Erica Sadun

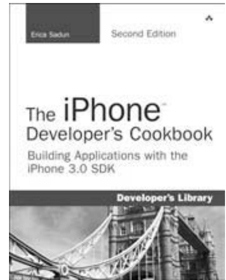
Special Electronic-Only Edition

The iOS 4 Developer's Cookbook

Core Concepts and Essential Recipes for iOS Programmers

Developer's Library

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



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

—**Tim Burks**, iPhone Software Developer, TootSweet Software

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

—**Jacqui Cheng**, Associate Editor, *Ars Technica*

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

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

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

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

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

—**John Zorko**, Sr. Software Engineer, Mobile Devices

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

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

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

—**Roberto Gamboni**, Software Engineer, AT&T Interactive

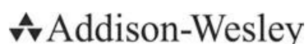
"It's rare that developer cookbooks can both provide good recipes and solid discussion of fundamental techniques, but Erica Sadun's book manages to do both very well."

—**Jeremy McNally**, Developer, entp

The iOS 4 Developer's Cookbook

Core Concepts and Essential Recipes for iOS Programmers

Erica Sadun



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

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

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

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

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

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

International Sales
international@pearsoned.com

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

Visit us on the Web: informit.com/aw

Copyright © 2012 Pearson Education, Inc.

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

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

ISBN-13: 978-0-132-91933-3
ISBN-10: 0-132-91933-8

Senior Acquisitions Editor
Chuck Toporek

Senior Development Editor
Chris Zahn

Managing Editor
Kristy Hart

Project Editors
Jovana Shirley

Samantha Sinkhorn

Proofreader
Bart Reed

Editorial Assistant
Olivia Basegio

Cover Designer
Gary Adair

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

Table of Contents

Preface

1 Introducing the iOS SDK

2 Building Your First Project

3 Objective-C Boot Camp

4 Designing Interfaces

5 Working with View Controllers

6 Assembling Views and Animations

7 Working with Images

8 Gestures and Touches

9 Building and Using Controls

10 Working with Text

11 Creating and Managing Table Views

12 A Taste of Core Data

Acknowledgments

This book would not exist without the efforts of Chuck Toporek (my editor and whip-cracker), Chris Zahn (the awesomely talented development editor), and Olivia Basegio (the faithful and rocking editorial assistant who kept things rolling behind the scenes). Special thanks go to Karen Gettman (Chuck's Editor-in-Chief) for her continued support of this ever-growing (and I do mean growing—just check out the page count) book. Also, a big thank you to the entire Addison-Wesley/Pearson production team, specifically Kristy Hart, Jovana Shirley, Samantha Sinkhorn, Gary Adair, and Bart Reed. Thanks also to the crew at Safari for getting my book up in Rough Cuts and for quickly fixing things when technical glitches occurred.

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

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

Special thanks go out to my family and friends, who supported me through month after month of new beta releases and who patiently put up with my unexplained absences and frequent howls of despair. I appreciate you all hanging in there with me. And thanks to my children for their steadfastness, even as they learned that a hunched back and the sound of clicking keys are a pale substitute for a proper mother. My kids

provided invaluable assistance over the last few months by testing applications, offering suggestions, and just being awesome people. I am such an insanely lucky mom that these kids are part of my life.

About the Author

Erica Sadun is the bestselling author, coauthor, and contributor to several dozen books on programming, digital video and photography, and web design, including the widely popular *The iPhone Developer's Cookbook: Building Applications with the iPhone 3.0 SDK, 2nd Edition*. Over the years, she has blogged for TUAW.com, Mac Devcenter, Lifehacker, and Ars Technica. In addition to being the author of dozens of iOS-native applications, Erica holds a Ph.D. in Computer Science from Georgia Tech's Graphics, Visualization and Usability Center. A geek, a programmer, and an author, she's never met a gadget she didn't love. When not writing, she and her geek husband parent three geeks-in-training, who regard their parents with restrained bemusement when they're not busy rewiring the house or plotting global dominance.

Preface

This is not the iOS Cookbook you're expecting.

Instead, it is the iOS Cookbook you *were* expecting. Last year. When iOS 4 debuted, my editor and I had a hard decision to make: Publish the book on iOS 4 and don't include Xcode 4 material, or hold off until Apple released Xcode 4. We chose to hold off for Xcode 4, feeling that many people would expect to see it covered in the book. What we couldn't anticipate, however, is that Apple's NDA would last until Spring 2011. We assumed Xcode 4 would release much sooner; so much for assuming.

By the time Apple finally lifted the NDA on Xcode 4, iOS 5 was—presumably—right around the corner. We were stuck between a rock and an iOS release. We decided to update the book to iOS 4.3, and are making it available in Rough Cuts on Safari. We're also going to make the iOS 4.3 version available as an electronic book only (that is, we aren't planning to print this version), at a discounted price so you can read it on your iPhone or iPad. This is the version you're currently reading. What's more, if you bought the electronic book version of *The iOS 4 Developer's Cookbook* from InformIT.com, you'll be notified when the iOS 5 version is available so you can purchase the new edition at a discounted rate.

But what about iOS 5?

Well, in the meantime, writing for *The iOS 5 Developer's Cookbook* is well underway, and this time we have a totally new strategy. The print version of the book will essentially mirror the iOS 4.3 version, except it will be updated for iOS 5, with some new material added. The print book will carry the same subtitle: *Core Concepts and Essential Recipes for iOS Programmers*, meaning that the book will cover the basics of what you need to know to get started. That version of the book will hit store shelves around mid-November 2011, and it will be about 700–800 pages in length. For someone who's just starting out as an iOS developer, this will be the ideal book for them, as it will cover the tools (Xcode and Interface Builder), the language (Objective-C), and the basic elements common to pretty much every iOS app out there (such as table views, custom controls, split views, and the like).

But we're not stopping there. See, mid-August 2011 is a cutoff date for getting the book to print this year. While the book is in production, my plan is to continue writing and adding more advanced material to *The iOS 5 Developer's Cookbook*, along with a bunch of new chapters that won't make it to print. Our plan is to combine all of that material to create *The Ultimate iOS 5 Developer's eBook*. This enhanced and expanded volume will only be offered electronically (no one wants to kill that many trees).

Back to the iOS 4 version.

This iOS 4 update gives you a sense of the material you'll find in *The Ultimate iOS 5 Developer's eBook*, with its page-busting chapters. The text chapter goes into CoreText in great detail, the controls chapter offers many custom controls, the gestures chapter tells you how to create your own recognizers, and so forth. Advanced material for these chapters will be updated for iOS 5 compatibility and expanded to include new iOS 5 technologies. That's what the eBook will be all about.

The printed version is going to be for getting going. It's for someone who wants to take the how-to along with them to sit in a coffee shop, catch up on a cruise ship, or read at the beach. It's going to focus on the core iOS 5 getting-started material (essentially Chapters 1–6) and then offer compressed versions of the remaining book. It will provide the know-how new developers need to start using the technology without taking up the page count with the esoteric.

Until then, I give you the iOS 4 Cookbook that could have been. There are references occasionally to chapters that aren't included, and an appendix that is now iOS 5–only (so it can't be included here). Available sample code will be found at https://github.com/erica/iOS-4-Cookbook, all of it written prior to WWDC 2011. The "rough" in Rough Cuts is there for a reason. This book has not been entirely polished, tech edited, or fully revised. Unlike other Rough Cut editions, there will be no in-place edits as the book travels to print. This book will never be printed.

If you have suggestions, bug fixes, corrections, or any thing else you'd like to contribute to the iOS 5 edition, please contact me at erica@ericasadun.com. Let me thank you all in advance. I appreciate all feedback that helps make this a better, stronger book.

What You'll Need

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

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

University Student Program

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

Your Roadmap to Mac/iOS Development

As mentioned earlier, one book can't be everything to everyone. And try as I might, if we were to pack everything you'd need to know into this book, you wouldn't be able to pick it up. There is, indeed, a lot you need to know to develop for the Mac and iOS platforms. If you are just starting out and don't have any programming experience, your first course of action should be to take a college-level course in the C programming language. 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.

