identifier, com.sadun.helloworld. You need to replace this identifier with one that matches your provision profile. This book uses one identifier to avoid clogging up your iPhone with dozens of samples at once. Each sample replaces the previous one, ensuring that SpringBoard remains relatively uncluttered. If you want to install several samples at once, simply edit the identifier, adding a unique suffix, such as com.sadun.helloworld.tableedits.

Getting the Sample Code

The source code for this book can be found at the open source GitHub hosting site at http://github.com/erica/iphone-3.0-cookbook-/tree. 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 us know.

We'd be 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 Web site, allowing you to copy and modify the Cookbook repository or create your own open source iPhone 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 iPhone developers. Please feel free to visit, download software, read documentation, and leave your comments.

16

Push Notifications

When developers need to communicate directly with users, push notifications provide the solution. They deliver messages directly to the iPhone screen via a special Apple service. Push notifications let the iPhone display an alert, play a custom sound, or update an application badge. In this way, off-phone services connect with an iPhone-based client, letting them know about new data or updates. Unlike most other iPhone development arenas, nearly all the push story takes place off the phone. Developers must create Web-based services to manage and deploy these updates. In this chapter, you learn how push notifications work and dive into the details needed to create your own push-based system.

Introducing Push Notifications

Push notifications, also called *remote notifications*, refer to a kind of message sent to iPhones by an outside service. These push-based services work with any kind of application that normally checks for information updates. For example, a service might poll for new direct messages on Twitter or respond to sensors for in-home security systems. When new information becomes available for a client, the service pushes that update through Apple's remote notification system. The notification transmits directly to the phone, which has registered to receive those updates.

The key to push is that these messages originate from outside the device itself. They are part of a client-server paradigm that lets Web-based server components communicate with iPhone clients through an Apple-supplied service. With push, developers can send nearly instant updates to iPhones that don't rely on users launching a particular application. Instead, processing occurs on the server side of things. When push messages arrive, the iPhone client can respond by displaying a badge, playing a sound, and/or showing an alert box.

According to Apple, battery life is the single biggest reason for endorsing push notification. When many applications run at once via background processes, these processes can put an undue burden on a device battery, shortening the amount of time available before a recharge is needed. With push, applications can learn about new updates even when

they're not running. This lets Apple enforce its strict one-third-party-application-at-a-time policy while at the same time allowing users to receive notifications that are tied to application state changes.

Moving application logic to a server also limits the client-side complexity. Offsite processing provides energy savings for iPhone-based applications. They can now rely on push rather than using the iPhone's local CPU resources to monitor and react to important information changes.

Push's reason for being is not only tied into local resources. It also offers a valuable solution for communicating with Web-based services that goes beyond poll-and-update applications. For example, push might allow you to hook into a recommendation service that produces restaurant suggestions even when an application isn't running or to a calendar service that sends you reminder notices about an upcoming appointment. So don't think about push solely as a battery saver. Also think about it as a conduit for Web services as well.

From social networking to monitoring RSS feeds, push lets iPhone users keep on top of asynchronous data feeds. It offers a powerful solution for connecting iPhone clients to Web-based systems of all kinds. With push, the services you write can connect to your installed iPhone base and communicate updates in a clean, functional manner.

How Push Works

Push notifications aren't just a general way to talk directly to iPhones at will. They are tied to specific applications and require several security checks. A push server can only communicate with those iPhones that are running its application, that are online, and that have opted to receive remote messages. Users have the ultimate say in push updates. They can allow or disallow that kind of communication, and a well-written application lets users opt-in and opt-out of the service at will.

The chain of communication between server and client works like this. Push providers deliver message requests through a central Apple server and via that server to their iPhone clients. In normal use, the server triggers on some event (like new mail or an upcoming appointment) and generates notification data aimed at a specific iPhone device. It sends this message request to the Apple Push Notification Service (APNS). This notification uses JSON formatting and is limited to 256 bytes each, so the information that can be pushed through on that message is quite limited. This formatting and size ensures that APNS limits bandwidth to the tightest possible configuration.

APNS offers a centralized system that negotiates communication with iPhones in the real world. It passes the message through to the designated iPhone. A handler on the iPhone decides how to process the message. As Figure 16-1 shows, push providers talk to APNS, sending their message requests, and APNS talks to phones, relaying those messages to handlers on the unit.

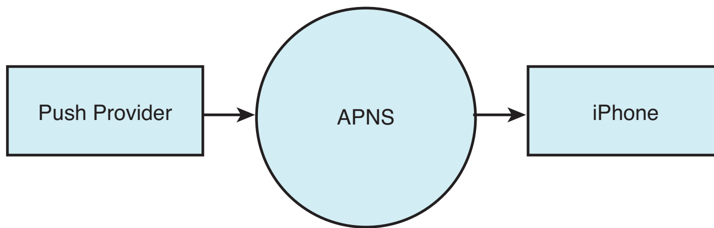

Figure 16-1 Providers send messages through Apple's centralized push notification service to communicate with an iPhone.

Multiple Provider Support

APNS was built to support multiple provider connections, allowing many services to communicate with it at once. It offers multiple gateways into the service so that each push service does not have to wait for availability before sending its message. Figure 16-2 illustrates the many-to-many relationship between providers and iPhones. APNS allows providers to connect at once through multiple gateways. Each provider can push messages to many different iPhones.

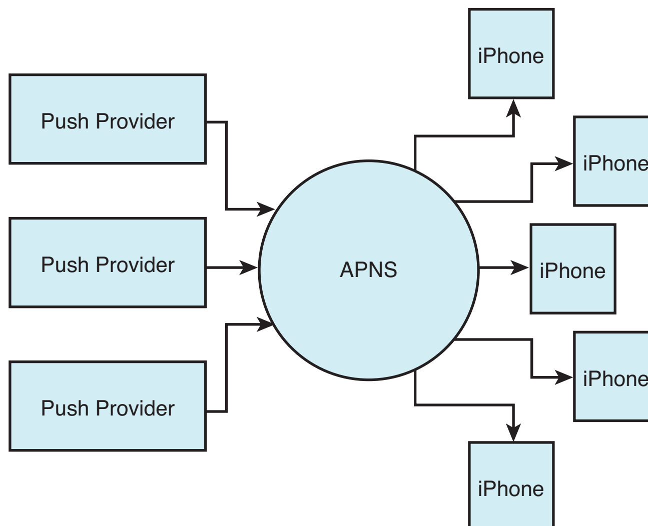

Figure 16-2 Apple's Push Notification Service offers many gateways on its provider-facing side, allowing multiple providers to connect in parallel. Each push provider may connect to any number of iPhone devices.

Security

Security is a primary component of remote notifications. The push provider must sign up for a secure sockets layer certificate for each application it works with. Services cannot communicate with APNS unless they authenticate themselves with this certificate. They must also provide a unique key called a token that identifies both the phone to message and the application to notify.

After receiving an authenticated message and device token, APNS contacts the phone in question. Each iPhone or member of the iPhone family such as the iPod touch must be online in some way to receive a notification. They can be connected to a cellular data network or to a Wi-Fi hotspot. APNS establishes a connection with the device and relays the notification request. If the device is offline and the APNS server cannot make a connection, the notification is queued for later delivery.

Upon receiving the request, the iPhone performs a number of checks. Push requests are ignored when the user disables push updates for a given application; users can do so in the Settings application on their iPhone. When updates are allowed, and only then, the iPhone determines whether the client application is currently running. If so, it sends a message directly to the running application via the application delegate. If not, it performs some kind of alert, whether displaying text, playing a sound, or updating a badge.

When an alert displays, users typically have the option to close the alert or tap View. If they choose View, the iPhone launches the application in question and sends it the notification message that it would have received while running. If the user taps Close, the notification gets ignored and the application does not launch.

This pathway, from server to APNS to iPhone to application, forms the core flow of push notifications. Each stage moves the message along the way. Although the multiple steps may sound extensive, in real life the notification arrives almost instantaneously. Once you set up your certificates, identifiers, and connections, the actual delivery of information becomes trivial. Nearly all the work lies in first setting up that chain and then in producing the information you want to deliver.

Make sure you treat all application certificates and device tokens as sensitive information. When storing these items on your server, you must ensure that they are not generally accessible. Should this information hit the wild, it could be exploited by third parties. This would likely result in Apple revoking your SSL push certificate. This would disable all remote notifications for any apps you have sold and might force you to pull the application from the store.

Push Limitations

Push notifications are not reliable. In reality, they can be fairly flaky. Apple does not guarantee the delivery of each notification or the order in which notifications arrive. Never send vital information by push. Reserve this feature for helpful notifications that update the user, but that the user can miss without consequence.

Items in the push delivery queue may be displaced by new notifications. That means that notifications may have to compete and may get lost along the way. Although Apple's

feedback service reports failed deliveries (i.e., messages that cannot be properly sent through the push service, specifically to applications that have been removed from a device), you cannot retrieve information regarding bumped notifications. From the APN service point of view, a lost message was still successfully "delivered."

Provisioning Push

To start push development, you must visit Apple's iPhone Developer Program portal. This portal is located at http://developer.apple.com/iphone/manage/overview/index.action. Sign in with your iPhone developer credentials to gain access to the site. Here at the portal, you can work through the steps needed to create a new application identifier that can be associated with a push service.

There's a fair amount of detail involved. Make sure you hit every point. The following sections walk you through the process. You see how to create a new identifier, generate a certificate, and request a special provisioning profile so you can build push-enabled applications. Without a push-enabled profile, your application will not be able to receive remote notifications.

Generate a New Application Identifier

At the developer portal, click on App IDs. You'll find this option in the column on the left side of the Web page. This opens a page that allows you to create new application identifiers. Each push service is based on a single identifier, which you must create and then set to allow remote notification. You cannot use a wild-card identifier with push applications; every push-enabled app demands a unique identifier.

In the App IDs section, click Add ID; this button appears at the top-right of the Web page. Once clicked, the site opens a new Create App ID page. Enter a name that describes your new identifier, such as "My First Push Application" and a new bundle identifier.

These IDs typically use reverse domain patterns like com.*domainname*.*appname*, such as com.sadun.firstpushapp. The identifier must be unique and may not conflict with any other registered application identifier in Apple's system. The bundle identifier for your application (set in the Info.plist file) needs to exactly match the last part of this string. If, for example, the ID in the portal is XYZZYPLUGH.com.sadun.pushapp, then the bundle identifier of your app should be com.sadun.pushapp.

Click Submit to add the new identifier. This adds the app ID irrevocably to Apple's system, where it is now registered to you. You return to the App ID page with its list of identifiers and are now ready to establish that identifier as push compliant.

> **Note**
>
> Apple does not provide any way to remove an application identifier from the program portal once it has been created.

Generate Your SSL Certificate

On the App ID page, you can see which identifiers work with push and which do not. The Apple Push Notification column shows whether push has been enabled for each app ID. The three states for this column are

- Unavailable (gray) for IDs that are no longer available
- Available (yellow) for apps that can be used with push but that haven't yet been set up to do so
- Enabled (green) for apps that are ready for push

You'll find two dots next to each application identifier—one for Development and another for Production. These options are configured separately. Locate your new app ID, make sure the yellow Available for Development is shown, and click Configure. This option appears in the rightmost column. When clicked, the browser opens a new Configure App ID page that permits you to associate your identifier with the push notification service.

An Enable Push Notification Services check box appears about halfway down the page. Check this box to start the certificate creation process. Once checked, the two Configure buttons on the right side of the page become enabled. Click that button. A page of instructions loads, showing you how to proceed. It guides you through creating a secure certificate that will be used by your server to sign messages it sends to the APNS.

As instructed, launch the Keychain Access application. This application is located on your Macintosh in the /Applications/Utilities folder. Once launched, choose Keychain Access > Certificate Assistant > Request a Certificate From a Certificate Authority (see Figure 16-3). You need to perform this step again even if you've already created previous requests for your developer and distribution certificates. The new request adds information that uniquely identifies the SSL certificate.

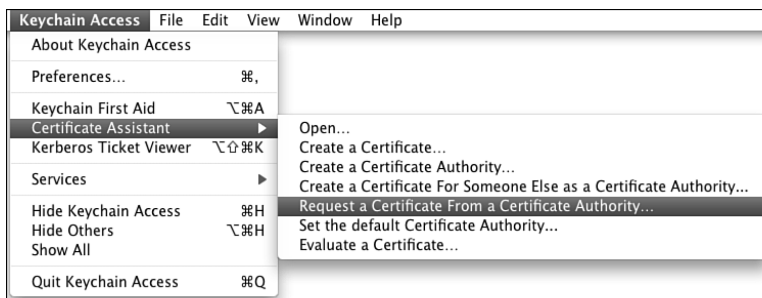

Figure 16-3 Create a new certificate request even though you've probably already done so in the past for your developer and distribution certificates.

Once the Certificate Assistant opens, enter your e-mail address and add a recognizable common name such as First Push App. This common name is important. It will come in

handy for the future, so choose one that is easy to identify and that describes your project accurately. The common name lets you distinguish otherwise similar looking keychain items from each other in the OS X Keychain Access utility.

After specifying a common name, choose Saved to Disk and click Continue. The Certificate Assistant prompts you to choose a location to save to (the Desktop is handy). Click Save, wait for the certificate to be generated, and then click Done. Return to your Web browser and click Continue. You are now ready to submit the certificate-signing request.

Click Choose File and navigate to the request you just generated. Select it and click Choose. Click Generate to build your new SSL push service certificate. This can take a minute or two, so be patient and do not close the Web page. Once the certificate has been generated, click Continue. Download the new certificate by clicking Download Now. Finally, click Done. You return to the App ID page where a new, green Enabled indicator should appear next to your app ID (see Figure 16-4). Apple also e-mails you a confirmation that your certificate request was approved.

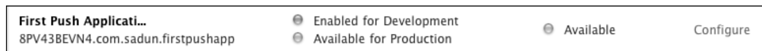

Figure 16-4 The Enabled label appears next to application identifiers that have been approved for push notification. You must create separate SSL certificates for development and for production.

> **Note**
>
> Should you ever need to download your SSL certificate again, click Configure to return to the Configure App ID page. There, you can click Download to request another copy.

If you plan to run your Push Server from your Macintosh, add the new certificate to your keychain by double-clicking the downloaded .cer file. It will be added to your login keychain and appear in your Certificates. Figure 16-5 shows that you can identify the certificate by clicking the small triangle next to it to reveal the common name you used when creating the certificate request.

Push-Specific Provisions

You cannot use wild-card provisions for push-enabled applications. Instead, you must create a single provision for just that application. This means that if you intend to create development, ad hoc, and distribution versions of your app, you must request three new mobile provision files in addition to whatever provisions you have already created for other work.

Go to the Provisioning section of the developer portal and choose whether to create a Development or Distribution profile by clicking the appropriate tab. Click Add Profile to begin creating your new provision. A Create iPhone Provisioning Profile page opens, whether for development or distribution.

- **Development Provision**—For development, enter a profile name such as "My First Push App Development." Check the certificate you will be using and choose

your application identifier from the pop-up list. Select the devices you will be using and click Submit.

- **Distribution Provision**—For distribution, select App Store or Ad Hoc. Enter a name for your new provision such as "My First Push App Distribution" or "My First Push App Ad Hoc." Choose your application identifier from the pop-up list. For Ad Hoc distribution only, select the devices to include in your provision. Click Submit to finish.

Figure 16-5 Identify which Push Service SSL certificate you are dealing with by clicking the down arrow. This reveals the common name used to generate the original certificate request.

It may take a minute or two for your profile to generate. Wait a short while and reload the page. The provision status should change from Pending to Active. Download your new provision and add it to Xcode by dragging it onto the Xcode application icon.

Registering Your Application

Signing an application with a push-compatible mobile provision is just the first step to working with push notifications. The application must request to register itself with the iPhone's remote notification system. You do this with a single `UIApplication` call, as follows. The application did finish launching delegate method provides a particularly convenient place to call this.

```
[[UIApplication sharedApplication]
    registerForRemoteNotificationTypes:types];
```

This call tells the iPhone OS that your application wants to accept push messages. The types you pass specify what kinds of alerts your application will receive. The iPhone offers three types of notifications:

- **`UIRemoteNotificationTypeBadge`**—This kind of notification adds a red badge to your application icon on SpringBoard.
- **`UIRemoteNotificationTypeSound`**—Sound notifications let you play sound files from your application bundle.
- **`UIRemoteNotificationTypeAlert`**—This style displays a text alert box in SpringBoard or any other application with a custom message using the alert notification.