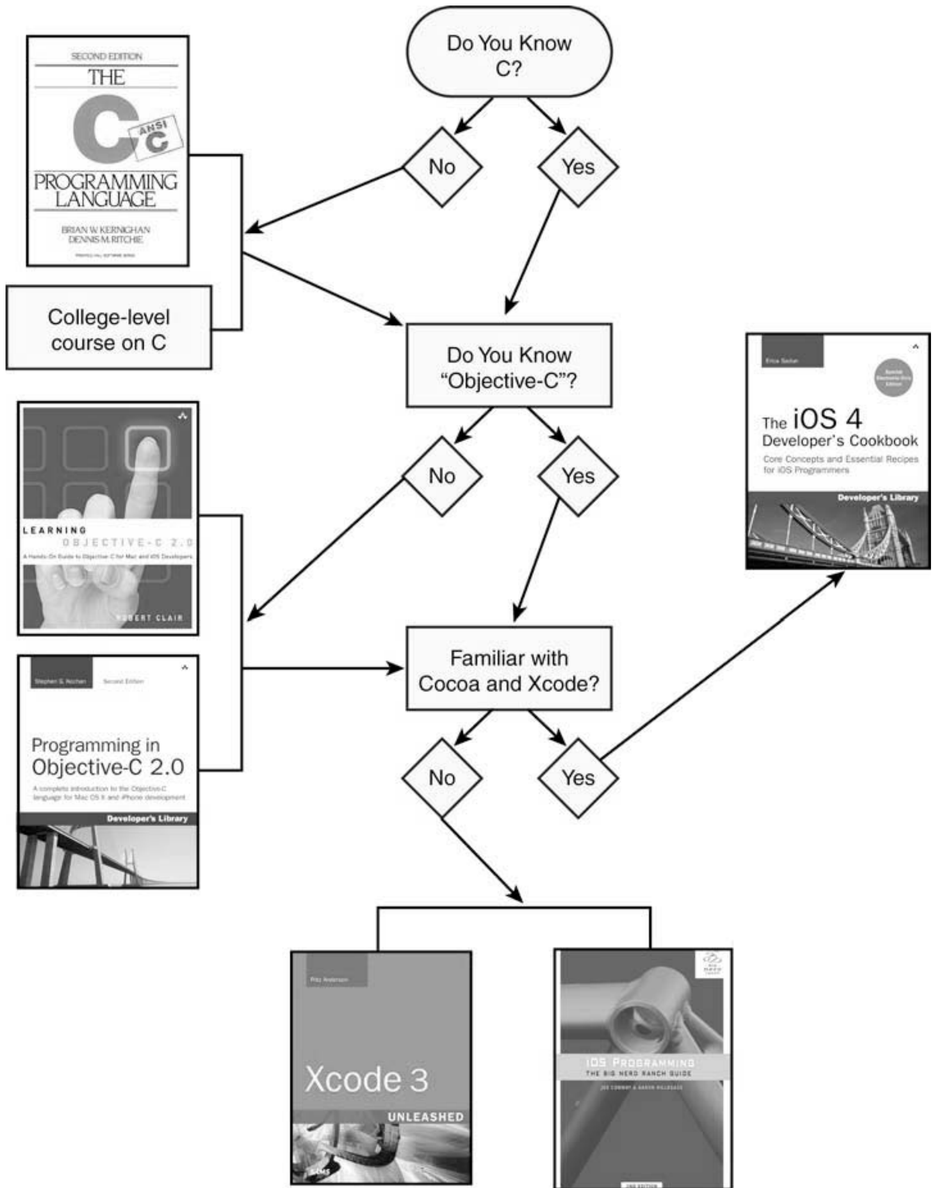


Figure P-1 What it takes to be an iOS programmer.

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 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, 2011).

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 coauthor of *iOS Programming: The Big Nerd Ranch Guide, Second Edition* and 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 and went ballistic in Xcode 4), 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 4 Development*, by Dave Mark, Jack Nutting, and Jeff LaMarche (Apress, 2011), 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, you'll know what I'm talking about. The time you spend at those conferences talking with other developers—and in the case of WWDC, talking with Apple's engineers—is well worth the expense if you are a serious developer.

How This Book Is Organized

This book offers single-task recipes for the most common issues new iOS developers face: laying out interface elements, responding to users, accessing local data sources, and connecting to the Internet. Each chapter groups related tasks together, allowing you

---

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

[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.

to jump directly to the solution you're looking for without having to decide which class or framework best matches that problem.

The iOS 4 Developer's Cookbook offers you "cut-and-paste convenience," which means you can freely reuse the source code from recipes in this book for your own applications and then tweak the code to suit your app's needs.

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

- **Chapter 1, "Introducing the iOS SDK"**—Chapter 1 introduces the iOS SDK and explores iOS as a delivery platform, limitations and all. It explains the breakdown of the standard iOS application and helps you get started with the iOS Developer Portal.
- **Chapter 2, "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 iOS, you'll appreciate this basic skills chapter. Objective-C is the standard programming language for both iOS and for Mac OS X. It offers a powerful object-oriented language that lets you build applications that leverage Apple's Cocoa and Cocoa Touch frameworks. Chapter 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 iOS's library of visual classes. It surveys these classes and their geometry. In this chapter, you learn how to work with these visual classes and discover how to handle tasks such as device reorientation. You'll read about solutions for laying out and customizing interfaces and learn about hybrid solutions that rely both on Interface Builder–created interfaces and Objective-C-centered ones.
- **Chapter 5, "Working with View Controllers and Split Views"**—The iOS paradigm in a nutshell is this: small screen, big virtual worlds. In Chapter 5, you discover the various view controller classes that enable you to enlarge and order the virtual spaces your users interact with. You learn how to let these powerful objects perform all the heavy lifting when navigating between iOS application screens or breaking down iPad applications into master-detail views.
- **Chapter 6, "Assembling Views and Animations"**—Chapter 6 introduces iOS views, objects that live on your screen. You see how to lay out, create, and order

your views to create backbones for your applications. You read about view hierarchies, geometries, and animations—features that bring your iOS applications to life.

- **Chapter 7, "Working with Images"**—Chapter 7 introduces images, specifically the `UIImage` class, and teaches you all the basic know-how you need for working with iOS images. You learn how to load, store, and modify image data in your applications. You see how to add images to views and how to convert views into images. And you discover how to process image data to create special effects, how to access images on a byte-by-byte basis, and how to take photos with your device's built-in camera.
- **Chapter 8, "Gestures and Touches"**—On iOS, the touch provides the most important way that users communicate their intent to an application. Touches are not limited to button presses and keyboard interaction. Chapter 8 introduces direct manipulation interfaces, multitouch, and more. You see how to create views that users can drag around the screen and read about distinguishing and interpreting gestures, as well as how to create custom gesture recognizers.
- **Chapter 9, "Building and Using Controls"**—Control classes provide the basis for many of iOS's interactive elements, including buttons, sliders, and switches. This chapter introduces controls and their use. You read about standard control interactions and how to customize these objects for your application's specific needs. You even learn how to build your own controls from the ground up, as Chapter 9 creates custom switches, star ratings controls, and a virtual touch wheel.
- **Chapter 10, "Working with Text"**—From text fields and text views to iOS's new and powerful Core Text abilities and inline spelling checkers, Chapter 10 introduces everything you need to know to work with iOS text in your apps.
- **Chapter 11, "Creating and Managing Table Views"**—Tables provide a scrolling interaction class that works particularly well on a small, cramped device. Many, if not most, apps that ship with the iPhone and iPod touch center on tables, including Settings, YouTube, Stocks, and Weather. Chapter 11 shows how iPhone tables work, what kinds of tables are available to you as a developer, and how you can use table features in your own programs.
- **Chapter 12, "A Taste of Core Data"**—Core Data offers managed data stores that can be queried and updated from your application. It provides a Cocoa Touch–based object interface that brings relational data management out from SQL queries and into the Objective-C world of iPhone development. Chapter 12 introduces Core Data. It provides just enough recipes to give you a taste of the technology, offering a jumping-off point for further Core Data learning. You learn how to design managed database stores, add and delete data, and query that data from your code and integrate it into your `UIKit` table views.

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 five or seven or nine individual files at once. Offering a single file concentrates that story, allowing access to that idea in a single chunk.

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

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

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

For the most part, the examples for this book use a single application identifier: com.sadun.helloworld. You need to replace this identifier with one that matches your provision profile. This book uses one identifier to avoid clogging up your iOS devices with dozens of examples at once. Each example replaces the previous one, ensuring that your home screen remains relatively uncluttered. If you want to install several examples at once, simply edit the identifier, adding a unique suffix such as com.sadun.helloworld.table-edits. Your Team Provisioning Profile matches com.sadun.helloworld, allowing you to install compiled code to devices; just make sure to update your signing identity in each project's build settings.

Getting the Sample Code

The source code for this book can be found at the open-source GitHub hosting site at http://github.com/erica/iOS-4-Cookbook. There, you find a chapter-by-chapter collection of source code that provides working examples of the material covered in this book.

Sample code is never a fixed target. It continues to evolve as Apple updates its SDK and the Cocoa Touch libraries. Get involved. You can pitch in by suggesting bug fixes and corrections as well as by expanding the code that's on offer. GitHub allows you to fork repositories and grow them with your own tweaks and features, and share those back to the main repository. If you come up with a new idea or approach, let 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 website, 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.

We Want to Hear from You!

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

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

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

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

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

3

Objective-C Boot Camp

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

The Objective-C Programming Language

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

Smalltalk is one of the earliest and best-known object-oriented languages. It was developed at Xerox PARC as a dynamically typed interactive language. Cox layered Smalltalk's object and message passing system on top of standard C to create his new language. This approach allowed programmers to continue using familiar C-language development while accessing object-based features from within that language. In the late 1980s, Objective-C was adopted as the primary development language for the NeXTStep operating system by Steve Jobs's startup computer company NeXT. NeXTStep became both the spiritual and literal ancestor of OS X. The current version of Objective-C is 2.0, which was released in October 2007 along with OS X Leopard.

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

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

Note

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

Classes and Objects

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

Each class lists its instance variables and methods in a public header file using the standard C .h convention. For example, you might define a `Car` object like the one shown in Listing 3-1. The Car.h header file shown here contains the interface that declares how a `Car` object is structured. Note that all classes in Objective-C should be capitalized.

Listing 3-1 Declaring the Car Interface (Car.h)

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

In Objective-C, the `@` symbol is used to indicate certain keywords. The two items shown here (`@interface` and `@end`) delineate the start and end of the class interface definition.

This class definition describes an object with three instance variables: year, make, and model. These three items are declared between the braces at the start of the interface.

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

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

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

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

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

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

Note

The code for this example (and all the examples in this chapter) is found in the sample code for this book. See the Preface for details about downloading the book sample code from the Internet.

Creating Objects

To create an object, you tell Objective-C to allocate the memory needed for the object and return a pointer to that object. Because Objective-C is an object-oriented language, its syntax looks a little different from regular C. Instead of just calling functions, you ask

an object to do something. This takes the form of two elements within square brackets, the object receiving the message followed by the message itself:

```
[object message]
```

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

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

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

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

Memory Allocation

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

That extra field is from the `NSObject` class. Notice `NSObject` at the right of the colon next to the word `Car` in the class definition of Listing 3-1. `NSObject` is the parent class of `Car`, and `Car` inherits all instance variables and methods from this parent. That means that `Car` is a type of `NSObject` and any memory allocation needed by `NSObject` instances is inherited by the `Car` definition. So that’s where the extra 4 bytes come from.

The final size of the allocated object is 16 bytes in total. That size includes two 4-byte `NSString` pointers, one 4-byte `int`, and one 4-byte allocation inherited from `NSObject`. You can easily print out the size of objects using C’s `sizeof` function. This code uses standard C `printf` statements to send text information to the console. `printf` commands work just as well in Objective-C as they do in ANSI C.

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

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

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

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

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

Releasing Memory

In C, you allocate memory with `malloc()` or a related call and free that memory with `free()`. In Objective-C, you allocate memory with `alloc` and free it with `release`. (In Objective-C, you can also allocate memory a few other ways, such as by copying other objects.)

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

As discussed in Chapter 2, "Building Your First Project," releasing memory is a little more complicated than in standard C. That's because Objective-C uses a reference-counted memory system. Each object in memory has a retain count associated with it. You can see that retain count by sending `retainCount` to the object, although in the real world you should never rely on using `retainCount` in your software deployment. It's used here only as a tutorial example to help demonstrate how a retain count works.

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

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

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

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

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

Sending messages to freed objects will crash your application. When the second `printf` executes, the `retainCount` message is sent to the already-freed `myCar`. This creates a memory access violation, terminating the program. As a general rule, it's good practice to assign instance variables to `nil` after the final release that deallocates the object. This prevents the `FREED(id)` error you see here when you access an already-freed object:

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

There is no garbage collection on iOS SDK. As a developer, you must manage your objects. Keep them around for the span of their use and free their memory when you are finished. Read more about basic memory management strategies later in this chapter.

Methods, Messages, and Selectors

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

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

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

```
printCarInfo(myCar);
```

but in Objective-C, you say

```
[myCar printCarInfo];
```

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

Dynamic Typing

Objective-C uses dynamic typing in addition to static typing. Static typing restricts a variable declaration to a specific class at compile time. With dynamic typing, the runtime system, not the compiler, takes responsibility for asking objects what methods they can perform and what class they belong to. That means you can choose what messages to send and which objects to send them to as the program runs. This is a powerful feature—one that is normally identified with interpreted systems such as Lisp. You can choose an object, programmatically build a message, and send the message to the object—all without knowing which object will be picked and what message will be sent at compile time.

With power, of course, comes responsibility. You can only send messages to objects that actually implement the method described by that selector (unless that class can handle messages that don't have implementations by implementing Objective-C invocation forwarding, which is discussed at the end of this chapter). Sending `printCarInfo` to an array object, for example, causes a runtime error and crashes the program. Arrays do not define that method. Only objects that implement a given method can respond to the message properly and execute the code that was requested.

```
2009-05-08 09:04:31.978 HelloWorld[419:20b] *** -[NSCFArray printCarInfo]:
unrecognized selector sent to instance 0xd14e80
2009-05-08 09:04:31.980 HelloWorld[419:20b] *** Terminating app due to uncaught
exception 'NSInvalidArgumentException', reason: '*** -[NSCFArray
```

```
printCarInfo]: unrecognized selector sent to instance 0xd14e80'
```

During compilation, Objective-C performs object message checks using static typing. The array definition in Figure 3-1 is declared statically, telling the compiler that the object in question is of type `(NSArray *)`. When the compiler finds objects that may not be able to respond to the requested methods, it issues warnings.

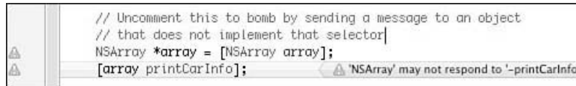


Figure 3-1 Xcode's Objective-C issues warnings when it finds a method that does not appear to be implemented by the receiver.

These warnings do not make the compilation fail, and it's possible that this code could run without error if `NSArray` implemented `printCarInfo` and did not declare that implementation in its published interface. Since `NSArray` does not, in fact, implement this method, running this code produces the actual runtime crash shown previously.

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

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

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

```
NSArray *anotherArray = [NSMutableArray array];
// This mutable-only method call is valid but
// produces a compile-time warning
[anotherArray addObject:@"Hello World"];
```

What produces a warning here is not the creation and assignment. It's the use. Sending `addObject:` to `anotherArray` uses our knowledge that the array is, in fact, mutable despite the fact that it is statically typed as `(NSArray *)`. That's something the compiler does not understand. This use generates a compile-time warning, namely:

```
'NSArray' may not respond to '-addObject:'
```

At runtime, however, the code works without error.

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

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

The code seen here produces just one warning, at the line where the standard array object is assigned to the mutable array pointer, namely "assignment from distinct Objective-C type." Parent-to-child assignments do not generate this warning. Child-to-parent assignments do. So do assignments between completely unrelated classes. Do not ignore this warning; fix your code. Otherwise, you're setting yourself up for a runtime crash. Because Objective-C is a compiled language that uses dynamic typing, it does not perform many of the runtime checks that interpreted object-oriented languages do.

Note

In Xcode, you can set the compiler to treat warnings as errors by checking Treat Warnings as Errors (a.k.a. GCC_TREAT_WARNINGS_AS_ERRORS) in Target Info > Build > GCC Warnings or by setting the `-Werror` flag in Project Info > Build > GCC Warnings > Other Warning Flags. Because Objective-C is so dynamic, the compiler cannot catch every problem that might crash at runtime the way static language compilers can. So pay special attention to warnings and try to eliminate them.