Choose the types you want to use and `or` them together. They are bit flags, which combine to tell the notification registration process how you want to proceed. For example, the following flags allow alerts and badges but not sounds.

```
types = UIRemoteNotificationTypeBadge | UIRemoteNotificationTypeAlert;
```

Performing the registration updates user settings. As Figure 16-6 shows, a Notifications pane gets added to Settings if one has not already been created by another program. Your application appears as a subpane, offering user control over notification types. Switches appear only for those notifications that you registered. If your application uses just two types, then two switches appear in that pane. Figure 16-6 shows an application that has registered for all three.

To remove your application from active participation in push notifications, send `unregisterForRemoteNotifications`. This unregisters your application for all notification types and does not take any arguments.

```
[[UIApplication sharedApplication] unregisterForRemoteNotifications];
```

Retrieving the Device Token

Your application cannot receive push messages until it generates and delivers a device token to your server. It must send that device token to the offsite service that pushes the actual notifications. Recipe 16-1, which follows this section, does not implement server functionality. It provides only the client software.

A token is tied to one device. In combination with the SSL certificate, it uniquely identifies the iPhone and can be used to send messages back to the phone in question. Be aware that device tokens can change after you restore iPhone firmware.

Device tokens are created as a byproduct of registration. Upon receiving a registration request, the iPhone OS contacts the Apple Push Notification Service. It uses a secure socket layer (SSL) request. Somewhat obviously, the unit must be connected to the Internet. If it is not, the request will fail. The iPhone forwards the request to APNS and waits for it to respond with a device token.

APNS builds the device token and returns it to the iPhone OS, which in turn passes it back to the application via an application delegate callback, namely

```
application:didRegisterForRemoteNotificationsWithDeviceToken:
```

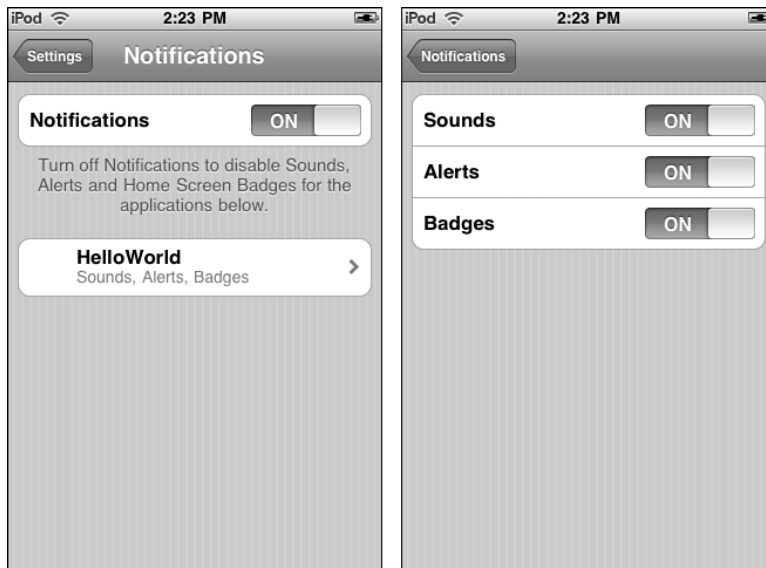

Figure 16-6 Remote notification controls appear for each application that has registered with the iPhone for push support. These controls are removed when applications unregister.

Your application must retrieve this token and pass it to the provider component of your service, where it needs to be stored securely. Anyone who gains access to a device token and the application's SSL certificate could spam messages to iPhones. You must treat this information as sensitive and protect it accordingly.

> **Note**
>
> At times, the token may take time to generate. Consider designing around possible delays into your application by registering at each application run. Until the token is created and uploaded to your site, you will not be able to provide remote notifications to your users.

Handling Token Request Errors

At times, APNS is unable to create a token or your device may not be able to send a request. For example, you cannot generate tokens from the simulator. A `UIApplicationDelegate` method `application: didFailToRegisterForRemote` ➥`NotificationsWithError:` lets you handle these token request errors. For the most part, you'll want to retrieve the error and display it to the user.

```
// Provide a user explanation for when the registration fails
- (void)application:(UIApplication *)application
    didFailToRegisterForRemoteNotificationsWithError:(NSError *)error
{
```

```
    UITextView *tv = (UITextView *)[[application keyWindow]
        viewWithTag:TEXTVIEWTAG];
    NSString *status = [NSString stringWithFormat:
        @"%@\nRegistration failed.\n\nError: %@", pushStatus(),
        [error localizedDescription]];
    tv.text = status;
}
```

Responding to Notifications

The iPhone uses a set chain of operations (see Figure 16-7) in responding to push notifications. When an application is running, the notification is sent directly to a `UIApplicationDelegate` method, `application: didReceiveRemoteNotification:`. The payload, which is sent in JSON format, is converted automatically into an `NSDictionary`, and the application is free to use the information in that payload however it wants. As the application is already running, no further sounds, badges, or alerts are invoked.

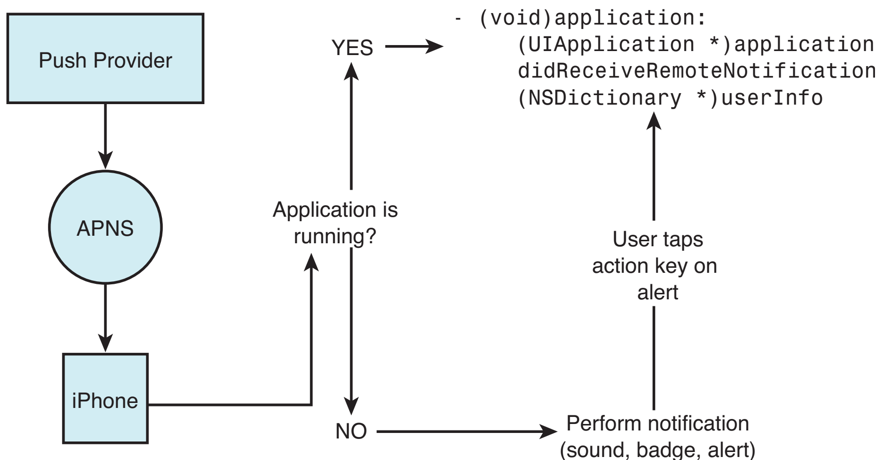

Figure 16-7 Visible and audible notification are only presented when the application is not running. Should the user click on an alert's action key (normally View), the application launches and the payload is sent as a notification to the `UIApplicationDelegate`.

```
// Handle an actual notification
- (void)application:(UIApplication *)application
    didReceiveRemoteNotification:(NSDictionary *)userInfo
{
    UITextView *tv = (UITextView *)[[application keyWindow]
```

```
        viewWithTag:TEXTVIEWTAG];
    NSString *status = [NSString stringWithFormat:
        @"Notification received:\n%@",[userInfo description]];
    tv.text = status;
    NSLog(@"%
}
```

When an application is not running, the iPhone performs all requested notifications that are allowed by registration and by user settings. These notifications may include playing a sound, badging the application, and/or displaying an alert. Playing a sound can also trigger iPhone vibration when a notification is received.

In the case of an alert, all two-buttoned alerts offer a pair of choices. The user can tap Close (the leftmost button) and close the alert or tap the alert's action key (the rightmost button) and launch the app. Upon launching, the application delegate receives the same remote notification callback that an already-running application would have seen (see Figure 16-8). Alerts appear on the lock screen when the iPhone is locked.

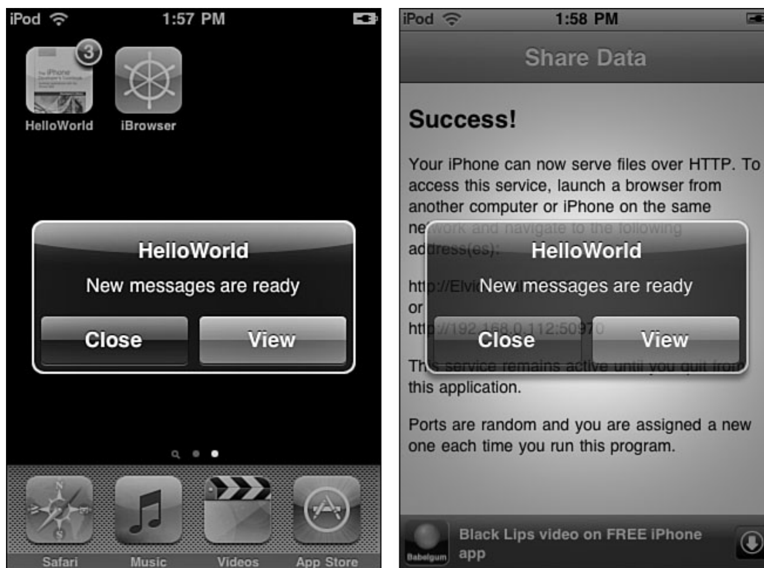

Figure 16-8 Remote alerts can appear in SpringBoard (left) or in third-party applications (right). Users may Close the alert or, by pressing the action button on the right, switch to the notifying application. In this case, that application is HelloWorld, whose name is clearly seen on the alert. The action button text is customizable.

Recipe: Push Client Skeleton

Recipe 16-1 introduces a basic client that allows users to register and unregister for push notifications. The interface (shown in Figure 16-9) uses three switches that control the services to be registered. When the application launches, it queries the app's enabled remote notification types and updates the switches to match. Thereafter, the client keeps track of registrations and unregistrations, adjusting the switches to keep sync with the reality of the settings.

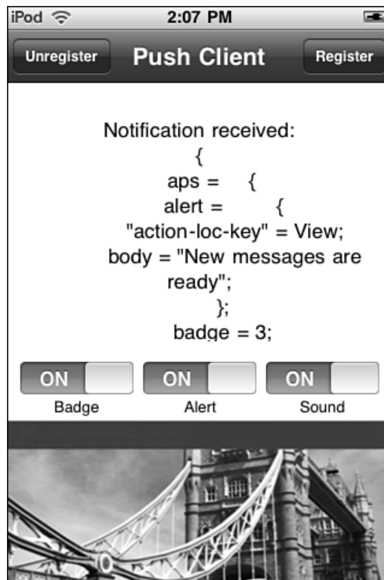

Figure 16-9 The Push Client skeleton introduced in Recipe 16-1 lets users specify which services they want to register.

Two buttons at the top left and right of the interface let users unregister and register their application. As mentioned earlier in this chapter, unregistering disables all services associated with the app. It provides a clean sweep. In contrast, registering apps requires flags to indicate which services are requested.

When requesting new services, the user is always prompted to approve. Figure 16-10 shows the dialog that appears. The user must confirm by explicitly granting the application permission. If the user does not, by tapping Don't Allow, the flags remain at their previous settings.

Unfortunately, the confirmation dialog does not generate a callback when it is dismissed, regardless of whether the user agreed or not. To catch this event, you can listen for

a general notification (`UIApplicationDidBecomeActiveNotification`) that gets generated when the dialog returns control to the application. It's a hack and is not guaranteed to work in the long term, but at the time of writing, Apple has not provided any other way to know when the user responded and how the user responded. In Recipe 16-1, the `confirmationWasHidden:` method catches this notification and updates the switches to match any new registration settings.

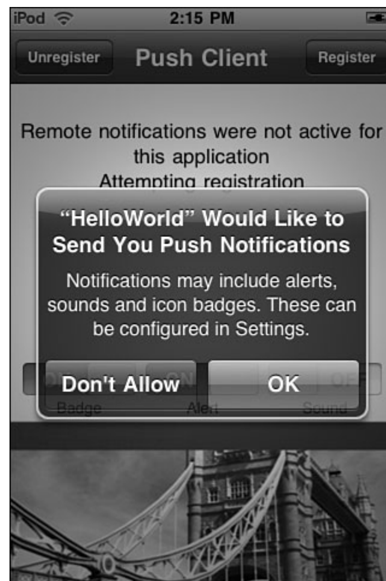

Figure 16-10 Users must explicitly grant permission for an application to receive remote notifications.

Being something of a skeletal system, this push client doesn't actually respond to push notifications beyond showing the contents of the user info payload that gets delivered. Figure 16-9 illustrates the actual payload that was sent in Figure 16-10. This display is performed in the `application: didReceiveRemoteNotification:` method in the application delegate.

> **Note**
>
> The three sound files included in the online sample project (ping1.caf, ping2.caf, and ping3.caf) let you test sound notifications with real audio.

Recipe 16-1 **Push Client Skeleton**

```
#define TEXTVIEWTAG    11

NSString *pushStatus ()
{
```

```
    return [[UIApplication sharedApplication]
        enabledRemoteNotificationTypes] ?
        @"Remote notifications were active for this application" :
        @"Remote notifications were not active for this application";
}

@implementation TestBedController

// Fetch the current switch settings
- (NSUInteger) switchSettings
{
    NSUInteger which = 0;
    if ([(UISwitch *)[self.view viewWithTag:101] isOn])
        which = which | UIRemoteNotificationTypeBadge;
    if ([(UISwitch *)[self.view viewWithTag:102] isOn])
        which = which | UIRemoteNotificationTypeAlert;
    if ([(UISwitch *)[self.view viewWithTag:103] isOn])
        which = which | UIRemoteNotificationTypeSound;
    return which;
}

// Change the switches to match reality
- (void) updateSwitches
{
    NSUInteger rntypes = [[UIApplication sharedApplication]
        enabledRemoteNotificationTypes];
    [(UISwitch *)[self.view viewWithTag:101] setOn:
        (rntypes & UIRemoteNotificationTypeBadge)];
    [(UISwitch *)[self.view viewWithTag:102] setOn:
        (rntypes & UIRemoteNotificationTypeAlert)];
    [(UISwitch *)[self.view viewWithTag:103] setOn:
        (rntypes & UIRemoteNotificationTypeSound)];
}

// Little hack work-around to catch the end when the
// confirmation dialog goes away. Apple has given this
// the thumbs up for use after I filed a technical query
- (void) confirmationWasHidden: (NSNotification *) notification
{
    [[UIApplication sharedApplication]
        registerForRemoteNotificationTypes: [self switchSettings]];
    [self updateSwitches];
}

// Register application for the services set out by the switches
- (void) doOn
{
```

```
    UITextView *tv = (UITextView *)[self.view viewWithTag:TEXTVIEWTAG];
    if (![self switchSettings])
    {
        tv.text = [NSString stringWithFormat:
            @"%@\nNothing to register. Skipping.\n\
            (Did you mean to press Unregister instead?)",
            pushStatus()];
        [self updateSwitches];
        return;
    }

    NSString *status = [NSString stringWithFormat:
        @"%@\nAttempting registration", pushStatus()];
    tv.text = status;
    [[UIApplication sharedApplication]
         registerForRemoteNotificationTypes:[self switchSettings]];
}

// Unregister application for all push notifications
- (void) doOff
{
    UITextView *tv = (UITextView *)[self.view viewWithTag:TEXTVIEWTAG];
    NSString *status = [NSString stringWithFormat:
        @"%@\nUnregistering.", pushStatus()];
    tv.text = status;

    [[UIApplication sharedApplication]
        unregisterForRemoteNotifications];
    [self updateSwitches];
}

- (void)loadView
{
    self.view = [[[NSBundle mainBundle] loadNibNamed:@"view" owner:self
        options:NULL] objectAtIndex:0];
    self.title = @"Push Client";

    self.navigationItem.rightBarButtonItem = BARBUTTON(@"Register",
        @selector(doOn));
    self.navigationItem.leftBarButtonItem = BARBUTTON(@"Unregister",
        @selector(doOff));
    [self updateSwitches];
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(confirmationWasHidden)
        name:@"UIApplicationDidBecomeActiveNotification" object:nil];
}
@end
```

```
@interface SampleAppDelegate : NSObject <UIApplicationDelegate>
@end

@implementation SampleAppDelegate
- (void) showString: (NSString *) aString
{
    UITextView *tv = (UITextView *)[[[UIApplication sharedApplication]
        keyWindow] viewWithTag:TEXTVIEWTAG];
    tv.text = aString;
}

// Retrieve the device token
- (void)application:(UIApplication *)application
        didRegisterForRemoteNotificationsWithDeviceToken:
           (NSData *)deviceToken
{
    NSUInteger rntypes = [[UIApplication sharedApplication]
       enabledRemoteNotificationTypes];
    NSString *results = [NSString stringWithFormat:
        @"Badge: %@, Alert:%@, Sound: %@",
        (rntypes & UIRemoteNotificationTypeBadge) ? @"Yes" : @"No",
        (rntypes & UIRemoteNotificationTypeAlert) ? @"Yes" : @"No",
        (rntypes & UIRemoteNotificationTypeSound) ? @"Yes" : @"No"];

    NSString *status = [NSString stringWithFormat:
        @"%@\nRegistration succeeded.\n\nDevice Token: %@\n%@",
        pushStatus(), deviceToken, results];
    [self showString:status];
    NSLog(@"deviceToken %@", deviceToken);
}

// Provide a user explanation for when the registration fails
- (void)application:(UIApplication *)application
       didFailToRegisterForRemoteNotificationsWithError:
          (NSError *)error
{
    NSString *status = [NSString stringWithFormat:
        @"%@\nRegistration failed.\n\nError: %@", pushStatus(),
        [error localizedDescription]];
    [self showString:status];
    NSLog(@"Error in registration. Error: %@", error);
}

// Handle an actual notification
- (void)application:(UIApplication *)application
        didReceiveRemoteNotification:(NSDictionary *)userInfo
```

```
{
    NSString *status = [NSString stringWithFormat:
        @"Notification received:\n%@",[userInfo description]];
    [self showString:status];
    CFShow([userInfo description]);
}

// Report the notification payload when launched by alert
- (void) launchNotification: (NSNotification *) notification
{
    [self performSelector:@selector(showString)
        withObject:[[notification userInfo] description]
        afterDelay:1.0f];
}

- (void)applicationDidFinishLaunching:(UIApplication *)application {
    UIWindow *window = [[UIWindow alloc]
        initWithFrame:[[UIScreen mainScreen] bounds]];
    UINavigationController *nav = [[UINavigationController alloc]
        initWithRootViewController:[[TestBedController alloc] init]];
    [window addSubview:nav.view];
    [window makeKeyAndVisible];

    // Listen for remote notification launches
    [[NSNotificationCenter defaultCenter] addObserver:self
        selector:@selector(launchNotification)
        name:@"UIApplicationDidFinishLaunchingNotification"
        object:nil];
}
@end
```