Inheriting Methods

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

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

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

Declaring Methods

As Listing 3-1 showed, a class interface defines the instance variables and methods that a new class adds to its parent class. This interface is normally placed into a header file, which is named with an .h extension. The interface from Listing 3-1 declared three methods, namely:

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

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

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

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

Implementing Methods

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

Listing 3-2 shows the implementation for the `Car` class example. It codes all three methods declared in the header file from Listing 3-1 and adds a fourth. This extra method redefines `init`. The `Car` version of `init` sets the `make` and `model` of the car to `nil`, which is the NULL pointer for Objective-C objects. It also initializes the `year` of the car to 1901.

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

Several things are notable about the `init` method seen here. First, the method returns a value, which is typed to `(id)`. As mentioned earlier in this chapter, the `id` type is more or less equivalent to `(NSObject *)`, although it's theoretically slightly more generic

than that. It can point to any object of any class (including Class objects themselves). You return results the same way you would in C, using `return`. The goal of `init` is to return a properly initialized version of the receiver via return `self`.

Second, the method calls `[super init]`. This tells Objective-C to send a message to a different implementation, namely the one defined in the object's superclass. The superclass of `Car` is `NSObject`, as shown in Listing 3-1. This call says, "Please perform the initialization that is normally done by my parent class before I add my custom behavior." Calling a superclass's implementation before adding new behavior demonstrates an important practice in Objective-C programming.

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

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

Note

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

Listing 3-2 The `Car` Class Implementation (Car.m)

```
#import "Car.h"

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

    // These initial nil assignments are not really needed.
    // All instance variables are initialized to zero by alloc.
    // Here is where you would perform any real assignments.
    make = nil;
    model = nil;
    year = 1901;
```

```
    return self;
}

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

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

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

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

Class Methods

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

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

Then you could code it up in your implementation:

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

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

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

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

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

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

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

Fast Enumeration

Fast enumeration was introduced in Objective-C 2.0 and offers a simple and elegant way to enumerate through collections such as arrays and sets. It adds a for-loop that iterates through the collection using concise `for/in` syntax. The enumeration is very efficient, running quickly. It is also safe. Attempts to modify the collection as it's being enumerated raise a runtime exception.

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

Note

Use caution when using methods such as arrayWithObjects: and `dictionaryWith-KeysAndValues:`, which are unnecessarily error-prone. Developers often use these methods with instance variables without first checking that these values are non-`nil`. Another common error is to forget the final "sentinel" `nil` at the end of your arguments. This missing `nil` will not be caught at compile time but will typically cause runtime errors.

Class Hierarchy

In Objective-C, each new class is derived from an already-existing class. The `Car` class described in Listings 3-1 and 3-2 is formed from `NSObject`, the root class of the Objective-C class tree. Each subclass adds or modifies state and behavior that it inherits from its parent, also called its "superclass." The `Car` class adds several instance variables and methods to the vanilla `NSObject` it inherits.

Figure 3-2 shows some of the classes found in the iOS SDK and how they relate to each other in the class hierarchy. Strings and arrays descend from `NSObject`, as does the `UIResponder` class. `UIResponder` is the ancestor of all onscreen iOS elements. Views, labels, text fields, and sliders are children, grandchildren, or other descendants of `UIResponder` and `NSObject`.

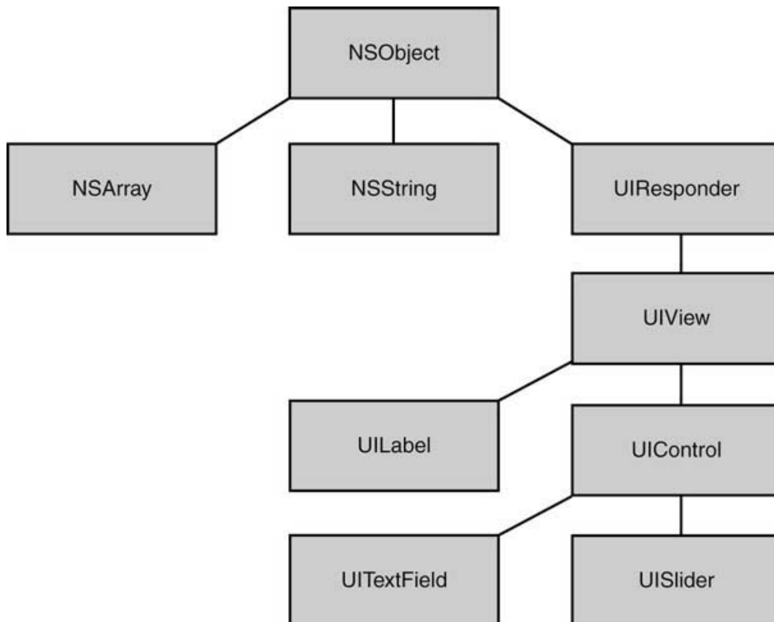


Figure 3-2 All Cocoa Touch classes are descended from `NSObject`, the root of the class hierarchy tree.

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

- Add new instance variables that are not allocated by their parent, also called the superclass. The Car class adds three: the make and model strings, and the year integer.
- Add new methods that are not defined by the parent. Car defines several new methods, letting you set the values of the instance variables and print out a report about the car.
- Override methods that the parents have already defined. The Car class's init method overrides NSObject's version. When sent an init message, a car object runs its version, not NSObject's. At the same time, the code for init makes sure to call NSObject's init method via [super init]. Referencing a parent's implementation, while extending that implementation, is a core part of the Objective-C design philosophy.

Logging Information

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

NSStrings are declared differently than C strings. They are prepended with the @ (at) symbol. A typical NSString looks @"like this"; the equivalent C string looks "like this" (omitting the @). Whereas C strings refer to a pointer to a string of bytes, NSStrings are objects. You can manipulate a C string by changing the values stored in each byte. NSStrings are immutable; you cannot access the bytes to edit them, and the actual string data is not stored within the object.

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

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

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

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

into

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

Table 3-1 shows some of the most common format specifiers. This is far from an exhaustive list, so consult Apple's String Programming Guide for Cocoa for more details.

Table 3-1 Common String Format Specifiers

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

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

```
2009-05-07 14:19:08.792 HelloWorld[11197:20b] Make: Ford
```

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

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

Another useful logging function is `CFShow()`. It takes one argument, an object, and prints out a snapshot description of that object to `stderr`:

```
CFShow(make);
```

Like `NSLog`, `CFShow` sends `description` to the objects it displays. Unlike `NSLog`, however, `CFShow` does not clutter your debugging console with timestamps, so it appeals to

anyone who prefers to skip that extra information. `CFShow` doesn't require format strings, which simplifies adding them to code, but they can only be used with objects. You cannot `CFShow` an integer or float.

You can combine `NSLog`'s object-formatting features with `printf`'s no-prefix presentation by implementing a simple NS-styled print utility such as the following function. This function automatically adds a carriage return, as `NSLog` does but `printf` does not. Adjust this accordingly for your own needs.

```
void nsprintf(NSString *formatString,...)
{
    va_list arglist;
    if (formatString)
    {
        va_start(arglist, formatString);
        {
            NSString *outstring = [[NSString alloc]
                initWithFormat:formatString arguments:arglist];
            fprintf(stderr, "%s\n", [outstring UTF8String]);
            [outstring release];
        }
        va_end(arglist);
    }
}
```

Properties

Properties expose class variables and methods to outside use through what are called "accessor methods" (that is, methods that access information). Using properties might sound redundant. After all, the class definition shown in Listing 3-1 already announces public methods. So why use properties? It turns out that there are advantages to using properties over hand-built methods, not the least of which are dot notation and memory management.

Dot Notation

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

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

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

rather than the more cumbersome

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

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

Properties and Memory Management

Properties can help simplify memory management. You can create properties that automatically retain instance variables for the lifetime of your objects and release them when you set those variables to `nil`. Setting a retained property ensures that memory will not be released until you say so. Of course, properties are not a magic bullet. They must be defined and used properly.

Assigning an object to a retained property means you're guaranteed that the objects will be available throughout the tenure of your ownership. Listing 3-2 did not retain its `make` and `model`. If those objects were released somewhere else in an application, the pointers to the memory that stored those objects would become invalid. At some point, the application might try to access that memory and crash. By retaining objects, you ensure that the memory pointers remain valid.

The `arrayWithObjects:` method normally returns an autoreleased object, whose memory is deallocated at the end of the event loop cycle. (See Chapter 1, "Introducing the iOS SDK," for details about autorelease pools. A deeper discussion about memory management follows later in this chapter.) Assigning the array to a retained property means that the array will stick around indefinitely. You retain the object, preventing its memory from being released until you are done using it.

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

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

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

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

Creating Properties

There are two basic styles of properties: read-write and read-only. Read-write properties, which are the default, let you modify the values you access; read-only properties do not. Use read-only properties for instance variables that you want to expose, without providing a way to change their values. They are particularly handy for properties that are generated by algorithm from several instance variables or from device sensors, such as a device orientation.

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

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

and then synthesize it in your class implementation with

```
@synthesize year;
```

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

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

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

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

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

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

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

```
CFShow(myCar.carInfo);
```

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

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

but not with

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

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

Creating Custom Getters and Setters

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

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

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

When building your own setters and getters, you might add some basic memory management. The following methods retain new items and release previous values:

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

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

Note

In the remote case that `newModel` is somehow a child of `model`, calling `[model release]` may free the memory of the new model. For that reason, a more complete setter method retains `newModel` before calling `[model release]`.

You could go even further by building more complicated routines that generate side effects upon assignment and retrieval. For example, you might keep a count of the number of times the value has been retrieved or changed, or send in-app notifications to other objects. The Objective-C compiler remains happy so long as it finds, for any property, a getter (typically named the same as the property name) and a setter (usually `setName:`, where `Name` is the name of the property). What's more, you can bypass any Objective-C naming conventions by specifying setter and getter names in the property declaration. This declaration creates a new Boolean property called `forSale` and declares a custom getter/setter pair. As always, you add any property declarations to the class interface:

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

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

```
@synthesize forSale;
```

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

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

Using this approach creates both the normal setter and getter via dot notation plus the two custom methods, `isForSale` and `setSalable:`. Oddly, while you can use dot notation to assign and retrieve `forSale`, you cannot use the equivalent methods, and you cannot use the customized setter in dot notation. Here is how the usage breaks down:

```
Car *myCar = [Car car];

// You can use the synthesized setter and getter of course
[myCar setSalable:YES];
printf("The car %s for sale\n",
    myCar.isForSale ? "is" : "is not");

// The normal getter and setter still work in dot notation
myCar.forSale = NO;
printf("The car %s for sale\n",
    myCar.forSale ? "is" : "is not");

// But not the method versions.
// These produce run-time errors
// [myCar setForSale:YES];
// printf("The car %s for sale\n",
//     [myCar forSale] ? "is" : "is not");

// You cannot use the customized setter via dot notation.
// This produces a compile-time error
// myCar.setSalable = YES;
```

Property Attributes

In addition to read-write and read-only attributes, you can specify whether a property is retained and/or atomic. The default behavior for properties is assign. Assignment acts exactly as if you'd assigned a value to an instance variable. There's no special retain/release behavior associated with the property, but by making it a property you expose the variable outside the class via dot notation. A property that's declared

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

uses the assign behavior.

Setting the property's attribute to `retain` does two things. First, it retains the passed object upon assignment. Second, it releases the previous value before a new assignment is made. Using the `retain` attribute introduces the memory management advantages discussed in the previous section. To create a retained property, add the attribute between parentheses in the declaration:

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

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

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

You can also retain the object as you assign it, as shown here:

```
myCar.make = @"Ford";
[myCar.make retain];
```

When you develop in a multithreaded environment, you want to use atomic methods. Xcode synthesizes atomic methods to automatically lock objects before they are accessed or modified and unlock them after. This ensures that setting or retrieving an object's value is performed fully regardless of concurrent threads. There is no atomic keyword. All methods are synthesized atomically by default. You can, however, state the opposite, allowing Objective-C to create accessors that are nonatomic:

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

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

Simple Memory Management

Memory management comes down to two simple rules. At creation, every object has a `retain` count of 0. At release, every object has (or, more accurately, is about to have) a `retain` count of 0. It is up to you as a developer to manage an object's retention over its lifetime. You should ensure that it moves from start to finish without being prematurely

released and guarantee that it does finally get released when it is time to do so. Complicating matters is Objective-C's autorelease pool. If some objects are autoreleased and others must be released manually, how do you best control your objects? Here's a quick-and-dirty guide to getting your memory management right.

Creating Objects

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

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

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

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

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

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

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

```
- (void) anotherProperMethod
{
    NSArray *array = [[[NSArray alloc] init] autorelease];
    // This won't crash the way release would
    printf("Retain count is %d\n", [array retainCount]);
    // use the array here

}
```

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

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

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

Creating Autoreleased Objects

As a rule, whenever you ask another method to create an object, it's good programming practice to return that object autoreleased. Doing so consistently lets you follow a simple rule: "If I didn't allocate it, then it was built and returned to me as an autorelease object."

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

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

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

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

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

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

```
+ (Car *) car
{
    return [[[Car alloc] init] autorelease];
}
```

Autorelease Object Lifetime

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

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

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

Note

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

Retaining Autorelease Objects

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

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

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

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

Retained Properties

Retained properties hold onto data that you assign to them and properly relinquish that data when you set a new value. Because of this, they tie in seamlessly to basic memory management. Here's how you create and use retained properties in your iOS applications.

First, declare your retained property in the class interface by including the `retain` keyword between parentheses:

```
@property (retain) NSArray *colors;
```

Then synthesize the property methods in your implementation:

```
@synthesize colors;
```

When given the `@synthesize` directive, Objective-C automatically builds routines that manage the retained property. The routines automatically retain an object when you assign it to the property. That behavior holds regardless of whether the object is set as autorelease. When you reassign the property, the previous value is automatically released.

Assigning Values to Retained Properties

When working with retained properties, you need to be aware of two patterns of assignment. These patterns depend on whether or not you're assigning an autorelease object. For autorelease-style objects, use a simple single assignment. This assignment sets the `colors` property to the new array and retains it:

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

The array is created and returned as an autorelease object with a count of +1. The assignment to the retained `colors` property brings the count to +2. Once the current event loop ends, the autorelease pool sends `release` to the array, and the count drops back to +1.

For normal (non-autorelease) objects, release the object after assigning it. Upon creation, the retain count for a normally allocated object is +1. Assigning the object to a retained property increases that count to +2. Releasing the object returns the count to +1.

```
// Non-autorelease object. Retain count is +1 at creation
NSArray *array = [[NSArray alloc]
    initWithObjects:@"Black", @"Silver", @"Gray", nil];

// Count rises to +2 via assignment to a retained property
myCar.colors = array;

// Now release to get that retain count back to +1
[array release];
```

You often see this pattern of create, assign, release in iOS development. You might use it when assigning a newly allocated view to a view controller object. Here's an example:

```
UIView *mainView = [[UIView alloc] initWithFrame:aFrame];
```

```
self.view = mainView;
[mainView release];
```

These three steps move the object's retain count from +1 to +2 and then back to +1.

A final count of +1 guarantees you that can use an object indefinitely. At the same time, you're assured that the object deallocates properly when the property is set to a new value and `release` is called on its prior value. That release on a +1 object allows the object to deallocate.

Reassigning a Retained Property

When you're done using a retained property, regardless of the approach used to create that object, set the property to `nil` or to another object. This sends a release message to the previously assigned object.

```
myCar.colors=nil;
```

If the `colors` property had been set to an array, as just shown, that array would automatically be sent a release message. Since each pattern of assignment produced a +1 retained object, this reassignment would send `release` to the +1 object. The object's life would be over.

Avoiding Assignment Pitfalls

Within a class implementation, it's handy to use properties to take advantage of this memory management behavior. To take advantage of this, avoid using instance variables directly. Direct assignment like this won't retain the array or release any previous value. This is a common pitfall for new iOS developers. Remember the dot notation when accessing the instance variables.

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

This same caution holds true for properties defined as `assign`. Note the following behavior carefully. Although both