Get This Recipe's Code

To get the code used for this recipe, go to http://github.com/erica/iphone-3.0-cookbook-, or if you've downloaded the disk image containing all of the sample code from the book, go to the folder for Chapter 16 and open the project for this recipe.

Building Notification Payloads

Delivering push notification through APNS requires three things: your SSL certificate, a device ID, and a custom payload with the notification you want to send. The payload uses JSON formatting. You've already read about generating the certificate and producing the device identifiers, which you need to pass up to your server. Building the JSON payloads basically involves transforming a small well-defined dictionary into JSON format.

JSON (JavaScript Object Notation) is a simple data interchange format based on key-value pairs. The JSON Web site (www.json.org) offers a full syntax breakdown of the format, which allows you to represent values that are strings, numbers, and arrays. The APNS

payload consists of up to 256 bytes, which must contain your complete notification information.

Notification payloads must include an `aps` dictionary. This dictionary defines the properties that produce the sound, badge, and/or alert sent to the user. In addition, you may add custom dictionaries with any data you need to send to your application so long as you stay within the 256 byte limit. Figure 16-11 shows the hierarchy for basic (nonlocalized) alerts.

```
aps
    badge : number
    sound : sound file name string
    alert : string
    alert
        body : string
        action-loc-key : string
```

Figure 16-11 The `aps` dictionary may contain one or more notification types including a badge request, a sound file, and/or an alert.

The `aps` dictionary contains one or more notification types. These include the standard types you've already read about: badges, sounds, and alerts. Badge and sound notifications each take one argument. The badge is set by a number, the sound by a string that refers to a file already inside the application bundle. If that file is not found (or the developer passes `default` as the argument), a default sound plays for any notification with a sound request. When a badge request is not included, the iPhone removes any existing badge from the application icon.

There are two ways to produce an alert. You can pass a string, which defines the message to show. This automatically produces a notification with two buttons under that message: Close and View. To customize buttons, pass a dictionary instead. Send the message text as the body and the string to use for the Action key (normally View) as `action-loc-key`. This replaces View with whatever text you specify.

To produce an alert with a single OK button, pass `null` as the argument to `action-loc-key`. This creates a special alert style with one button. Just as when a user taps Close, the OK style alert will not pass any data directly to your application. The app must poll for any updates when next opened by the user.

Localized Alerts

When working with localized applications, construct your `aps > alert` dictionary with two additional keys. Use `loc-key` to pass a key that is defined in your application's Localizable.strings file. The iPhone looks up the key and replaces it with the string found for the current localization.

At times, localization strings use arguments like `%@` and `%n$@`. Should that hold true for the localization you are using, you can pass those arguments as an array of strings via `loc-args`. As a rule, Apple recommends against using complicated localizations as they can consume a major portion of your 256-byte bandwidth.

Transforming from Dictionary to JSON

Once you've designed your dictionary, you must transform it to JSON. The JSON format is simple but precise. If you can, use an automated library to convert your dictionary to the JSON string. There are numerous solutions for this for any number of programming languages, including JavaScript, Perl, and so on. Here's a quick rundown of JSON basics. Table 16-1 offers examples of these rules in action.

Table 16-1 **JSON Payload Samples**

Sample Type	JSON
Hello message, displays with two buttons.	`{"aps":{"alert":"hello"}}`
Hello message, displays with two buttons, but built using JSON with an alert dictionary.	`{"aps":{"alert":{"body":"hello"}}}`
Hello message with one OK button.	`{"aps":{"alert":{"action-loc-key":null,"body":"hello"}}}`
Hello message with two buttons, Close and Open, the latter being a custom replacement for View.	`{"aps":{"alert":{"action-loc-key":"Open","body":"hello"}}}`
Hello message that adds an application badge of 3.	`{"aps":{"badge":3,"alert":{"body":"hello"}}}`
Play a sound without an alert.	`{"aps":{"sound":"ping2.caf","alert":{}}}`
Play sound, display badge, display alert, use a custom button.	`{"aps":{"sound":"ping2.caf","badge":2,"alert":{"action-loc-key":"Open","body":"Hello"}}}`
Add a custom payload including an array.	`{"aps":{"alert":{"body":"Hello"}},"key1":"value1","key2":["a","b","c"]}`

- The entire payload is a dictionary. Dictionaries consist of key-value pairs stored between brackets, that is, {key:value, key:value, key:value, ...}.
- Key-value pairs are separated with commas.
- Strings use double quotes; numbers do not. Reserved words include true, false, and null. Reserved words are not quoted.
- Arrays consist of a list of items between square brackets, that is, [item, item, item,...].

- The following symbols must be escaped in strings by using a backslash literal indicator: `' " \ /`.
- You may want to remove carriage returns (`\r`) and new lines (`\n`) from your payloads when sending messages.
- Spaces are optional. Save space by omitting them between items.
- The `aps` dictionary appears within the top-level folder, so the most basic payload looks something like `{aps:{}}`.

Custom Data

So long as your payload has room left, keeping in mind your tight byte budget, you can send additional information in the form of key-value pairs. As Table 16-1 showed, these custom items can include arrays and dictionaries as well as strings, numbers, and constants. You define how to use and interpret this additional information. The entire payload dictionary is sent to your application so whatever information you pass along will be available to the `application: didReceiveRemoteNotification:` method via the user dictionary.

A dictionary containing custom key-value pairs does *not* need to provide an alert, although doing so allows your user to choose to open your application if it isn't running. If your application is already launched, the key-value pairs arrive as a part of the payload dictionary.

Receiving Data on Launch

When your client receives a notification, tapping the action key (by default, View) launches your application. Then after launching, the iPhone sends your application delegate an optional callback. The delegate recovers its notification dictionary by implementing a method named `application:didFinishLaunchingWithOptions:`. Unfortunately, this method might not work properly. So here are both the standard ways of retrieving notification information plus a work-around.

Normally, the iPhone passes the notification dictionary to the delegate method via the launch options parameter. For remote notifications, this is the official callback to retrieve data from an alert-box launch. The `didReceiveRemoteNotification:` method is not called when the iPhone receives a notification and the application is not running.

This "finished launching" method is actually designed to handle two completely different circumstances. First, it handles these notification alert launches, allowing you to recover the payload dictionary and use the data that was sent. Second, it works with application launches from `openURL:`. If your app has published a URL scheme, and that scheme is used by another application, the application delegate handles that launch with this method.

In either case, the method must return a Boolean value. As a rule, return YES if you were able to process the request or NO if you were not. This value is actually ignored in the case of remote notification launches, but you must still return a value.

At the time of writing, implementing this method does not work properly. The application will hang without displaying a GUI. Fortunately, there's an easy work-around that

does not rely on the callback method. You can, instead, listen for a launch notification and catch the `userInfo` dictionary that is sent with it. This solution has the advantage of being reliable and tested. Keep an eye on Apple's developer forums (http://devforums.apple.com) to keep track of when this issue gets fixed.

Start by adding your application delegate as a listener via the default `NSNotificationCenter` in your normal `applicationDidFinishLaunching` method.

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

Then implement the method for the selector you provided. Here, the application waits for the GUI to finish loading and then displays the user info dictionary, where the remote notification data has been stored.

```
- (void) launchNotification: (NSNotification *) notification
{
    [self performSelector:@selector(showString) withObject:
    [[notification userInfo] description] afterDelay:1.0f];
}
```

Between the notification listener and the method callback, you can reliably grab the user data from remote notifications. This work-around should remain viable regardless of when and how Apple addresses the `didFinishLaunchingWithOptions` method.

> **Note**
>
> When your user taps Close and later opens your application, the notification is not sent on launch. You must check in with your server manually to retrieve any new user information. Applications are not guaranteed to receive alerts. In addition to tapping Close, the alert may simply get lost. Always design your application so that it doesn't rely solely on receiving push notifications to update itself and its data.

Recipe: Sending Notifications

The notification process involves several steps (see Figure 16-12). First, you build your JSON payload, which you just read about in the previous section. Next, you retrieve the SSL certificate and the device token for the unit you want to send to. How you store these is left up to you, but you must remember that these are sensitive pieces of information. Open a secure connection to the APNS server. Finally, you handshake with the server, send the notification package, and close the connection.

This is the most basic way of communicating and assumes you have just one payload to send. In fact, you can establish a session and send many packets at a time; however, that is left as an exercise for the reader as is creating services in languages other than Objective-C. The Apple Developer Forums (devforums.apple.com) host ongoing discussions about push providers and offer an excellent jumping off point for finding sample code for PHP, Perl, and other languages.

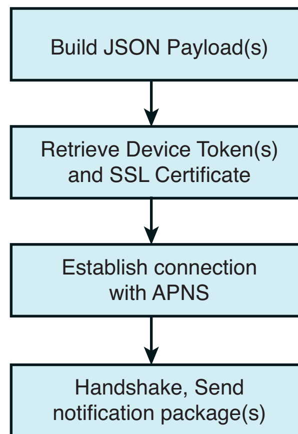

Figure 16-12 The steps for sending remote notifications.

Be aware that APNS may react badly to a rapid series of connections that are repeatedly established and torn down. If you have multiple notifications to send at once, go ahead and send them during a single session. Otherwise, APNS might confuse your push deliveries with a denial of service attack.

Recipe 16-2 demonstrates how to send a single payload to APNS, showing the steps needed to implement the fourth and final box in Figure 16-12. The recipe is built around code developed by Stefan Hafeneger and uses Apple's ioSock sample source code.

The individual server setups vary greatly depending on your security, databases, organization, and programming language. Recipe 16-2 demonstrates a minimum of what is required to implement this functionality and serves as a template for your own server implementation in whatever form this might take.

Sandbox and Production

Apple provides both sandbox (development) and production (distribution) environments for push notification. You must create separate SSL certificates for each. The sandbox helps you develop and test your application before submitting to App Store. It works with a smaller set of servers and is not meant for large-scale testing. The production system is reserved for deployed applications that have been accepted to App Store.

- The Sandbox servers are located at gateway.sandbox.push.apple.com, port 2195.
- The Production servers are located at gateway.push.apple.com, port 2195.

Recipe 16-2 **Pushing Payloads to the APNS Server**

```
// Adapted from code by Stefan Hafeneger
- (BOOL) push: (NSString *) payload
{
```

```
otSocket socket;
SSLContextRef context;
SecKeychainRef keychain;
SecIdentityRef identity;
SecCertificateRef certificate;
OSStatus result;

// Ensure device token
if (!self.deviceTokenID)
{
    printf("Error: Device Token is nil\n");
    return NO;
}

// Ensure certificate
if (!self.certificateData)
{
    printf("Error: Certificate Data is nil\n");
    return NO;
}

// Establish connection to server.
PeerSpec peer;
result = MakeServerConnection("gateway.sandbox.push.apple.com",
    2195, &socket, &peer);
if (result)
{
    printf("Error creating server connection\n");
    return NO;
}

// Create new SSL context.
result = SSLNewContext(false, &context);
if (result)
{
    printf("Error creating SSL context\n");
    return NO;
}

// Set callback functions for SSL context.
result = SSLSetIOFuncs(context, SocketRead, SocketWrite);
if (result)
{
    printf("Error setting SSL context callback functions\n");
    return NO;
}
```

```
// Set SSL context connection.
result = SSLSetConnection(context, socket);
if (result)
{
    printf("Error setting the SSL context connection\n");
    return NO;
}

// Set server domain name.
result = SSLSetPeerDomainName(context,
    "gateway.sandbox.push.apple.com", 30);
if (result)
{
    printf("Error setting the server domain name\n");
    return NO;
}

// Open keychain.
result = SecKeychainCopyDefault(&keychain);
if (result)
{
    printf("Error accessing keychain\n");
    return NO;
}

// Create certificate from data
CSSM_DATA data;
data.Data = (uint8 *)[self.certificateData bytes];
data.Length = [self.certificateData length];
result = SecCertificateCreateFromData(&data, CSSM_CERT_X_509v3,
    CSSM_CERT_ENCODING_BER, &certificate);
if (result)
{
    printf("Error creating certificate from data\n");
    return NO;
}

// Create identity.
result = SecIdentityCreateWithCertificate(keychain, certificate,
    &identity);
if (result)
{
    printf("Error creating identity from certificate\n");
    return NO;
}

// Set client certificate.
```

```
CFArrayRef certificates = CFArrayCreate(NULL,
    (const void **)&identity, 1, NULL);
result = SSLSetCertificate(context, certificates);
if (result)
{
    printf("Error setting the client certificate\n");
    return NO;
}

CFRelease(certificates);

// Perform SSL handshake.
do {result = SSLHandshake(context);}
    while(result == errSSLWouldBlock);


// Convert string into device token data.
NSMutableData *deviceToken = [NSMutableData data];
unsigned value;
NSScanner *scanner = [NSScanner
    scannerWithString:self.deviceTokenID];
while(![scanner isAtEnd]) {
    [scanner scanHexInt:&value];
    value = htonl(value);
    [deviceToken appendBytes:&value length:sizeof(value)];
}

// Create C input variables.
char *deviceTokenBinary = (char *)[deviceToken bytes];
char *payloadBinary = (char *)[payload UTF8String];
size_t payloadLength = strlen(payloadBinary);

// Prepare message
uint8_t command = 0;
char message[293];
char *pointer = message;
uint16_t networkTokenLength = htons(32);
uint16_t networkPayloadLength = htons(payloadLength);

// Compose message.
memcpy(pointer, &command, sizeof(uint8_t));
pointer += sizeof(uint8_t);
memcpy(pointer, &networkTokenLength, sizeof(uint16_t));
pointer += sizeof(uint16_t);
memcpy(pointer, deviceTokenBinary, 32);
pointer += 32;
memcpy(pointer, &networkPayloadLength, sizeof(uint16_t));
```

```
    pointer += sizeof(uint16_t);
    memcpy(pointer, payloadBinary, payloadLength);
    pointer += payloadLength;

    // Send message over SSL.
    size_t processed = 0;
    result = SSLWrite(context, &message, (pointer - message),
        &processed);
    if (result)
    {
        printf("Error sending message via SSL.\n");
        return NO;
    }
    else
    {
        printf("Message sent.\n");
        return YES;
    }
}
```

Get This Recipe's Code

To get the code used for this recipe, go to http://github.com/erica/iphone-3.0-cookbook-, or if you've downloaded the disk image containing all of the sample code from the book, go to the folder for Chapter 16 and open the project for this recipe.

Recipe: Push in Action

Once you set up a client such as the one discussed in Recipe 16-1 and routines like Recipe 16-2 that let you send notifications, it's time to think about deploying an actual service. Recipe 16-3 introduces a Twitter client that repeatedly scans a search.twitter.com RSS feed and pushes notifications whenever a new tweet is found (see Figure 16-13).

This code is built around the push routine from Recipe 16-2 and the XML parser from Recipe 13-13. This utility pulls down Twitter search data as an XML tree and finds the first tree node of the type "entry," which is how Twitter stores each tweet.

Next, it creates a string by combining the poster name (from the "name" leaf) and the post contents (from the "title" leaf). It then adds a JSON-escaped version of this string to the `aps` > `alert` dictionary as the message body. The alert sound and one-button style are fixed in the main `aps` payload dictionary.

The application runs in a loop with a time delay set by a command-line argument. Every n seconds (determined by the second command-line argument), it polls, parses, and checks for a new tweet, and if it finds one, pushes it out through APNS. Figure 16-13 shows this utility in action, displaying a tweet alert on the client iPhone.

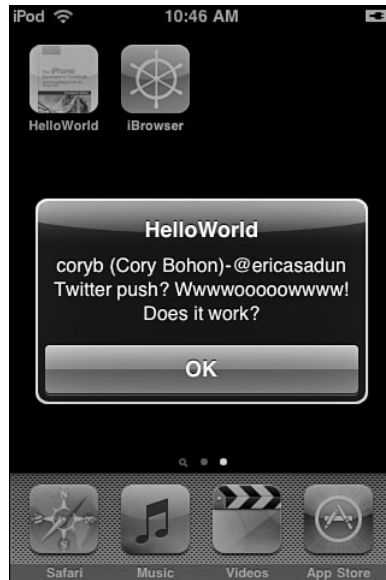

Figure 16-13 Twitter provides an ideal way to test a polled RSS feed.

Recipe 16-3 **Wrapping Remote Notifications into a Simple Twitter Utility**

```
#define TWEET_FILE      [NSHomeDirectory()\
    stringByAppendingPathComponent:@".tweet"]
#define URL_STRING \
    @"http://search.twitter.com/search.atom?q=+ericasadun"
#define SHOW_TICK    NO
#define CAL_FORMAT    @%Y-%m-%dT%H:%M:%SZ"

int main (int argc, const char * argv[]) {

    if (argc < 2)
    {
        printf("Usage: %s delay-in-seconds\n", argv[0]);
        exit(-1);
    }

    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

    // Fetch certificate and device information from the current
    // directory as set up with pushutil
    char wd[256];
    getwd(wd);
```

```
NSString *cwd = [NSString stringWithCString:wd];
NSArray *contents = [[NSFileManager defaultManager]
    directoryContentsAtPath:cwd];

NSArray *dfiles = [contents pathsMatchingExtensions:
    [NSArray arrayWithObject:@"devices"]];
if (![dfiles count])
{
    printf("Error retrieving device token\n");
    exit(-1);
}
NSDictionary *dict = [NSDictionary dictionaryWithContentsOfFile:
    [cwd stringByAppendingPathComponent:[dfiles lastObject]]];
if (!dict || ([[dict allKeys] count] < 1))
{
    printf("Error retrieving device token\n");
    exit(-1);
}
[APNSHelper sharedInstance].deviceTokenID = [dict objectForKey:
    [[dict allKeys] objectAtIndex:0]];

NSArray *certs = [contents pathsMatchingExtensions:
    [NSArray arrayWithObject:@"cer"]];
if ([certs count] < 1)
{
    printf("Error finding SSL certificate\n");
    exit(-1);
}
NSString *certPath = [certs lastObject];
NSData *dCert = [NSData dataWithContentsOfFile:certPath];
if (!dCert)
{
    printf("Error retrieving SSL certificate\n");
    exit(-1);
}
[APNSHelper sharedInstance].certificateData = dCert;

// Set up delay
int delay = atoi(argv[1]);
printf("Initializing with delay of %d\n", delay);

// Set up dictionaries
NSMutableDictionary *mainDict = [NSMutableDictionary dictionary];
NSMutableDictionary *payloadDict =
    [NSMutableDictionary dictionary];
NSMutableDictionary *alertDict = [NSMutableDictionary dictionary];
```

```
[mainDict setObject:payloadDict forKey:@"aps"];
[payloadDict setObject:alertDict forKey:@"alert"];
[payloadDict setObject:@"ping1.caf" forKey:@"sound"];
[alertDict setObject:[NSNull null] forKey:@"action-loc-key"];

while (1 > 0)
{

    NSAutoreleasePool *wadingpool =
        [[NSAutoreleasePool alloc] init];
    TreeNode *root = [[XMLParser sharedInstance] parseXMLFromURL:
        [NSURL URLWithString:URL_STRING]];
    TreeNode *found = [root objectForKey:@"entry"];

    if (found)
    {
        // Recover the string to tweet
        NSString *tweetString = [NSString stringWithFormat:
            @"%@-%@", [found leafForKey:@"name"],
            [found leafForKey:@"title"]];

        // Recover pubbed date
        NSString *dateString = [found leafForKey:@"published"];
        NSCalendarDate *date = [NSCalendarDate dateWithString:
            dateString calendarFormat:CAL_FORMAT];

        // Recover stored date
        NSString *prevDateString = [NSString
            stringWithContentsOfFile: TWEET_FILE
            encoding:NSUTF8StringEncoding error:nil];
        NSCalendarDate *pDate = [NSCalendarDate dateWithString:
            prevDateString calendarFormat:CAL_FORMAT];

        // Tweet only if there is either no stored date or
        // the dates are not equal
        if (!pDate || ![pDate isEqualToDate:date])
        {
            // Update with the new tweet information
            NSLog(@"\nNew tweet from %\n   \"%@\"\n\n",
               [found leafForKey:@"name"],
               [found leafForKey:@"title"]);

            // Store the tweet time
            [dateString writeToFile:TWEET_FILE atomically:YES
                encoding:NSUTF8StringEncoding error:nil];
```

```
            // push it
            [alertDict setObject:jsonescape(tweetString)
                forKey:@"body"];
            [[APNSHelper sharedInstance] push: [JSONHelper
                jsonWithDict:mainDict]];
        }
    }

    root = nil;
    found = nil;

    [wadingpool drain];

    [NSThread sleepForTimeInterval:(double) delay];
    if (SHOW_TICK) printf("tick\n");
}

[pool drain];
return 0;
}
```

Get This Recipe's Code

To get the code used for this recipe, go to http://github.com/erica/iphone-3.0-cookbook-, or if you've downloaded the disk image containing all of the sample code from the book, go to the folder for Chapter 16 and open the project for this recipe.

Feedback Service

Apps don't live forever. Users add, remove, and replace applications on their iPhones all the time. From an APNS point of view, it's pointless to deliver notifications to iPhones that no longer host your application. As a push provider, it's your duty to remove inactive device tokens from your active support list. As Apple puts it, "APNS monitors providers for their diligence in checking the feedback service and refraining from sending push notifications to nonexistent applications on devices." Big Brother *is* watching.

Apple provides a simple way to manage inactive device tokens. When users uninstall apps from a device, push notifications begin to fail. Apple tracks these failures and provides reports from its APNS feedback server. The APNS feedback service lists devices that failed to receive notifications. As a provider, you need to fetch this report on a periodic basis and weed through your device tokens.

The feedback server hosts sandbox and production addresses, just like the notification server. You find these at feedback.push.apple.com (port 2196) and feedback.sandbox.push.apple.com. You contact the server with a production SSL certificate and shake hands in the same way you do to send notifications. After the handshake, read your results. The server sends data immediately without any further explicit commands on your side.

The feedback data consists of 38 bytes. This includes the time (4 bytes), the token length (2 bytes), and the token itself (32 bytes). The timestamp tells you when APNS first

determined that the application no longer existed on the device. This uses a standard UNIX epoch, namely seconds since Midnight, January 1st, 1970. The device token is stored in binary format. You need to convert it to a hex representation to match it to your device tokens if you use strings to store token data. At the time of writing this book, you can ignore the length bytes. They are always 0 and 32, referring to the 32-byte length of the device token.

```
// Retrieve message from SSL.
size_t processed = 0;
char buffer[38];
do
{
    // Fetch the next item
    result = SSLRead(context, buffer, 38, &processed);
    if (result) break;

    // Recover Date from data
    char *b = buffer;
    NSTimeInterval ti = ((unsigned char)b[0] << 24) +
       ((unsigned char)b[1] << 16) +
       ((unsigned char)b[2] << 8) +
       (unsigned char)b[3];
    NSDate *date = [NSDate dateWithTimeIntervalSince1970:ti];

    // Recover Device ID
    NSMutableString *deviceID = [NSMutableString string];
    b += 6;
    for (int i = 0; i < 32; i++) [
        deviceID appendFormat:@"%02x", (unsigned char)b[i]];

    // Add dictionary to results
    [results addObject:
        [NSDictionary dictionaryWithObject:date
        forKey:deviceID]];

} while (processed > 0);
```

> **Note**
> Search your Xcode Organizer Console for “aps” to locate APNS error messages.

Designing for Push

When designing for push, keep scaling in mind. Normal computing doesn’t need to scale. When coding is done, an app runs on a device using the local CPU. Should a developer deploy an extra 10,000 copies, there’s no further investment involved other than increased technical support.

Push computing does scale. Whether you have 10,000 or 100,000 or 1,000,000 users matters. That's because developers must provide the service layer that handles the operations for every unit sold. The more users supported, the greater the costs will be. Consider that these services need to be completely reliable and that consumers will not be tolerant of extended downtimes.

Consider an application with just 10,000 users. It might service a million uses per day, assuming update checks every 15 minutes. More time-critical uses might demand checks every few minutes or even several times a minute. As the computational burden builds, so do the hosting costs. While cloud computing provides an excellent match to these kinds of needs, that kind of solution comes with a real price in development, maintenance, and day-to-day operations.

On top of reliability, add in security concerns. Many polled services require secure credentials. Those credentials must be uploaded to the service for remote use rather than being stored solely on the device. Even if the service in question does not use that kind of authentication, the device token that allows your service to contact a specific phone is sensitive in itself. Should that identifier be stolen, it could let spammers send unsolicited alerts. Any developer who enters this arena must take these possible threats seriously and provide highly secure solutions for storing and protecting information.

These concerns, when taken together, point to the fact that push notifications are serious business. Some small development houses may completely opt out of being push providers for apps that depend on new information notifications. Between infrastructure and security concerns, the work it will take to properly offer this kind of service may price itself out of reach for those developers. Third party providers like Key Lime Tie (keylimetie.com) and Urban Airship (urbanairship.com) offer ready-to-use Push infrastructure with affordable pricing plans. They handle the remote notification deployment for you.

On the other hand, many developers may employ push for occasional opt-in notifications, such as alerting users that upgrades are now available in the App Store or to send tips about using the product. How tolerant iPhone users will be of this kind of use remains to be seen.

Summary

In this chapter, you saw push notifications both from a client-building point of view and as a provider. You learned about the kinds of notifications you can send and how to create the payload that moves those notifications to the device. You discovered registering and unregistering devices and how users can opt in and out from the service. You saw how to create a provider utility that pushes new Twitter items.

Much of the push story lies outside this chapter. It's up to you to set up a server and deal with security, bandwidth, and scaling issues. The reality of deployment is that there are many platforms and languages that can be used that go beyond the Objective-C sample code shown here. Regardless, the concepts discussed and recipes shown in this

chapter give you a good stepping off point. You know what the issues are and how things have to work. Now it's up to you to put them to good use.

- The big wins of notifications are their instant updates and immediate presentation. Like SMS messages, they're hard to overlook when they arrive on your iPhone. There's nothing wrong in opting out of push if your application does not demand that kind of immediacy.
- Guard your SSL certificate and device tokens. Although it's too early to say how Apple will respond to security breaches, experience suggests that it will be messy and unpleasant.
- Don't leave users without service when you have promised to provide it to them. Build a timeline into your business plan that anticipates what it will take to keep delivering notifications over time and how you will fund this. Consumers will not be tolerant of extended downtimes; your service must be completely reliable.
- Build to scale. Although your application may not initially have tens of thousands of users, you must anticipate a successful app launch as well as a modest one. Create a system that can grow along with your user base.

Symbols

+ (plus), class methods, 101
- (dash), method declarations, 98
2.x support, adding to image selection, 263
3.1 support, adding to image selection, 263
@ (at) symbol, 92, 103

A

ABAddressBookCopyArrayOfAllPeople() function, 724
ABAdressBookCreate() function, 724
Abbott, Jay, 86
ABContact class, 738
ABContactsHelper class, 738
ABGroup class, 738
ABGroupAddMember() function, 737
ABGroupCreate() function, 736
ABGroupRemoveMember() function, 737
ABPeoplePickerNavigationController class, 742, 744
ABPeoplePickerNavigationControllerDelegate protocol, 743
ABPersonHasImageData() function, 733
ABRecordCopyValue() function, 728
ABRecordRef type, 724-725
ABRecordSetValue() function, 726, 730
ABUnknownPersonViewController, 750-752
accelerometer
  detecting shakes, 605-608
  locating "up," 597-599
  moving onscreen objects, 599-601
AccelerometerHelper class, 605
access points (Wi-Fi), 690
accessibility (VoiceOver)
  adding from code, 802-803
  adding with Interface Builder, 799-802
  common VoiceOver gestures, 805-806
  overview, 799
  testing on iPhone, 803-804, 806
  testing with simulator, 803
accessing
  Address Book image data, 741-742
  arrays, 133
  camera, 148
  device information, 589-590, 592-593
  FTP sites, 586-587
  SDKAPIs from Xcode, 50-51

sets, 135
substrings, 128-129
accessor methods, 105
accessory views
check marks in table cells, 446-448
disclosure accessories in table cells, 449-451
accounts, test accounts (StoreKit)
creating, 781-782
signing into, 790
action sheets
displaying text in, 405-406
menus, creating, 403-405
pop-ups versus, 403
actions
adding, 162
connecting buttons to, 347
ad hoc distributions, 83
applications, building, 84
artwork, adding, 84-85
devices, registering, 83
entitlement files, 83
mobile provisions, building, 83
Address Book
ABContact class, 738
ABContactsHelper class, 738
ABGroup class, 738
ABRecordRef type, 724-725
ABUnknownPersonViewController, 750-752
AddressBookUI framework, 724
contacts
adding, 747-748
adding random contact art, 752-754
limiting contact picker properties, 745-747
modifying, 748-750
picking people, 742-745
searching for, 735
groups, 736-738
images, 733-734, 741-742
overview, 723
properties
address and instant message properties, 730-733
date properties, 726-730
multivalue record properties, 727-730
records
adding, 734
creating, 734
deleting, 735-736
multivalue record properties, 727-730
retrieving and setting ABRecord strings, 725-726
referencing, 724
searching, 738-740
address book controllers, 150
address properties (Address Book), 730-733
AddressBookUI framework, 724
addSubview method, 29
affine transform of UIView, 233-234
alert sounds, creating, 418
alerts, 391, 673. *See also* progress indicators
application badges, updating, 416-417
audio alerts, 417-420
classes for, 394
creating, 391-392
delegate methods in, 392-394
displaying, 190-191, 394
localized alerts, 673
modal alerts, creating with run loops, 396-399
network activity indicators, 415-416
no-button alerts, creating, 394-396
orientable scroll-down alerts, creating, 412-415
requesting text input via, 399-402
tappable overlays, creating, 411-412
variadic arguments with, 402-403
volume alert, displaying, 420-421
allApplicationSubviews() function, 214
allocating memory, 94
allSubviews() function, 214
alternating table cell colors, 439-441
animations. *See also* transitions
in buttons, 351-354
view animations
bouncing views, 248-250
building UIView animation blocks, 236-237
callbacks, 237
Core Animation calls, 244-246
Core Animation transitions, 242-244

curl transitions, 246-247
fading views in and out, 237-238
flipping views, 240-241
image view animations, 250-251
overview, 236
swapping views, 239-240

annotations
map annotations
adding, 710
annotation views, 710-712
creating, 710
MapAnnotation class, 709-710
responding to annotation button taps, 712-716
user location annotations, 707-708