```
@property NSArray *colors;
```

and

```
@property (assign) NSArray *colors;
```

allow you to use dot notation, assignment via these properties does not retain or release objects. Assign properties expose the `colors` instance variable to the outside world, but they do not provide the same memory management that retain properties do.

> **Note**
>
> As a general rule of thumb, Apple recommends you avoid using properties in your `init` methods. Instead, use instance variables directly.

High Retain Counts

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

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

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

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

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

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

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

Other Ways to Create Objects

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

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

C-Style Object Allocations

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

There are, however, no simple rules. As the following code shows, you may end up using `free()`, `CFRelease()`, and custom methods such as `CGContextRelease()` all in the same scope, side-by-side with standard Objective-C class convenience methods such as `imageWithCGImage:`. The function used to create the context object used here is `CGBitmapContextCreate()`, and like most Core Foundation function calls, it does not return an autoreleased object. This code snippet builds a `UIImage`, the iOS SDK class that stores image data.

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

Carbon and Core Foundation

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

Table 3-2 explains the key terms involved. To summarize the issue, early OS X used a C-based framework called Core Foundation to provide a transitional system for developing applications that could run on both Classic Mac systems as well as Mac OS X. Although Core Foundation uses object-oriented extensions to C, its functions and constructs are all C based, not Objective-C based.

Table 3-2 Key OS X Development Terms

Term	Definition
Foundation	The core classes for Objective-C programming, offering all the fundamental data types and services needed for Cocoa and Cocoa Touch. A section at the end of this chapter introduces some of the most important Foundation classes you'll use in your applications.

Term	Definition
Core Foundation	A library of C-based classes that are based on Foundation APIs but that are implemented in C. Core Foundation uses object-oriented data but is not built using the Objective-C classes.
Carbon	An early set of libraries provided by Apple that use a procedural API. Carbon offered event handling support, a graphics library, and many more frameworks. Some Carbon APIs live on through Core Foundation. Carbon was introduced for the Classic Mac OS, first appearing in Mac OS 8.1.
Cocoa	Apple's collection of frameworks, APIs, and runtimes that make up the modern Mac OS X runtime system. Frameworks are primarily written in Objective-C, although some continue to use C/C++.
Cocoa Touch	Cocoa's equivalent for the iOS SDK, where the frameworks are tuned for the touch-based mobile iOS user experience. Some iOS frameworks such as Core Audio and Open GL are considered to reside outside Cocoa Touch.
Toll Free Bridging	A method of Cocoa/Carbon integration. Toll Free Bridging refers to sets of interchangeable data types. For example, Cocoa's Foundation (`NSString *`) object can be used interchangeably with Carbon's Core Foundation's `CFStringRef`. Bridging connects the C-based Core Foundation with the Objective-C Foundation world.

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

Deallocating Objects

iPhone devices use reference count–managed Objective-C. On the iPhone, iPod touch, and iPad, there's no garbage collection and little likelihood there ever will be. Every object cleans up after itself. So what does that mean in practical terms? Here's a quick rundown of how you end an object's life, cleaning up its instance variables and preparing it for deallocation.

Instance variables must release retained objects before deallocation. You as the developer must ensure that those objects return to a retain count of 0 before the parent object is itself released. To do this, you implement `dealloc`, a method automatically called by the runtime system when an object is about to be released. If you use a class with object instance variables (that is, not just `float`s, `int`s, and `Bool`s), you probably need to implement a deallocation method. The basic `dealloc` method structure looks like this:

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

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

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

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

Retained Properties

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

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

Variables

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

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

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

This assignment style means that `salesman` could point to an object with a +1 retain count at any time during the object's lifetime. Therefore, in your `dealloc` method, you must release any object currently assigned to `salesman`. You can guard this with a check

if `salesman` is not `nil`, but practically, you're free to send `release` to `nil` without consequence, so feel free to skip the check.

```
[salesman release];
```

A Sample Deallocation Method

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

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

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

Cleaning Up Other Matters

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

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

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

Using Blocks

In Objective-C 2.0, blocks refer to a language construct that supports "closures," a way of treating code behavior as objects. First introduced to iOS in the 4.0 SDK, Apple's language extension makes it possible to create "anonymous" functionality, a small coding element that works like a method without having to be defined or named as a method.

This allows you to pass that code as parameters, providing an alternative to traditional callbacks. Instead of creating a separate "doStuffAfterTaskHasCompleted" delayed

execution method and using the method selector as a parameter, you can use blocks to pass that same behavior directly into your calls. This has two important benefits.

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

Second, blocks allow you to share lexical scope. Instead of explicitly passing local variable context in the form of callback parameters, blocks can implicitly read locally declared variables and parameters from the calling method or function. This context-sharing provides a simple and elegant way to specify the ways you need to clean up or otherwise finish a task without having to re-create that context elsewhere.

Defining Blocks in Your Code

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

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

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

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

Using blocks for simple enumeration works similarly to existing Objective-C 2.0 “for in” loops. What blocks give you further in the iOS SDK 4 APIs are two things. First is the ability to perform concurrent and/or reversed enumeration using the `enumerateObjectsAtIndexes:options:usingBlock:` method. This method extends array and set iteration into new and powerful areas. Second is dictionary support. Two methods (`enumerateKeysAndObjectsUsingBlock:` and `enumerateKeysAndObjectsWithOptions:usingBlock:`) provide dictionary-savvy block operations with direct access to

the current dictionary key and object, making dictionary iteration cleaner to read and more efficient to process.

Assigning Block References

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

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

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

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

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

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

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

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

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

    // Update the image on the main thread using the main queue
    [[NSOperationQueue mainQueue] addOperationWithBlock:^{
```

```
        UIImage *weatherImage = [UIImage imageWithData:imageData];
        imageView.image = weatherImage;}];
```

This code demonstrates how blocks can be nested as well as used at different stages of a task. Take note that any pending blocks submitted to a queue hold a reference to that queue, so the queue is not deallocated until all pending blocks have completed. That allows me to create an autoreleased queue in this example without worries that the queue will disappear during the operation of the block.

Blocks and Local Variables

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

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

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

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

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

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

Blocks and Memory Management

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

Other Uses for Blocks

In this section, you have seen how to define and apply blocks. In addition to the examples shown here, blocks play other roles in iOS development. Blocks can be used with notifications (see example in Chapter 1), animations, and as completion handlers. Many new iOS 4 APIs use blocks to simplify calls and coding. You can easily find many block-based APIs by searching the Xcode Document set for method selectors containing "block:" and "handler:".

Crafting Singletons

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

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

Building a singleton takes very little code. You define a static shared instance inside the class implementation and add a class method pointing to that instance. In this snippet, which is taken from the tagging example of Chapter 6, "Assembling Views and Animations," the instance is built the first time it is requested:

```
@implementation ViewIndexer
static ViewIndexer *sharedInstance = nil;

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

// Class behavior defined here

@end
```

To use this singleton, call `[ViewIndexer sharedInstance]`. This returns the shared object and lets you access any behavior that the singleton provides. For thread-safe use, you may want to use a more guarded approach to singleton creation, such as this one:

```
+ (ViewIndexer *)sharedInstance
{
    static dispatch_once_t pred;
```

```
    dispatch_once(&pred, ^{
        sharedInstance = [[self alloc] init];
    });
    return sharedInstance;
}
```

Categories (Extending Classes)

Objective-C's built-in capability to expand already-existing classes is one of its most powerful features. This behavioral expansion is called a "category." Categories extend class functionality without subclassing. You choose a descriptive expansion name, build a header, and then implement the functionality in a method file. Categories add methods to existing classes even if you did not define that class in the first place and do not have the source code for that class.

To build a category, you declare a new interface. Specify the category name (it's arbitrary) within parentheses, as you see here. List any new public methods and properties and save the header file. This Orientation category expands the `UIDevice` class, which is the SDK class responsible for reporting device characteristics, including orientation, battery level, and the proximity sensor state. This interface adds a single property to `UIDevice`, returning a read-only Boolean value. The new `isLandscape` property reports back whether the device is currently using a landscape orientation.

```
@interface UIDevice (Orientation)
@property (nonatomic, readonly) BOOL isLandscape;
@end
```

You cannot add new instance variables to a category interface as you could when subclassing. You are instead expanding a class's behavior, as shown in the source code of Listing 3-3. The code implements the landscape check by looking at the standard `UIDevice` orientation property.

You might use the new property like this:

```
NSLog(@"The device orientation is %@",
      [UIDevice currentDevice].isLandscape ? @"Landscape" : @"Portrait");
```

Here, the landscape orientation check integrates seamlessly into the SDK-provided `UIDevice` class via a property that did not exist prior to expanding the class. Just FYI, `UIKit` does offer a pair of device orientation macros (`UIDeviceOrientationIsPortrait` and `UIDeviceOrientationIsLandscape`), but you must pass these an orientation value, which you have to poll from the device.

Note

In addition to adding new behavior to existing classes, categories also let you group related methods into separate files for classes you build yourself. For large, complex classes, this helps increase maintainability and simplifies the management of individual source files.

Please note that when you add a category method that duplicates an existing method signature, the Objective-C runtime uses your implementation and overrides the original.