APNS (Apple Push Notification Service), 656

App Store
compiling clean builds for, 80-81
debugging uploads, 81-82

appending strings, 126

Apple Push Notification Service (APNS), 656

application approval for in-app purchase items (StoreKit), 785-786

application badges, updating, 416-417

application bundles, 257
components, 26-27
application folder hierarchy, 22-23
executable, 23
icon and default images, 25-26
Info.plist files, 23-25
NIB files, 26
loading images from, 258

application delegate, 20-21

application identifiers
editing, 66-67
generating for push notifications, 659
registering, 15

application limits, platform differences, 11

application registration for push notifications, 662-663
error handling, 664-665
responding to notifications, 665-666
retrieving device tokens, 663-664

applications. *See also* projects
IPA archives, 27
overview, 17-18
sandboxes, 27
sharing keychains between, 575-577
skeleton, 18-22
submitting for review, 787

applying image processing, 293-295

aps dictionary, 673-675

archiving, persistence through, 314-315

arguments, variadic arguments with alerts, 402-403

arrays, 133
accessing, 133
converting strings to, 128
converting to strings, 134
creating, 133
NSArray class, 97
NSMutableArray class, 97
for table sections, creating, 468-469
testing, 134
view controller arrays, loading, 198

artwork, adding to ad hoc distributions, 84-85

assembling applications
application skeleton, 18-19
application delegate, 20-21
main.m file, 19-20
view controller, 21-22
overview, 17-18

assigning
data sources for tables, 425-426
delegates for tables, 426
properties, 109
retained, 113-114
self-assigning, 112

asynchronous downloads, 560-565

at (@) symbol, 92, 103

atomic methods, 109

attributes
Core Data, 758
of properties, 109-110

audio
handling interruptions, 621-622
ignoring lock events, 622-624
iPod library contents, filtering, 645-649
looping, 618-620
picking, 641-645
playing, 611, 615-618
catching end of playback, 614
initializing audio players, 611-612
monitoring audio levels, 613

with MPMusicPlayerController class, 649-653
scrubbing audio, 614
recording, 624-628
with Audio Queues, 629-634
audio alerts, 417
alert sounds, creating, 418
system sounds, creating, 417-420
vibration, creating, 418
audio players, initializing, 611-612
Audio Queues, 417, 629-634
Audio Services, 418-420
authentication challenges, handling, 565-566
autorelease memory management, 58
autorelease objects
creating, 111-112
explained, 110-111
lifetime of, 112
retaining, 112-113
autorelease pools, 19-20
autosizing, 176-179
text editors, 372
availability, checking, 555-557. *See also* reachability
availability date, setting, 781
AVAudioPlayer class, 417, 611. *See also* audio
catching end of playback, 614
initializing audio players, 611-612
monitoring audio levels, 613
playing audio, 615-618
scrubbing audio, 614
AVAudioRecorder class, 624
AVAudioSession class, 624

B

background color of tables, changing, 430-432
background images for tables, creating, 432-433
badges, 673
application badges, updating, 416-417
Ballard, Kevin, 388
bar button items, 347
bars, 146-147
battery state, monitoring, 594-595
behavior limits, platform differences, 12
bitmaps, 291
applying image processing, 293-295
drawing into bitmap contexts, 291-293
image-processing limitations, 295-297
testing touches against bitmap alpha levels, 309-311
Bluetooth, GameKit and, 495
limitations, 496-497
Bonjour, 495
GameKit sessions and, 496, 498
iPhone servers
creating, 515-520
Mac clients, creating, 520-523
names and ports, registering, 528-529
scanning for services, 540-543
BonjourHelper class, 528-537
bookmarks, 76-77
bouncing views, 248-250
bounded movement, 306-307
bounded views, moving randomly, 231-232
breakpoints, 53-55
Britten, Ben, 629
browsing parse trees, 580-582
buffers, NSData class, 136
built-in controls in table cells, 441-443
Bundle Seed IDs, 16
buttons, 344-345
adding in Interface Builder, 345-347
animation in, 351-354
connecting to actions, 347
custom buttons
building in Xcode, 348-351
creating, 346-347
multiline button text, 351
in segmented controls, 362-363
toggle buttons, 354-356

C

C programming language, 91, 116
C strings, converting to/from, 127
caching, 768
memory management, 59
monitoring with Instruments application, 62-64
calculating lines, 323-325

callbacks
adding to protocols, 123
animation callbacks, 237
optional callbacks, 123-124
camera. *See also* images
accessing, 148
capturing time-lapse photos, 273-275
custom camera overlays, 275, 277-278
model differences, 7
selecting and customizing images from camera roll, 265-267
snapping photos and writing to photo album, 268-270
camera roll, selecting and customizing images from, 265-268
canceling peer picker alerts, 499
capability requirements, adding, 590-592
capturing
colors, 165
time-lapse photos, 273-275
Carbon, explained, 117
case (of strings), changing, 129
catching end of audio playback, 614
categories, explained, 120-121
cell tower positioning, 690
cells (table)
adding, 453
alternating colors, 439-441
building custom, 435-439
with built-in controls, 441-443
check marks in, 446-448
deleting, 451-456
disclosure accessories in, 449-451
removing selection highlights, 448-449
reordering, 456-457
retaining state, 443-445
returning from sections, 470
reusing, 425, 428
selection color, setting, 429
swiping, 453
types of, 433-435
visualizing reuse, 445-446
centering landscape views, 234-235
certificates, 14
CFShow function, 105
CGFont class, 388
CGRect structure, 223-224, 227
CGRectCreateDictionaryRepresentation() function, 223
CGRectFromString() function, 223, 313
CGRectGetCenter() function, 227
CGRectInset() function, 223
CGRectIntersectsRect() function, 223
CGRectMake function, 223
CGRectMoveToCenter() function, 227
CGRectZero() function, 223
chat, 512-515
check marks in table cells, 446-448
chevrons, 449-451
child-view undo support, 316
choices, views for, 145
chunked data for asynchronous downloads, 562
circles, detecting, 325-327
circular hit test, 308
Clang static analyzer, 64-65, 98
class headers, inspecting, 163
class methods, 101
classes. *See also* Foundation classes
for alerts, 394
explained, 92-93
extending, 120-121
hierarchy, 102-103
implementing, 100
logging information, 103-105
naming in Cocoa Touch, 92
for progress indicators, 406-407
clean builds, 80-81
clearing console log, 56-57
CLHeading class, 698
client mode (peer pickers), 500, 502
client skeleton example (push notifications), 667-672
clients
in GameKit, 498
Mac clients, creating for iPhone Bonjour servers, 520-523
clipboard for simulator, 48, 524-525
CLLocation class, properties, 694-695
closing connections with BonjourHelper class, 530
Cocoa, explained, 117. *See also* Foundation classes
Cocoa Touch
class names, 92
definition of, 4-5
explained, 117

CocoaDev Web site, 816
code, adding VoiceOver accessibility from, 802-803
code signing identity, setting, 67-68
code-based temperature converter example, 166-169
collapsing methods, 77
collections, 133-136
arrays, 133–134
dictionaries, 134–135
fast enumeration, 101
memory management, 135
sets, 135
writing to files, 135–136
color
background color of tables, changing, 430–432
capturing, 165
selection color for table cells, setting, 429
of table cells, alternating, 439–441
com.yourcompany, overriding, 86
comparing dates, 131
compiler directives
explained, 73–74
iPhone-specific definitions, recovering, 74–75
pragma marks, 76–77
runtime checks, 75–76
compiler warnings
message forwarding, 140
treating as errors, 98
compiling
applications, 68–69
clean builds for App Store, 80–81
complex data, sending via GameKit, 510-512
compound predicates in fetch requests (Core Data), 771
computing speed and distance, 696-697
configurations, creating/editing distribution configurations, 78-79
conforming to protocols, 124-125
connecting buttons to actions, 347
connection process, GameKit peers, 498-500, 502
connections
adding, 163–164
asynchronous downloads, 560–565
authentication challenges, handling, 565–566
closing with BonjourHelper class, 530
connectivity changes, scanning for, 549–552
data uploads, 572–575
FTP access, 586–587
GameKitHelper class, 503–504
IP and host information, retrieving, 552–555
network activity indicators, 415–416
network status, checking, 545–547
online connections, creating with GameKit, 537–540
peer-to-peer connections. *See* Bonjour; GameKit
POST requests, uploading via, 569–572
site availability, checking, 555–557
synchronous downloads, 557–560
UIDevice class, extending for reachability, 547–549
WiFi connections with BonjourHelper class, 528–537
connectivity changes, scanning for, 549-552
console
clearing log, 56–57
running, 55–56
Console tab (Organizer), 72
constraining movement, 305, 307
consumable purchases, 784
Contact Add button, 344
contacts (Address Book)
ABUnknownPersonViewController, 750–752
adding, 747–748
adding random contact art, 752–754
limiting contact picker properties, 745–747
modifying, 748–750
picking people, 742–745
searching for, 735
content length for asynchronous downloads, 562
contents controllers, 149
contexts (Core Data)
creating, 760–761
inserting entities into, 761–763

controller behavior
delegation, 30-31
notifications, 33
overview, 30
target-actions, 32
controls, 145-146, 341
buttons, 344-345
adding in Interface Builder, 345-347
animation in, 351-354
building in Xcode, 348-351
connecting to actions, 347
creating custom buttons, 346-347
multiline button text, 351
events, 341-344
page indicators, 376-383
remove controls, displaying/dismissing, 452
segmented controls, 362-363
sending events, 364
sliders, custom slider thumbs, 356-361
subclassing UIControl class, 363-366
switches, 354-356
in table cells, 441-443
text fields
dismissing keyboards, 366-369
text entry filtering, 374-376
text views
creating text editors, 371-374
dismissing keyboards, 370-371
smart labels, 387-388
toolbars
creating in Interface Builder, 384-385
creating in Xcode, 385-386
tips for, 387
types of, 341
convenience methods, 111
conversion method, defining, 165-166
converting
aps dictionary to JSON, 674-675
arrays to strings, 134
C strings, 127
Interface Builder files to Objective-C, 51-53
strings to arrays, 128
XML data to tree data structures, 577-582
coordinate systems, 224
Core Animation
calls, 244-246
transitions, 242-244
Core Data
contexts, creating, 760-761
explained, 757-758
header files, generating, 759-760
model files, creating/editing, 758
objects
creating, 761-763
removing, 765-767
retrieving, 763-764
search tables example, 770-772
table data sources example, 767-770
table editing example, 773-775
table undo/redo support example, 775-778
Core Foundation
explained, 117
memory management, 116-117
Core Graphics, masking reflections with, 253-255
Core Location
cell tower positioning, 690
computing speed and distance, 696-697
detecting direction of north, 698-700
GPS positioning, 690
hybrid positioning approaches, 691
Internet provider positioning, 691
model differences, 8
overview, 689
SkyHook Wi-Fi positioning, 690
tracking latitude and longitude
code listing, 693
location properties, 694-695
step-by-step process, 691-692
tracking speed, 695-696
counting table sections/rows, 469-470
Cox, Brad J., 91
Crash Logs tab (Organizer), 72-73
credentials, 566-569
cross-promotion, 815
curl transitions, 246-247
custom buttons. *See also* buttons
building in Xcode, 348-351
creating, 346-347
custom camera overlays, 275, 277-278
custom getters/setters, creating, 107-109

custom key-value pairs in notification payloads, 675
custom modal controllers example, 199-201
custom overlays for progress indicators, creating, 409-411
custom popping options example (navigation controllers), 197-199
custom settings bundles, adding, 806-807
avoiding sensitive information, 808
checking user defaults, 813-814
creating custom settings page, 810-813
defining settings bundle, 809
Llama Settings project, 813
Settings app, 807
settings schema, 808
custom slider thumbs, 356-361
custom table cells, building, 435-439
custom templates, creating, 86-88
custom undo routine, 318-319
customized paged scroller example, 379-383
customizing
images from camera roll, 265-268
selected table cells, 439
table headers/footers, 474-476
toolbars, 56-57
Xcode identities, 85-86
cylinder roll example (picker views), 484-487

D

dash (-), method declarations, 98
data access limits, platform differences, 10
data display, views for, 144
data handling, GameKitHelper class, 504-505
data length, checking in GameKit, 523-527
data retrieval via pasteboards, 525
data sharing via pasteboards, 524
data source methods, building searchable, 465-466
data sources, 34-35
explained, 122
for tables
assigning, 425-426
methods, 427-428
data storage via pasteboards, 524
data structures for table sections, creating, 468-469
data uploads, 572-575
date properties (Address Book), 726-730
date/time
entering in tables, 487-490
formatting, 490-493
NSDate class, 131-132
NSDateFormatter class, 132
deallocating objects, 117-119
example, 119
retained properties, 118
variables, 118
debugger, 53
breakpoints, 53-55
console, 55-57
customizing toolbars, 56-57
objects, inspecting, 55
opening, 53
running, 53
zombies, enabling, 57
debugging
App Store uploads, 81-82
tethered debugging, overview, 6-7
declaring
interfaces, 92
methods, 98-99
optional callbacks, 123-124
URL, 815-816
default settings, checking user defaults, 813-814
Default.png files, 25-26
defining
conversion method, 165-166
protocols, 122-123
settings bundle, 809
delays
in registering purchases (StoreKit), 794
in system sounds, 419
delegate methods
alerts, 392-394
assigning for tables, 426
table searches, 467
table sections, 472
delegation, 30-31, 122
delete rules (Core Data), 766
deleting. ***See*** **removing**
deployment
application identifiers, editing, 66-67
code signing identity, setting, 67-68
compiling applications, 68

development provisions, installing, 66
signing applications, 68-69
deselecting table cells, 448
deserializing property lists, 510-512
Detail Disclosure button, 344
Detail pane (Xcode projects), 41-42
detecting
circles, 325-327
device orientation, 601-603
direction of north, 698-700
leaks with Instruments application, 59-60, 62
multitouch, 327-329
shakes
with accelerometer, 605-608
with motion events, 603-604
developer portal
overview, 13
provisioning, 16
registering application identifiers, 15
registering devices, 14-15
requesting certificates, 14
setting up teams, 13-14
developer programs. *See also* developer portal
Enterprise Developer Program, 2
Online Developer Program, 2
registering for, 3
Standard Developer Program, 2
table of, 1-2
University Developer Program, 3
development devices, 5
development process for push notifications, 659
application identifier, generating, 659
push-specific provisions, 661-662
SSL certificate, generating, 660-661
development provisions, installing, 66
device capability requirements, adding, 590-592
device information, accessing, 589-590, 592-593
device orientation, detecting, 601-603
device tokens
managing inactive, 685-686
retrieving, 663-664
devices, registering, 14-15, 83
Devices list (Organizer), 71
dictionaries, 133-135
creating, 134
listing keys, 135
removing objects, 135
replacing objects, 134
searching, 134
direct manipulation interfaces. *See also* touches
calculating lines, 323-325
detecting circles, 325-327
gesture distinction, 329-333
interactive resize and rotation, 333-338
multitouch, 303-304, 327-329
persistence, 311-315
simple direct manipulation interface, 304-305
touch-based painting, 321-323
undo support, 316-320
direction of north, detecting, 698-700
direction sensing
locating "up," 597-599
moving onscreen objects, 599-601
directives. *See* compiler directives
disabling proximity sensor, 596-597
disclosure accessories in table cells, 449-451
disconnections
BonjourHelper class, 530
GameKitHelper class, 503-504
disk space, checking, 608-609
dismissing
keyboards, 366-371
remove controls, 452
displaying
alerts, 190-191, 394
data, views for, 144
images in scrollable view, 278-280
multiimage paged scroll, 280-281
remove controls, 452
text in action sheets, 405-406
volume alert, 420-421
peer picker, 498-499
distance, computing, 696-697
distribution configurations, creating/editing, 78-79. *See also* ad hoc distributions
Documents folder, saving images to, 270-271

dot notation, 105
double-taps, 330
downloads
asynchronous downloads, 560-565
iPhone SDK, 3
synchronous downloads, 557-560
draggable views, creating, 304-305
drags, 330
drawing
into bitmap contexts, 291-293
touch-based painting, 321-323
duplex connections with BonjourHelper class, 530
dynamic typing, explained, 96-98

E

e-mailing images, 272-273
editing
Address Book contacts, 748-750
application identifiers, 66-67
distribution configurations, 78-79
main.m (hybrid temperature converter example), 172-173
model files (Core Data), 758
navigation bar, 159
simulator library, 48
tables in Core Data, 773-775
video, 639-641
view controller implementation, 171-172
views, 44-45
editor windows (Xcode projects), 42
efficiency of custom slider thumbs, 358
embedding images onto scrollers, 278-280
enabling
accessibility, 802
interactions, 160
proximity sensor, 596-597
reorientation, 175-176
simulated elements, 160
zombies, 57
energy limits, platform differences, 11
Enterprise Developer Program, 2
entities (Core Data), 758
header files, generating, 759-760
inserting into contexts, 761-763
entitlement files, 83
epochs, 131
error handling for device token requests, 664-665
errors, treating warnings as, 98
evaluating autosize option, 178-179
events
control events, 341-344
motion events, detecting shakes, 603-604
sending from controls, 364
executable, 23
extending
classes, 120-121
UIDevice class for reachability, 547-549
extracting
numbers from strings, 130
view hierarchy tree, 213

F

fading views, 237-238
fast enumeration of collections, 101
feedback service for push notifications, 685-686
fetch requests (Core Data), 763-764
fetch results (Core Data)
search tables example, 770-772
table data sources example, 767-770
file extensions, 19
file management, 136-138
file system size, checking, 608-609
File Transfer Protocol (FTP), accessing sites, 586-587
files
executable, 23
file types, 19
Info.plist, 23-25
IPA archives, 27
NIB files, 26
writing collections to, 135-136
writing/reading strings, 127-128
filtering
iPod library contents, 645-649
text entries, 374-376
finding
Address Book contacts, 735
best location match, 704-707

fixpng utility, 10-11
flipping views, 240-241
FlipView interface, 241
font table example, 428-430
FontLabel, 388
footers for tables, customizing, 474-476
form data uploads, 572-575
format specifiers for strings, 104
formatting date/time, 490-493
Foundation, explained, 117
Foundation classes, 125-126
collections, 133-136
dates, 131-132
file management, 136-138
index paths, 132
NSData, 136
numbers, 131
strings, 126-130
timers, 132
URLs, building, 136
frames. *See* views
free space, checking, 608-609
freeform group tables, 473, 477-480
FTP (File Transfer Protocol), accessing sites, 586-587
FTPHelper class, 586-587

G

GameKit, 495
Bluetooth and, 495-497
clients, 498
complex data, sending, 510-512
limitations, overcoming, 523-527
online connections, creating, 537-540
peers, 498
connection process, 498-500, 502
state changes, 503
sending/receiving data, 502
servers, 498
sessions, 496, 498
status logs, monitoring, 509-510
Voice Chat, 512-515
GameKitHelper class, creating, 503, 505-509
connections/disconnections, 503-504
data handling, 504-505
gaming with BonjourHelper class, 528-537
Garbage Collection, 12
geocoding, 717-720
reverse geocoding, 700-702
geometry
interface design, 151
keyboards, 154-155
navigation bars/toolbars/tab bars, 153-154
status bar, 151-152
text fields, 155
UIScreen class, 155
view geometry, 222-223
coordinate systems, 224
frames, 223-224
transforms, 224
gesture distinction, 329-333
getters
creating custom, 107-109
explained, 106-107
GKPeerPickerController class, 151, 498
GKSession class, 500
GKVoiceChatService class, 512
GPS positioning, 690
Graphics Convert application, 11
grayscale images, 298-299
grouped tables, creating, 473, 477-480
groups (Address Book), 736-738
guides, adding, 184

H

.h file extension, 19
handler methods, adding, 816-817
hardware requirements, 3
header files, 92
Core Data, generating, 759-760
importing, 93
viewing side-by-side with method file, 88
header titles for table sections, creating, 470-471
headers for tables, customizing, 474-476
Hewitt, Joe, 281
hiding status bar, 152
hierarchies, view, 211-213
hints, accessibility, 801
Hockenberry, Craig, 814
host information, retrieving, 552-555
hybrid positioning approaches, 691

hybrid temperature converter example, 170
adapting template, 170
adding view controller, 170
designing interface, 171
editing main.m, 172-173
editing view controller implementation, 171-172
running application, 173

I

IB (Interface Builder). ***See*** **Interface Builder**
icon.png files, 25-26
id type, 99
identities (Xcode), customizing, 85-86
ignoring lock events, 622-624
iLime service, 793
image backdrops, creating, 160
Image Picker, 150, 263
image processing
applying, 293-295
limitations, 295-297
image view animations, 250-251
ImageHelper class, 260-261
images
adding random contact art, 752-754
Address Book images, 733-734
accessing image data, 741-742
background images for tables, creating, 432-433
bitmaps, 291
applying image processing, 293-295
drawing into bitmap contexts, 291-293
image-processing limitations, 295-297
creating from scratch, 281-285
custom camera overlays, 275, 277-278
e-mailing, 272-273
grayscale, 298-299
loading
from application bundle, 258
with ImageHelper class, 260-261
from photo album, 260, 262-265
from sandbox, 258-259
from URLs, 259-260
photo orientation, 288-290
saving to Documents folder, 270-271
screenshots, 290-291
scroll views
creating multiimage paged scroll, 280-281
displaying images in scrollable view, 278-280
selecting and customizing from camera roll, 265-268
snapping photos with iPhone and writing to photo album, 268-270
sources, 257-258
thumbnails, creating, 285-288
time-lapse photos, capturing, 273-275
uploading to TwitPic, 572-575
importing header files, 93
in-app purchase items (StoreKit), creating, 782-786
adding item details, 784-785
application approval, 785-786
pricing section, 783-784
submitting purchase GUI screenshot, 785
inactive device tokens, managing, 685-686
index path access (Core Data), 767
index paths, 132, 425
index titles (Core Data), 768
indexed characters of strings, 126
indexed substrings, requesting, 128
indexes for table sections, creating, 471-472
Info Dark button, 344
Info Light button, 344
Info.plist files, 23-25
list of keys, 821-824
inheriting methods, 98
initializing audio players, 611-612
inserting entities into contexts (Core Data), 761-763
inspecting
class headers, 163
objects in debugger, 55
installing development provisions, 66
instance methods. ***See*** **methods**
instance variables, 91
instances, 94
instant message properties (Address Book), 730-733

Instruments application
definition of, 4
detecting leaks, 59-60, 62
monitoring caching, 62-64
interaction limits, platform differences, 11
interactions, enabling, 160
interactive resize and rotation, 333-338
Interface Builder
adding buttons, 345-347
adding VoiceOver accessibility from, 799-802
converting to Objective-C, 51-53
custom table cells, building, 435-439
definition of, 4
tab bar controllers in, 207-208
table cells with built-in controls, 441-443
temperature converter example, 156-159
adding connections, 163-164
adding labels, 160
adding media to, 157
adding outlets/actions, 162
capturing colors, 165
creating image backdrops, 160
creating new project, 156
defining conversion method, 165-166
editing navigation bar, 159
enabling simulated elements, 160
inspecting class header, 163
replacing main view, 159-160
running application, 166
testing interface, 161
tips for, 184-185
toolbars, creating, 384-385
views, editing, 44-45
.xib files, opening, 43-44
interface creation, 155-156
code-based example, 166-169
hybrid example, 170
adapting template, 170
adding view controller, 170
designing interface, 171
editing main.m, 172-173
editing view controller implementation, 171-172
running application, 173
Interface Builder example, 156-159
adding connections, 163-164
adding labels, 160
adding media to, 157
adding outlets/actions, 162
capturing colors, 165
creating image backdrops, 160
creating new project, 156
defining conversion method, 165-166
editing navigation bar, 159
enabling simulated elements, 160
inspecting class header, 163
replacing main view, 159-160
running application, 166
testing interface, 161
loading .xib files from code example, 173-174
interface design, 143
bars, 146-147
controls, 145-146
geometry, 151-155
hybrid temperature converter example, 171
Interface Builder tips, 184-185
pickers, 146
progress indicators, 147
for rotation, 174-175
autosizing, 176-179
enabling reorientation, 175-176
moving views, 179-180, 182
swapping views, 183
tables, 146
UIView class, 143-144
UIWindow class, 143-144
view controllers, 147-148
address book controllers, 150
GKPeerPickerController class, 151
media player controllers, 151
MFMailComposeViewController class, 150
table controllers, 149-150
UIImagePickerController class, 150
UINavigationController class, 148-149
UITabBarController class, 149
UIViewController class, 148

views
displaying data, 144
making choices, 145
interfaces
declaring, 92
FlipView, 241
Internet, downloading images from, 257
Internet provider positioning, 691
interruptions to audio, handling, 621-622
IP information, retrieving, 552-555
IPA archives, 27
iPhone deployment. *See* deployment
iPhone developer programs
Enterprise Developer Program, 2
Online Developer Program, 2
registering for, 3
Standard Developer Program, 2
table of, 1-2
University Developer Program, 3
iPhone Development Tools list (Organizer), 71
iPhone model differences
OpenGL ES, 9
cameras, 7
core location differences, 8
microphones, 7-8
overview, 7
processor speeds, 9
speakers, 7-8
telephony, 8
vibration support and proximity, 9
iPhone platform limitations
application limits, 11
behavior limits, 12
data access limits, 10
energy limits, 11
interaction limits, 11
memory limits, 10
overview, 9
storage limits, 10
iPhone SDK Simulator. *See* Simulator
iPhone SDK. *See* SDK (Software Developer's Kit)
iPhone servers
creating with Bonjour, 515-520
Mac clients, creating, 520-523
iPhone-specific definitions, recovering, 74-75
iPod library contents, filtering, 645-649
item details for in-app purchase items (StoreKit), 784-785
iTunes Connect, registering for, 3

J–K

JSON (JavaScript Object Notation), 672
converting aps dictionary to, 674-675
payload samples, 674
key-value pairs, custom data in notification payloads, 675
keyboards
dismissing, 366-371
geometry of, 154-155
keychain
persistence of data, 567
sharing between applications, 575-577
storing user credentials, 566-569
KeychainItemWrapper class, 567
keys, dictionary keys, listing, 135
Kosmaczewski, Adrian, 51
Krasner, Glenn, 29

L

labels
accessibility, 800-801
adding, 160
smart labels, 387-388
landscape views, centering, 234-235
languages for item details (StoreKit), 784-785
latitude and longitude, tracking, 691
code listing, 693
location properties, 694-695
step-by-step process, 692
launching applications, receiving notification data, 675-676
laying out table views, 424
layout guides, adding, 184
leaks, memory management, 58-62
length of strings, 126
Library folder, 259
limitations
of iPhone SDK, 12-13
platform limitations, 9-12
Simulator limitations, 5-6
limiting contact picker properties (Address Book), 745-747
lines, calculating, 323-325
listing dictionary keys, 135
Llama Settings project, 813

loading
images
from application bundle, 258
with ImageHelper class, 260-261
from photo album, 260, 262-265
from sandbox, 258-259
from URLs, 259-260
view controller arrays, 198
.xib files from code, 173-174
localization for item details (StoreKit), 784-785
localized alerts, 673
location properties (CLLocation object), 694-695
locations
geocoding, 717-720
map annotations, 710-716
user location annotations, 707-708
viewing, 703-707
lock events, ignoring, 622-624
log files, monitoring, 509-510
logging class information, 103-105
looping audio, 618-620
loops, run loops, creating modal alerts with, 396-399

M

.m file extension, 19
Mac clients for iPhone Bonjour servers, creating, 520-523
mail composition, 150
main view, replacing, 159-160
main.m file
autorelease pools, 19-20
hybrid temperature converter example, editing, 172-173
purpose of, 19
UIApplicationMain function, 20
managed contexts. *See* contexts (Core Data)
map annotations
adding, 710
annotation views, 710-712
creating, 710
geocoding, 717-720
MapAnnotation class, 709-710
responding to annotation button taps, 712-716
MapAnnotation class, 709-710
MapKit. *See also* map annotations
reverse geocoding, 700-702
user location annotations, 707-708
viewing locations, 703-707
masking reflections with Core Graphics, 253-255
measurements in interface design, 151
keyboards, 154-155
navigation bars/toolbars/tab bars, 153-154
status bar, 151-152
text fields, 155
UIScreen class, 155
media. *See also* audio; video
adding to projects, 157
adding to views, 184
media player controllers, 151
Media Queries
creating, 645-649
types of, 645
memory limits, platform differences, 10
memory management, 58
allocating memory, 94
autorelease, 58
autorelease object lifetime, 112
caching, 59
monitoring with Instruments application, 62-64
Clang static analyzer, 64-65
collections, 135
Core Foundation, 116-117
creating autorelease objects, 111-112
creating objects, 110-111, 115-116
deallocating objects, 117-119
explained, 110
high retain counts, 115
leaks, 58-62
releasing memory, 94-95
properties and, 105-106
retained properties, 113-114
retaining autorelease objects, 112-113
menus
creating, 403-405
scrolling, 405
two-item menu example (navigation controllers), 192-193

message forwarding
compiler warnings, 140
explained, 138
implementing, 139
method signatures, building, 139
multiple inheritance, 140-141
undocumented methods of, 141
message tracking, 35
messages, sending to nil, 100. *See also* alerts
method files, viewing side-by-side with header file, 88
method signatures, building, 139
methods, 91
accessor methods, 105
class methods, 101
collapsing, 77
data source methods for tables, 427-428
declaring, 98-99
delegate methods
in alerts, 392-394
for table searches, 467
for table sections, 472
dynamic typing, 96-98
explained, 93, 95-96
frame utility methods, 227-231
for group tables, 478
implementing, 99
inheriting, 98
nesting invocations, 100
for picker views, 482
searchable data source methods, building, 465-466
variadic arguments with alerts, 402-403
MFMailComposeViewController class, 150
MFMailComposeViewControllerDelegate protocol, 272
microphones, model differences, 7-8
MKAnnotation class, 709
MKAnnotationView class, 711
MKMapView class, 144
MKPlaceMark class, 701
MKReverseGeocoder class, 701
MKReverseGeocoderDelegate class, 701
MKUserLocation class, 707
mobile provisions
building, 83
definition of, 23
modal alerts, creating with run loops, 396-399
modal controllers, 190
custom example, 199-201
model differences
OpenGL ES, 9
cameras, 7
core location differences, 8
microphones, 7-8
overview, 7
processor speeds, 9
speakers, 7-8
telephony, 8
vibration support and proximity, 9
model files (Core Data), creating/editing, 758
model-view-controller design pattern. *See* MVC (model-view-controller) design pattern
models (MVC)
data sources, 34-35
message tracking, 35
overview, 34
UIApplication class, 35
modifying Address Book contacts, 748-750
momentary views, pushing, 198-199
monitoring
audio levels, 613
battery state, 594-595
caching with Instruments application, 62-64
status logs, 509-510
motion events, detecting shakes, 603-604
movement, constraining, 305, 307
movies. *See* video
moving
bounded views, 231-232
objects, 185
onscreen objects with accelerometer, 599-601
views, 179-180, 182
MPMediaItem class, 642-644
MPMediaPickerController class, 151, 641, 647
MPMoviePlayer class, 634-636
MPMoviePlayerController class, 151, 634, 653
MPMusicPlayerController class, 151, 649-653
multiimage paged scroll, creating, 280-281

multiline button text, 351
multimedia. ***See*** **audio; video**
multiple buttons in segmented controls, 362-363
multiple inheritance, message forwarding, 140-141
multiple item purchases (StoreKit), 794
multiple provider support for push notifications, 657
multitouch, 303-304
  detecting, 327-329
multivalue record properties (Address Book), 727-730
multiwheel tables, building, 480-484
music. ***See*** **audio**
mutable arrays, 97, 133
mutable buffers, 136
mutable dictionaries, 134
mutable strings, 130
MVC (model-view-controller) design pattern
  controller behavior
    delegation, 30-31
    notifications, 33
    overview, 30
    target-actions, 32
  message tracking, 35
  models, 34-35
  overview, 28-29
  view classes, 29-30