Listing 3-3 Building an Orientation Category for the `UIDevice` Class

```
@interface UIDevice (Orientation)
@property (nonatomic, readonly) BOOL isLandscape;
@end

@implementation UIDevice (Orientation)
- (BOOL) isLandscape
{
    return
        (self.orientation == UIDeviceOrientationLandscapeLeft) ||
        (self.orientation == UIDeviceOrientationLandscapeRight);
}
@end
```

Protocols

Chapter 1 introduced the notion of delegates. Delegates implement details that cannot be determined when a class is first defined. For example, a table knows how to display rows of cells, but it can't know what to do when a cell is tapped. The meaning of a tapped row changes with whatever application implements that table. A tap might open another screen, send a message to a web server, or perform any other imaginable result. Delegation lets the table communicate with a smart object that is responsible for handling those taps but whose behavior is written at a completely separate time from when the table class itself is created.

Delegation basically provides a language that mediates contact between an object and its handler. A table tells its delegate "I have been tapped," "I have scrolled," and other status messages. The delegate then decides how to respond to these messages, producing updates based on its particular application semantics.

Data sources operate the same way, but instead of mediating action responses, data sources provide data on demand. A table asks its data source, "What information should I put into cell 1 and cell 2?" The data source responds with the requested information. Like delegation, data sourcing lets the table place requests to an object that is built to understand those demands.

In Objective-C, both delegation and data sourcing are produced by a system called protocols. Protocols define *a priori* how one class can communicate with another. They contain a list of methods that are defined outside any class. Some of these methods are required. Others are optional. Any class that implements the required methods is said to conform to the protocol.

Defining a Protocol

Imagine, if you would, a jack-in-the-box toy. This is a small box with a handle. When you turn the crank, music plays. Sometimes a puppet (called the “jack”) jumps out of the box. Now imagine implementing that toy (or a rough approximation) in Objective-C. The toy provides one action, turning the crank, and there are two possible outcomes: the music or the jack.

Now consider designing a programmatic client for that toy. It could respond to the outcomes, perhaps, by gradually increasing a boredom count when more music plays or reacting with surprise when the jack finally bounces out. From an Objective-C point of view, your client needs to implement two responses: one for music, another for the jack. Here’s a client protocol you might build:

```
@protocol JackClient <NSObject>
- (void) musicDidPlay;
- (void) jackDidAppear;
@end
```

This protocol declares that to be a client of the toy, you must respond to music playing and the jack jumping out of the box. Listing these methods inside an `@protocol` container defines the protocol. All the methods listed here are required unless you specifically declare them as `@optional`, as you read about in the next sections.

Incorporating a Protocol

Next, imagine designing a class for the toy itself. It offers one action, turning the crank, and requires a second object that implements the protocol, in this case called `client`. This class interface specifies that the client needs to be some kind of object (`id`) that conforms to the `JackClient` protocol (`<JackClient>`). Beyond that, the class does not know at design time what kind of object will provide these services.

```
@interface JackInTheBox : NSObject
{
    id <JackClient> client;
}
- (void) turnTheCrank;
@property (retain)      id <JackClient> client;
@end
```

Adding Callbacks

Callbacks connect the toy class to its client. Since the client must conform to the `JackClient` protocol, you can send `jackDidAppear` and `musicDidPlay` messages to the object and they will compile without error. The protocol ensures that the client implements these methods. In this code, the callback method is selected randomly. The music plays approximately nine out of every ten calls, sending `musicDidPlay` to the client.

```
- (void) turnTheCrank
```

```
{
    // You need a client to respond to the crank
    if (!self.client) return;

    // Randomly respond to the crank turn
    int action = random() % 10;
    if (action < 1)
        [self.client jackDidAppear];
    else
        [self.client musicDidPlay];
}
```

Declaring Optional Callbacks

Protocols include two kinds of callbacks: required and optional. By default, callbacks are required. A class that conforms to the protocol must implement those methods or they produce a compiler warning. You can use the `@required` and `@optional` keywords to declare a protocol method to be of one form or the other. Any methods listed after an `@required` keyword are required; after an `@optional` keyword, they are optional. Your protocol can grow complex accordingly.

```
@protocol JackClient <NSObject>
- (void) musicDidPlay; // required
@required
- (void) jackDidAppear; // also required
@optional
- (void) nothingDidHappen; // optional
@end
```

In practice, using more than a single `@optional` keyword is overkill. The same protocol can be declared more simply. When you don't use any optional items, skip the keyword entirely. Notice the `<NSObject>` declaration here. It's required to effectively implement optional protocols. It says that a `JackClient` object conforms to and will be a kind of `NSObject`.

```
@protocol JackClient <NSObject>
- (void) musicDidPlay;
- (void) jackDidAppear;
@optional
- (void) nothingDidHappen;
@end
```

Implementing Optional Callbacks

Optional methods let the client choose whether to implement a given protocol method. They reduce the implementation burden on whoever writes that client but add a little extra work to the class that hosts the protocol definition. When you are unsure whether a class does or does not implement a method, you must test before you send a message. Fortunately, Objective-C and the `NSObject` class make it easy to do so:

```
// optional client method
if ([self.client  respondsToSelector: @selector(nothingDidHappen)])
```

```
    [self.client nothingDidHappen];
```

`NSObject` provides a `respondsToSelector:` method, which returns a Boolean `YES` if the object implements the method or `NO` otherwise. By declaring the client with `<NSObject>`, you tell the compiler that the client can handle this method, which allows you to check the client for conformance before sending the message.

Conforming to a Protocol

Classes include protocol conformance in interface declarations. A view controller that implements the `JackClient` protocol announces it between angle brackets. A class might conform to several protocols. Combine these within the brackets, separating protocol names with commas:

```
@interface TestBedViewController :
    UIViewController <JackClient>
{
    JackInTheBox *jack;
}
@property (retain) JackInTheBox *jack;
@end
```

Declaring the `JackClient` protocol lets you assign the host's `client` property. The following code compiles without error because the class for `self` was declared in conformance with `JackClient`:

```
self.jack = [JackInTheBox jack];
jack.client = self;
```

Had you omitted the protocol declaration in your interface, this assignment would produce an error at compile time.

Once you include that protocol between the angle brackets, you must implement all required methods in your class. Omitting any of them produces the kind of compile-time warnings shown in Figure 3-3. The compiler tells you which method is missing and what protocol that method belongs to.

Figure 3-3 You must implement all required methods to conform to a protocol. Objective-C warns about incomplete implementations.

The majority of protocol methods in the iOS SDK are optional. Both required and optional methods are detailed exhaustively in the developer documentation. Note that protocols are documented separately from the classes they support. For example, Xcode documentation provides three distinct `UITableView` reference pages: one for the

`UITableView` class, one for the `UITableViewDelegate` protocol, and another for the `UITableViewDataSource` protocol.

Foundation Classes

If you're new to Objective-C, there are a few key classes you absolutely need to be familiar with before moving forward. These include strings, numbers, and collections, and they provide critical application building blocks. The `NSString` class, for example, provides the workhorse for nearly all text manipulation in Objective-C. However, it, like other fundamental classes, is not defined in Objective-C itself. It is part of the Foundation framework, which offers nearly all the core utility classes you use on a day-to-day basis.

Foundation provides over a dozen kinds of object families and hundreds of object classes. These range from value objects that store numbers and dates, to strings that store character data; from collections that store other objects, to classes that access the file system and retrieve data from URLs. Foundation is often referred to (slightly inaccurately) as Cocoa. (Cocoa and its iPhone-device-family equivalent Cocoa Touch actually include all the frameworks for OS X programming.) To master Foundation is to master Objective-C programming, and thorough coverage of the subject demands an entire book of its own.

As this section cannot offer an exhaustive introduction to Foundation classes, you're about to be introduced to a quick-and-dirty survival overview. Here are the classes you need to know about and the absolutely rock-core ways to get started using them. You find extensive code snippets that showcase each of the classes to give you a jumping-off point if, admittedly, not a mastery of the classes involved.