N

names (Bonjour), registering, 528-529
naming
  classes in Cocoa Touch, 92
  views, 184, 219-222
navigating between view controllers example (navigation controllers), 195-197
navigation applications, 37
navigation bars, 146-147
  editing, 159
  geometry of, 153-154
  undo support, 316-317
navigation controllers, 148-149, 187
  custom modal controllers example, 199-201
  custom popping options example, 197-199
  modal controllers, 190
  navigating between view controllers example, 195-197
  persistence example, 204-207
  pushing/popping, 188-189
  segmented control example, 193-195
  setup, 187-188
  tab bars
    example, 201-204
    in Interface Builder, 207-208
  two-item menu example, 192-193
  UINavigationItem class, 189-190
nesting method invocations, 100
network activity indicators, 415-416
network connections. ***See*** **connections**
network status, checking, 545-547
NeXTStep operating system, 91
NIB files, 26
nil, 100
no-button alerts, 394-396
non-consumable purchases, 783
north, detecting direction of, 698-700
notification payloads
  building, 672
    converting aps dictionary to JSON, 674-675
    custom key-value pairs, 675
    localized alerts, 673
    notification types, 673
    receiving data on launch, 675-676
  sending, 676-681
notifications. ***See*** **push notifications**
NSArray class, 97, 133-134
NSBundle class, 137
NSData class, 136
NSDate class, 131-132
NSDateFormatter class, 132, 490
NSDictionary class, 134-135
NSFetchedResultsController class, 150, 764
NSFileManager class, 136-138, 608
NSHomeDirectory() function, 259
NSIndexPath class, 132
NSKeyedArchiver class, 314
NSKeyedUnarchiver class, 314
NSLog function, 103-105
NSMutableArray class, 97, 133
NSMutableData class, 136
NSMutableDictionary class, 134

NSMutableString class, 130
NSNetServiceBrowser class, 520, 540
NSNotificationCenter class, 33, 426
NSNumber class, 131
NSObject class, 94, 102
NSOperation class, 570
NSOperationQueue class, 570
NSSet class, 135
NSString class, 92, 103, 126-130
accessing substrings, 128-129
building strings, 126
changing case, 129
converting to/from C strings, 127
extracting numbers from strings, 130
indexed characters, 126
length of strings, 126
mutable strings, 130
search/replace with, 129
testing strings, 130
writing to/reading from files, 127-128
NSStringFromCGRect() function, 223, 313
NSTimeInterval class, 131
NSTimer class, 132
NSUndoManager class, 457
NSURL class, 136
NSURLConnection class, 557
NSURLCredential class, 565
NSURLRequest class, 569
NSUserDefaults class, 806
NSXMLParser class, 577
numbers
extracting from strings, 130
NSNumber class, 131

O

object layout, viewing, 185
object-oriented programming, 28, 91-92
Objective-C
categories, 120-121
classes
explained, 92-93
hierarchy, 102-103
logging information, 103-105
collections, 101
converting Interface Builder files to, 51-53
dynamic typing, 96-98
explained, 91-92
Foundation classes, 125-126
collections, 133-136
dates, 131-132
file management, 136-138
index paths, 132
NSData, 136
numbers, 131
strings, 126-130
timers, 132
URLs, building, 136
header files, 92
memory management, 94-95
autorelease object lifetime, 112
Core Foundation, 116-117
creating autorelease objects, 111-112
creating objects, 110-111, 115-116
deallocating objects, 117-119
explained, 110
high retain counts, 115
retained properties, 113-114
retaining autorelease objects, 112-113
message forwarding, 138-141
methods, 93-101
objects, 92-94
properties, 105-110
protocols, 122-125
singletons, 119-120
objects. *See also specific objects*
autorelease objects, 111-113
creating, 93-94, 110-111, 115-116
deallocating, 117-119
explained, 92-93
inspecting in debugger, 55
moving, 185
retain counts, 95
online connections, creating with GameKit, 537-540
Online Developer Program, 2
onscreen objects, moving with accelerometer, 599-601
OpenAL audio, 629
OpenGL ES, 9, 37
opening
debugger, 53
.xib files, 43-44

operation queues, 570
operations, 570
optional callbacks, 123-124
Organizer, 69
Console tab, 72
Crash Log tab, 72-73
Devices list, 71
iPhone Development Tools list, 71
Projects and Sources list, 70
Screenshot tab, 73
Summary tab, 71-72
orientable scroll-down alerts, 412-415
orientation
designing for rotation, 174-175
autosizing, 176-179
enabling reorientation, 175-176
moving views, 179-182
swapping views, 183
device orientation, detecting, 601-603
of photos
fixing, 288-290
test images, adding, 290
of status bar, 152
outlets, 162
overcoming GameKit limitations, 523-527
overlays
custom overlays
creating for progress indicators, 409-411
custom camera overlays, 275-278
orientable scroll-down alerts, 412-415
tappable overlays, 411-412

P

page indicators
adding, 376-378
customized paged scroller example, 379-383
parse trees
browsing, 580-582
building, 578
passwords, storing in keychain, 566-569
pasteboards, 524-525
pathToView() function, 214
payloads. *See* notification payloads
payments (StoreKit), responding to, 791-792
peer pickers, 151
peer-to-peer connections. *See* Bonjour; GameKit
peers in GameKit, 498
connection process, 498-502
state changes, 503
people picker (Address Book), 742-745
performance of Media Queries, 647-649
persistence, 311
of keychain data, 567
navigation controllers example, 204-207
persistence through archiving, 314-315
recovering state, 313-314
storing state, 312-313
in text editors, 371
phases of touches, 302
phone calls, 621-622
photo album, 257. *See also* images
loading images from, 260-265
writing photos to, 268-270
picker views
building multiwheel tables, 480-484
cylinder roll example, 484-487
date/time, entering, 487-490
pickers, 146
picking
audio, 641-645
GameKit peers, 498-502
people (Address Book), 742-745
video, 639-640
platform limitations
application limits, 11
behavior limits, 12
data access limits, 10
energy limits, 11
interaction limits, 11
memory limits, 10
overview, 9
storage limits, 10
playing
audio, 611-618
catching end of playback, 614
ignoring lock events, 622-624
initializing audio players, 611-612
looping audio, 618-620
monitoring audio levels, 613
resuming after interruption, 621-622

scrubbing audio, 614
with MPMusicPlayerController class, 649-653
video with MPMoviePlayer, 634-636
plus (+) class methods, 101
pngcrush utility, 10
pop-ups, 403
Pope, Stephen, 29
popping navigation controllers, 188-189, 197-199
populating tables, 427
ports (Bonjour), registering, 528-529
positioning
cell tower positioning, 690
GPS positioning, 690
hybrid approaches, 691
Internet provider positioning, 691
SkyHook Wi-Fi positioning, 690
POST requests, uploading via, 569-572
pragma marks, 76-77
predicates
in fetch requests (Core Data), 770-772
in Media Queries, 646-647
in table searches, 466
preferences tables, 473, 477-480
pricing section for in-app purchase items (StoreKit), 783-784
processor speeds, 9
production environments for push notifications, 677
progress indicators, 147
classes for, 406-407
creating, 407-409
custom overlays, 409-411
projects
adding media to, 157
compiling, 68
creating, 37-39
Detail pane, 41-42
editing views, 44-45
editor windows, 42
from scratch, 48-52
opening .xib files, 43-44
project files, list of, 43
running in simulator, 46
styles of, 37-38
Xcode project window, 40-41
signing compiled, 68-69
Projects and Sources list (Organizer), 70
properties. *See also specific properties*
AddressBook properties
address and instant message properties, 730-733
date properties, 726-730
multivalue record properties, 727-730
attributes, 109-110
of CLLocation object, 694-695
creating, 106-107
custom getters/setters, 107-109
dot notation, 105
explained, 105
memory management, 105-106
of MPMediaItem class, 643-644
of MPMoviePlayerController class, 653
retained properties
assigning values to, 113-114
cautions about, 114
deallocating objects, 118
reassigning, 114
self-assigning, 112
of text fields, 367-368
of UIDatePicker class, 487
of UIView class, 235-236
property lists, serializing/deserializing, 510-512
protocols
adding callbacks, 123
conforming to, 124-125
declaring optional callbacks, 123-124
defining, 122-123
explained, 122
implementing optional callbacks, 124
incorporating, 123
provisioning, 16
mobile provisions, 83
push-specific provisions, 661-662
proxies, 43
proximity sensor, enabling/disabling, 596-597
purchase GUI (StoreKit)
creating, 787-789
screenshot for in-app purchase items, submitting, 785

purchase models (StoreKit)
application submission, 787
explained, 779-781
in-app purchase items, creating, 782-786
purchase GUI, creating, 787-789
purchasing items, 789-794
test accounts, creating, 781-782
validating receipts, 794-796
purchase types (StoreKit), 783
purchasing items (StoreKit), 789-794
multiple items, 794
registering purchases, 792-794
responding to payments, 791-792
restoring purchases, 793-794
signing into test accounts, 790
push notifications, 33
advantages of, 655-656
application registration, 662-663
error handling, 664-665
responding to notifications, 665-666
retrieving device tokens, 663-664
building notification payloads, 672
converting aps dictionary to JSON, 674-675
custom key-value pairs, 675
localized alerts, 673
notification types, 673
receiving data on launch, 675-676
client skeleton example, 667-672
designing for, 686-687
development process
application identifier, generating, 659
push-specific provisions, 661-662
SSL certificate, generating, 660-661
explained, 656
feedback service, 685-686
limitations of, 658-659
multiple provider support, 657
security, 658
sending notification payloads, 676-681
table notifications, 426
Twitter client example, 681-685
push-specific provisions, 661-662
pushing
navigation controllers, 188-189
temporary views, 198-199

Q–R

querying subviews, 214-215
queues, 417, 629-634

random contact art, adding, 752-754
ranges, generating substrings from, 129
reachability, extending UIDevice class for, 547-549. *See also* availability
read-only properties, 106-107
read-write properties, 106
reading
with BonjourHelper class, 530
image data, 258
loading image files with ImageHelper class, 260-261
loading images from application bundle, 258
loading images from photo album, 260-265
loading images from sandbox, 258-259
loading images from URLs, 259-260
strings from files, 127-128
reassigning retained properties, 114
receipts (StoreKit), validating, 794-796
receivers, 99
receiving
GameKit data, 502
notification data on launch, 675-676
recording
audio, 624-634
video, 636-639
records (Address Book)
adding, 734
creating, 734
deleting, 735-736
multivalue record properties, 727-730
retrieving and setting ABRecord strings, 725-726
recovering
iPhone-specific definitions, 74-75
state, 313-314
view hierarchy tree, 213
redo/undo support
in Core Data, 775-778
Redo buttons, adding to tables, 458-460

referencing system address book, 724
reflections
adding to views, 251-252
masking with Core Graphics, 253-255
registration
application identifiers, 15
Bonjour names and ports, 528-529
devices, 14-15, 83
for developer programs, 3
for iTunes Connect, 3
purchases (StoreKit), 792-794
for push notifications, 662-666
registering schemes
adding handler method, 816-817
declaring URL, 815-816
undos, 317-318
relationships (Core Data), 758, 766
releasing memory, 94-95
reliable mode, sending/receiving data, 502
remote notifications. *See* push notifications
remove controls, 452
removing
Address Book records, 735-736
breakpoints, 55
dictionary objects, 135
objects (Core Data), 765-767
selection highlights in table cells, 448-449
simulator data, 48
subviews, 216
table cells, 451-456
tree data structures, 582
reordering
subviews, 216
table cells, 456-457
reorientation, enabling, 175-176
replacing
dictionary objects, 134
main view, 159-160
search/replace, 129
requesting
certificates, 14
indexed substrings, 128
requirements, device capability requirements, 590-592
resizing
frames, 225-226
text editors, 372
responder chain, 603-604
responding
to annotation button taps, 712-716
to payments (StoreKit), 791-792
to push notifications, 665-666
to URL scheme requests, 818-819
restoring purchases (StoreKit), 793-794
resuming audio playback after interruption, 621-622
retain counts, 95, 115
retaining
autorelease objects, 112-113
properties, 109, 113-114, 118
retrieving
ABRecord strings, 725-726
data via pasteboards, 525
device tokens, 663-664
IP and host information, 552-555
objects (Core Data), 763-764
views, 217-218
returning
control to calling application, 817-818
table cells from sections, 470
reusing table cells, 425, 428, 445-446
reverse geocoding, 700-702
review, submitting applications for, 787
root view controllers, 156
rotation
designing for, 174-175
autosizing, 176-179
enabling reorientation, 175-176
moving views, 179-182
swapping views, 183
interactive resize and rotation, 333-338
Rounded Rectangle button, 344
rows in tables, counting, 469-470
run loops, 396-399
running
console, 55-56
debugger, 53
projects in simulator, 46
runtime checks, 75-76

S

sandbox
- loading images from, 258-259
- overview, 27, 257
- sandbox environments for push notifications, 677
- sandbox files, 47

saving images to Documents folder, 270-271
.sb file extension, 28
scaling for push notifications, 686-687
scanning
- for Bonjour services, 540-543
- for connectivity changes, 549-552

screen
- screen orientation for scroll-down alerts, 412-415
- UIScreen class, 155

Screenshot tab (Organizer), 73
screenshots, 290-291, 785
scroll-down alerts, 412-415
scrolling
- changing background color based on, 431-432
- menus, 405
- scroll views
  - creating multiimage paged scroll, 280-281
  - displaying images in scrollable view, 278-280

scrubbing audio, 614
SDK (Software Developer's Kit), 3
- Cocoa Touch, 4-5
- development devices, 5
- downloading, 3
- hardware requirements, 3
- IB (Interface Builder), 4
- Instruments, 4
- limitations of, 12-13
- SDK APIs, accessing from Xcode, 50-51
- Shark, 4
- Simulator, 4-6
- Xcode, 4

search display controllers, 149, 464-465
search tables, Core Data for, 770-772
searchable data source methods, 465-466
searching
- Address Book, 738-740
- Address Book contacts, 735
- dictionaries, 134
- search/replace, 129
- tables
  - delegate methods, 467
  - search display controller, building, 464-465
  - searchable data source methods, building, 465-466

section groups (Core Data), 768
section key paths (Core Data), 767
sectioned tables, 467
- building with Core Data, 769
- counting, 469-470
- data structure, creating, 468-469
- delegate methods with, 472
- header titles, creating, 470-471
- indexes, creating, 471-472
- returning cells from, 470

security
- for push notifications, 658, 686-687
- Security framework, 567
- user credentials, storing in keychain, 566-569

segmented controls, 193-195, 362-363
selected table cells, customizing, 439
selecting
- images from camera roll, 265-268
- from stacked views, 184

selection color for table cells, 429
selection highlights in table cells, removing, 448-449
selectors, 93
self variable, 99
self-assigning properties, 112
sending
- complex data via GameKit, 510-512
- events from controls, 364
- GameKit data, 502
- messages to nil, 100
- notification payloads, 676-681

serializing property lists, 510-512
server mode (peer pickers), 500, 502

servers
in GameKit, 498
iPhone servers
creating with Bonjour, 515-520
Mac clients, creating, 520-523
Web servers, 582-586
services, URL-based
adding handler method, 816-817
cross-promotion, 815
declaring URL, 815-816
implementing custom schemes, 818
overview, 814
responding to URL scheme requests, 818-819
returning control to calling application, 817-818
service downsides, 815
URL schemes, 814-815
session objects, creating, 500
sessions in GameKit, 496-498
sets, 133-135
setters
creating custom, 107-109
explained, 106-107
setting ABRecord strings, 725-726
Settings app, 807
settings schema, 808
shake-controlled undo support, 319-320, 458
shakes, detecting
with accelerometer, 605-608
with motion events, 603-604
sharing keychains between applications, 575-577
sharing data via pasteboards, 524
Shark, 4
showAlert( ) function, 190-191
side-by-side code, viewing, 88
signatures, method signatures, 139
signing
compiled applications, 68-69
into test accounts (StoreKit), 790
simple direct manipulation interface, 304-305
simulated elements, enabling, 160
Simulator
clipboard for, 48
definition of, 4
explained, 46-48
limitations, 5-6
running projects in, 46
singletons, 101, 119-120
site availability, checking, 555-557
sizing
frames, 225-226
interactive resize and rotation, 333-338
SkyHook Wi-Fi positioning, 690
sleep mode, ignoring, 622-624
sliders, custom slider thumbs, 356-361
Smalltalk, 28, 91
smart labels, 387-388
Software Developer's Kit. *See* SDK
sorting tables, 462-463
sound. *See* audio
source files
application delegate, 20-21
main.m, 19-20
overview, 18-19
view controller, 21-22
speakers, 7-8
speed
computing, 696-697
for Media Queries, 647-649
tracking, 695-696
springs, 176
sqlite3 utility, 762
SSL certificates, generating for push notifications, 660-661
stacked views, selecting from, 184
Standard Developer Program, 2
state
recovering, 313-314
state changes in GameKit peers, 503
storing, 312-313
of table cells, retaining, 443-445
static analyzer, 64-65
static typing, 96
status bar, 151-152
status logs, monitoring, 509-510
storage limits, 10
StoreKit
application submission, 787
explained, 779-781
in-app purchase items
adding item details, 784-785
application approval, 785-786

creating, 782-786
pricing section, 783-784
submitting purchase GUI screenshot, 785
purchase GUI, 787-789
purchasing items, 789-794
test accounts, 781-782
validating receipts, 794-796

storing
data via pasteboards, 524
state, 312-313
user credentials in keychain, 566-569

stretching views, 349

strings. *See also* NSString class
ABRecord strings, 725-726
converting arrays to, 134
format specifiers, 104

struts, 176

subclassing UIControl class, 363-366

submitting
applications for review, 787
purchase GUI screenshot (StoreKit), 785

subscription purchases, 784

substrings, accessing, 128-129

subviews
adding, 216
querying, 214-215
removing, 216
reordering, 216
view callbacks, 216-217

Summary tab (Organizer), 71-72

swapping views, 183, 239-240

swipes, 329, 453

switches, 354-356

symbolication, 73

synchronize method, 807

synchronous downloads, 557-560

sysctl() method, 592-593

sysctlbyname() method, 592-593

System Audio services, 417-418

System Configuration framework, 546

system information, 589-590, 592-593

system sounds
creating, 417-420
delays, 419

T

tab bars, 149
geometry of, 153-154
in Interface Builder, 207-208
navigation controllers example, 201-204
persistence example, 204-207
Tab bar applications, 37

table controllers, 149-150

table notifications, 426

tables, 146
background color, 430-432
background image, 432-433
cells. *See* cells (table)
creating, 424
data sources, 427-428, 767-770
date/time, entering, 487-490
editing in Core Data, 773-775
font table example, 428-430
group tables, 477-480
grouped tables, 473
headers/footers, 474-476
multiwheel tables, 480-484
populating, 427
searching
delegate methods, 467
search display controller, building, 464-465
searchable data source methods, building, 465-466
sections, 467
counting, 469-470
data structure, creating, 468-469
delegate methods with, 472
header titles, creating, 470-471
indexes, creating, 471-472
returning cells from, 470
sorting, 462-463
UITableView class, 423-426
UITableViewController class, 424
undo support, 457-462
adding Undo/Redo buttons, 458-460
in Core Data, 775-778
shake-to-edit, 458

tagging views, 173-174, 217-218

tappable overlays, 411-412

taps, 329

target-actions, 32
target settings, 83
TCPConnection class, 515
TCPServer class, 515
teams, 13-14
tearing down tree data structures, 582
telephony, 8
temperature converter example
code-based, 166–169
hybrid
adapting template, 170
adding view controller, 170
designing interface, 171
editing main.m, 172–173
editing view controller implementation, 171–172
running application, 173
Interface Builder, 156–159
adding connections, 163–164
adding labels, 160
adding media to, 157
adding outlets/actions, 162
capturing colors, 165
creating image backdrops, 160
creating new project, 156
defining conversion method, 165–166
editing navigation bar, 159
enabling simulated elements, 160
inspecting class header, 163
replacing main view, 159–160
running application, 166
testing interface, 161
loading .xib files, 173–174
templates
adapting, 170
creating custom, 86–88
creating projects, 37–39
Detail pane, 41–42
editing views, 44–45
editor windows, 42
opening .xib files, 43–44
project files, list of, 43
running in simulator, 46
Xcode project window, 40–41
moving views, 180–182
temporary views, pushing, 198-199
testing
accessibility, 803–806
arrays, 134
interface, 161
network status, 545–547
strings, 130
test accounts (StoreKit)
creating, 781–782
signing into, 790
test images, adding, 290
touches, 307–308
circular hit test, 308
testing against bitmap alpha levels, 309–311
tethering, 6-7
text
displaying in action sheets, 405–406
multiline button text, 351
text fields
geometry of, 155
keyboards, dismissing, 366–369
properties, 367–368
text entry filtering, 374–376
text input, requesting via alerts, 399–402
text editors, 371-374
text views
keyboards, dismissing, 370–371
smart labels, 387–388
text editors, creating, 371–374
thumbnails, creating from images, 285-288
thumbs (sliders), 356
time-lapse photos, capturing, 273-275
time/date
entering in tables, 487–490
formatting, 490–493
timers, NSTimer class, 132
timestamp property (CLLocation object), 695
To Do List view hierarchy, 212-213
toggle buttons, 354-356
Toll Free Bridging, 117
toolbars, 146-147
creating
in Interface Builder, 384–385
tips for, 387
in Xcode, 385–386
customizing, 56–57
geometry of, 153–154

touch wheels, 363-366
touch-based painting, 321-323
touches
calculating lines, 323-325
constraining movement, 305-307
detecting circles, 325-327
gesture distinction, 329-333
interactive resize and rotation, 333-338
methods, 302-303
multitouch, 303-304, 327-329
overview, 301-302
persistence, 311-315
phases, 302
simple direct manipulation interface, 304-305
testing, 307-311
touch-based painting, 321-323
touching views, 303
tracking, 364
undo support
child-view undo support, 316
creating undo managers, 316
custom undo routine, 318-319
navigation bars, 316-317
registering undos, 317-318
shake-controlled undo support, 319-320
tracking
latitude and longitude
code listing, 693
location properties, 694-695
step-by-step process, 691-692
speed, 695-696
touches, 364
transaction observers, 789
transforming views, 224, 232
affine transform of UIView, 233-234
centering landscape views, 234-235
transitions (view)
Core Animation calls, 244-246
Core Animation transitions, 242-244
curl transitions, 246-247
transparency, animating transparency changes in views, 237-238
tree data structures, converting XML data to, 577-582
troubleshooting
device orientation sensing, 602
enabling interactions, 160
TwitPic, uploading images to, 572-575
Twitter client example (push notifications), 681-685
Twitterrific, 814
two-item menu example (navigation controllers), 192-193

U

UDIDs (unique device identifiers), 14
UIAcceleration class
locating "up," 597-599
moving onscreen objects, 599-601
UIAccelerometerDelegate protocol, 597
UIAccessibility protocol, 802
UIActionSheet class, 145, 391, 394, 403-405
UIActivityIndicatorView class, 147, 406-407
UIAlertView class, 145, 391-394
UIApplication class, 35, 119
UIApplicationMain function, 20
UIBarButtonItem class, 145, 347
UIButton class, 145, 344-345, 351
UIControl class. *See also* controls
control events, 341-344
subclassing, 363-366
types of controls, 341
UIDatePicker class, 146, 487-490
UIDevice class, 119-121, 589-590
battery state, monitoring, 594-595
device orientation, detecting, 601-603
enabling/disabling proximity sensor, 596-597
extending for reachability, 547-549
retrieving IP and host information, 552-555
UIEdgeInsetsInsetRect() function, 231
UIImage class. *See* images
UIImageJPEGRepresentation() function, 270
UIImageOrientation class, 288-290
UIImagePickerController class, 148-150, 262-267, 638
UIImagePNGRepresentation() function, 270
UIImageView class, 144
UIImageWriteToSavedPhotosAlbum() function, 268

UILabel class, 144, 387
UINavigationBar class, 146
UINavigationController class, 30, 148-149, 187. *See also* navigation controllers
UINavigationItem class, 189-190
UIPageControl class, 146, 376-378
UIPasteboard class, 524
UIPickerView class, 146, 480-482
UIProgressView class, 147, 406-409
UIResponder class, 102
UIScreen class, 155
UIScrollView class, 144
UISearchBar class, 147, 464
UISearchDisplayController class, 464
UISegmentedControl class, 146, 193-195, 362
UISlider class, 146, 356-361
UISwitch class, 146, 354
UITabBar class, 147
UITabBarController class, 30, 149, 201
UITableView class, 30, 146, 423. *See also* tables
UITableViewCellStyleDefault class, 433
UITableViewCellStyleSubtitle class, 433
UITableViewCellStyleValue1 class, 434
UITableViewCellStyleValue2 class, 434
UITableViewController class, 149, 424
  font table example, 428
  populating tables, 427
  views, laying out, 424
UITextField class, 146
  keyboards, dismissing, 366-369
  text entry filtering, 374-376
UITextInputTraits protocol, 367
UITextView class, 144
  keyboards, dismissing, 370-371
  smart labels, 387-388
  text editors, creating, 371-374
UIToolbar class, 147
UITouchPhaseBegan class, 302
UITouchPhaseCancelled class, 302
UITouchPhaseEnded class, 302
UITouchPhaseMoved class, 302
UITouchPhaseStationary class, 302
UIVideoEditorController class, 640
UIView class, 29, 143-144. *See also* views
UIViewController class, 29-30, 148
UIWebView class, 144
UIWindow class, 143-144
undo support
  child-view undo support, 316
  in Core Data, 775-778
  creating undo managers, 316
  custom undo routine, 318-319
  navigation bars, 316-317
  registering undos, 317-318
  shake-controlled undo support, 319-320
  in tables, 457-462
    adding Undo/Redo buttons, 458-460
    shake-to-edit, 458
  in text editors, 371
  Undo buttons, adding to tables, 458-460
  undo managers, creating, 316
unique device identifiers (UDIDs), 14
University Developer Program, 3
unreliable mode, sending/receiving data, 502
updating
  application badges, 416-417
  fetch requests (Core Data), 764
  loadView method, 174
uploading
  form data, 572-575
  via POST requests, 569-572
uploads to App Store, debugging, 81-82
Urban Airship, 793
URL-based services, creating
  adding handler method, 816-817
  cross-promotion, 815
  declaring URL, 815-816
  implementing custom schemes, 818
  overview, 814
  responding to URL scheme requests, 818-819
  returning control to calling application, 817-818
  service downsides, 815
  URL schemes, 814-815
URLs
  building, 136
  loading images from, 259-260
user credentials, storing in keychain, 566-569

user defaults, checking, 813-814
user interface design. *See* interface design
user location annotations, 707-708
utilities, 10-11, 38

V

validating
receipts (StoreKit), 794-796
text entries, 374-376
variables, deallocating objects, 118
variadic arguments, 402-403
vibration, 9, 418-420
video
editing, 639-641
picking, 639-640
playing, 634-636
recording, 636-639
view callbacks, 216-217
view classes, 29-30
view controllers, 21-22, 147-148. *See also* navigation controllers
adding, 170
address book controllers, 150
alerts, displaying, 190-191
arrays, loading, 198
custom modal controllers example, 199-201
editing implementation, 171-172
GKPeerPickerController class, 151
media player controllers, 151
MFMailComposeViewController class, 150
root view controller, 156
table controllers, 149-150
UIImagePickerController class, 150
UINavigationController class, 148-149
UITabBarController class, 149
UIViewController class, 148
view-based applications, 38
viewing
locations
finding best location match, 704-707
overview, 703-704
object layout, 185
side-by-side code, 88
views, 143-144
adding media to, 184
animations
bouncing views, 248-250
building UIView animation blocks, 236-237
callbacks, 237
Core Animation calls, 244-246
Core Animation transitions, 242-244
curl transitions, 246-247
fading views in and out, 237-238
flipping views, 240-241
image view animations, 250-251
overview, 236
swapping views, 239-240
annotation views, 710-712
bounded views, moving randomly, 231-232
creating, 50
display and interaction traits, 235-236
displaying data, 144
draggable views, 304-305
editing, 44-45
fading in and out, 237-238
flipping, 240-241
frames
adjusting sizes, 225-226
CGRect structure, 223-224, 227
overview, 224-225
utility methods, 227-231
geometry
keyboards, 154-155
navigation bars/toolbars/tab bars, 153-154
status bar, 151-152
text fields, 155
UIScreen class, 155
main view, replacing, 159-160
making choices, 145
moving, 179-182
naming, 184, 219-222
picker views, 484-487
reflections
creating, 251-252
masking with Core Graphics, 253-255

retrieving, 217-218
scroll views
creating multiimage paged scroll, 280-281
displaying images in scrollable view, 278-280
stacked views, 184
stretching, 349
subviews
adding, 216
querying, 214-215
removing, 216
reordering, 216
view callbacks, 216-217
swapping, 183, 239-240
table views, 424
tagging, 173-174, 217-218
temporary views, 198-199
touch-based painting, 321-323
touching, 303
transforming, 232
affine transform of UIView, 233-234
centering landscape views, 234-235
transitions
Core Animation calls, 244-246
Core Animation transitions, 242-244
curl transitions, 246-247
view geometry, 222-223
coordinate systems, 224
frame rectangles, 223-224
transforms, 224
view hierarchies, 211-212
recovering view hierarchy tree, 213
To Do List view hierarchy, 212-213
visualizing cell reuse, 445-446
Voice Chat (GameKit), 512-515
Voice Control, 799
VoiceOver accessibility
adding from code, 802-803
adding with Interface Builder, 799
enabling accessibility, 802
hints, 801
labels, 800-801
traits, 802
common VoiceOver gestures, 805-806
overview, 799
testing on iPhone, 803-806
testing with simulator, 803
volume alert, 420-421
Vulcano, Emanuele, 816

W

warnings, treating as errors, 98
Web servers, 582-586
Web sites, authentication challenges, 565-566
wheel tables, 480-484
WiFi connections, 528-537
window-based applications, 38
writing
collections to files, 135-136
photos to photo album, 268-270
strings to files, 127-128
WWDR intermediate certificate, 14

X–Y–Z

.xcdatamodel files, 758
Xcode
accessing SDKAPIs, 50-51
building custom buttons, 348-351
compiler directives, 73-76
debugger
breakpoints, 53-55
console, 55-57
customizing toolbars, 56-57
inspecting objects, 55
opening, 53
running, 53
zombies, 57
definition of, 4
projects
creating, 37-39
creating from scratch, 48-52
Detail pane, 41-42
editing views, 44-45
editor windows, 42
opening .xib files, 43-44
project files, list of, 43
project window, 40-41
running in simulator, 46

side-by-side code, viewing, 88
templates. *See* templates
toolbars, 385-386
Xcode identities, customizing, 85-86
Xcode Organizer. *See* Organizer
.xib files, 19
loading from code, 173-174
opening, 43-44
XML data, converting to tree data structures, 577-582
XMLParser class, 578

Yahoo Geocoding API, 717

zombies, enabling, 57