Strings

Cocoa strings store character data, just as their cousins the `(char *)` C strings do. They are, however, objects and not byte arrays. Unlike C, the core `NSString` class is immutable in Cocoa. That is, you can use strings to build other strings, but you can't edit the strings you already own. String constants are delineated by quote marks and the `@` character. Here is a typical string constant, which is assigned to a string variable:

```
NSString *myString = @"A string constant";
```

Building Strings

You can build strings using formats, much as you would using `sprintf`. If you're comfortable creating `printf` statements, your knowledge transfers directly to string formats. Use the `%@` format specifier to include objects in your strings. String format specifiers are thoroughly documented in the Cocoa String Programming Guide, available via Xcode's documentation window (Command-Option-?). The most common formats are listed in Table 3-1.

```
NSString *myString = [NSString stringWithFormat:
```

```
    @"The number is %d", 5];
```

To create new strings, you can append strings together. This call outputs “The number is 522” and creates a new instance built from other strings:

```
NSLog(@"%@", [myString stringByAppendingString:@"22"]);
```

Appending formats provides even more flexibility. You specify the format string and the components that build up the result:

```
NSLog(@"%@", [myString stringByAppendingFormat:@"%d", 22]);
```

> **Note**
>
> Allow custom objects to respond to `%@` delimiters by implementing the description method. Simply return an `NSString` object that describes your object’s state.

Length and Indexed Characters

Every string can report its length (via length) and produce an indexed character on demand (via `characterAtIndex:`). The two calls shown here output 15 and e, respectively, based on the previous `@"The number is 5"` string. Cocoa characters use the `unichar` type, which store Unicode-style characters.

```
NSLog(@"%d", myString.length);
printf("%c", [myString characterAtIndex:2]);
```

Converting to and from C Strings

The realities of normal C programming often crop up despite working in Objective-C. Being able to move back and forth between C strings and Cocoa strings is an important skill. Convert an `NSString` to a C string either by sending `UTF8String` or `cStringUsingEncoding:`. These are equivalent, producing the same C-based bytes:

```
printf("%s\n", [myString UTF8String]);
printf("%s\n", [myString cStringUsingEncoding: NSUTF8StringEncoding]);
```

You can also go the other way and transform a C string into an `NSString` by using `stringWithCString:` encoding. The examples here use UTF-8 encoding, but Objective-C supports a large range of options, including ASCII, Japanese, Latin, Windows-CP1251, and so forth.

```
NSLog(@"%@", [NSString stringWithCString:"Hello World"
    encoding: NSUTF8StringEncoding]);
```

Writing Strings to and Reading Strings from Files

Writing to and reading strings from the local file system offers a handy way to save and retrieve data. This snippet shows how to write a string to a file:

```
NSString *myString = @"Hello World";
NSError *error;
NSString *path = [NSHomeDirectory()
    stringByAppendingPathComponent:@"Documents/file.txt"];
```

```
if (![myString writeToFile:path atomically:YES
    encoding:NSUTF8StringEncoding error:&error])
{
    NSLog(@"Error writing to file: %@", [error localizedDescription]);
    return;
}
NSLog(@"String successfully written to file");
```

The path for the file is `NSHomeDirectory()`, a function that returns a string with a path pointing to the application sandbox. Notice the special `append` method that properly appends the Documents/file.txt subpath.

In Cocoa, most file access routines offer an atomic option. When you set the `atomically` parameter to `YES`, the iOS SDK writes the file to a temporary auxiliary and then renames it into place. Using an atomic write ensures that the file avoids corruption.

The request shown here returns a Boolean, namely `YES` if the string was written or `NO` if it was not. Should the write request fail, this snippet logs the error using a language-localized description. It uses an instance of the `NSError` class to store that error information and sends the `localizedDescription` selector to convert the information into a human-readable form. Whenever iOS methods return errors, use this approach to determine which error was generated.

Reading a string from a file follows a similar form but does not return the same Boolean result. Instead, check to see whether the returned string is `nil`, and if so display the error that was returned:

```
NSString *inString = [NSString stringWithContentsOfFile:path
    encoding:NSUTF8StringEncoding error:&error];
if (!inString)
{
    NSLog(@"Error reading from file % %@", [path lastPathComponent],
        [error localizedDescription]);
    return;
}
NSLog(@"String successfully read from file");
NSLog(@"%@", inString);
```

Accessing Substrings

Cocoa offers a number of ways to extract substrings from strings. Here's a quick review of some typical approaches. As you'd expect, string manipulation is a large part of any flexible API, and Cocoa offers many more routines and classes to parse and interpret strings than the few listed here. This quick `NSString` summary skips any discussion of `NSScanner`, `NSXMLParser`, and so forth.

Converting Strings to Arrays

You can convert a string into an array by separating its components across some repeated boundary. This example chops the string into individual words by splitting around spaces. The spaces are discarded, leaving an array that contains each number word.

```
NSString *myString = @"One Two Three Four Five Six Seven";
```

```
NSArray *wordArray = [myString componentsSeparatedByString: @" "];
NSLog(@"%@", wordArray);
```

Requesting Indexed Substrings

You can request a substring from the start of a string to a particular index, or from an index to the end of the string. These two examples return `@"One Two"` and `@"Two Three Four Five Six Seven"`, respectively, using the to and from versions of the indexed substring request. As with standard C, array and string indexes start at 0.

```
NSString *sub1 = [myString substringToIndex:7];
NSLog(@"%@", sub1);

NSString *sub2 = [myString substringFromIndex:4];
NSLog(@"%@", sub2);
```

Generating Substrings from Ranges

Ranges let you specify exactly where your substring should start and stop. This snippet returns `@"Tw"`, starting at character 4 and extending two characters in length. `NSRange` provides a structure that defines a section within a series. You use ranges with indexed items such as strings and arrays.

```
NSRange r;
r.location = 4;
r.length = 2;
NSString *sub3 = [myString substringWithRange:r];
NSLog(@"%@", sub3);
```

Search and Replace with Strings

With Cocoa, you can easily search a string for a substring. Searches return a range, which contain both a location and a length. Always check the range location. The location `NSNotFound` means the search failed. This returns a range location of 18, with a length of 4.

```
NSRange searchRange = [myString rangeOfString:@"Five"];
if (searchRange.location != NSNotFound)
    NSLog(@"Range location: %d, length: %d", searchRange.location, search-
Range.length);
```

Once you've found a range, you can replace a subrange with a new string. The replacement string does not need to be the same length as the original; thus, the result string may be longer or shorter than the string you started with:

```
NSLog(@"%@", [myString stringByReplacingCharactersInRange:
    searchRange withString: @"New String"]);
```

A more general approach lets you replace all occurrences of a given string. This snippet produces `@"One * Two * Three * Four * Five * Six * Seven"` by swapping out each space for a space-asterisk-space pattern:

```
NSString *replaced = [myString stringByReplacingOccurrencesOfString:
```

```
    @" " withString: @" * "];
NSLog(@"%@", replaced);
```

Changing Case

Cocoa provides three simple methods that change a string's case. Here, these three examples produce a string all in uppercase, all in lowercase, and one where every word is capitalized ("Hello World. How Do You Do?"). Because Cocoa supports case-insensitive comparisons, you rarely need to apply case conversions when testing strings against each other.

```
NSString *myString = @"Hello world. How do you do?";
NSLog(@"%@",[myString uppercaseString]);
NSLog(@"%@",[myString lowercaseString]);
NSLog(@"%@",[myString capitalizedString]);
```

Testing Strings

The iOS SDK offers many ways to compare and test strings. The three simplest check for string equality and match against the string prefix (the characters that start the string) and suffix (those that end it). More complex comparisons use `NSComparisonResult` constants to indicate how items are ordered compared with each other.

```
NSString *s1 = @"Hello World";
NSString *s2 = @"Hello Mom";
NSLog(@"%@ %@ %@", s1, [s1 isEqualToString:s2] ?
    @"equals" : @"differs from", s2);
NSLog(@"%@ %@ %@", s1, [s1 hasPrefix:@"Hello"] ?
    @"starts with" : @"does not start with", @"Hello");
NSLog(@"%@ %@ %@", s1, [s1 hasSuffix:@"Hello"] ?
    @"ends with" : @"does not end with", @"Hello");
```

Extracting Numbers from Strings

Convert strings into numbers by using a `Value` method. These examples return 3, 1, 3.141592, and 3.141592, respectively.

```
NSString *s1 = @"3.141592";
NSLog(@"%d", [s1 intValue]);
NSLog(@"%d", [s1 boolValue]);
NSLog(@"%f", [s1 floatValue]);
NSLog(@"%f", [s1 doubleValue]);
```

Mutable Strings

The `NSMutableString` class is a subclass of `NSString`. It offers you a way to work with strings whose contents can be modified. Once it is instantiated, you can append new contents to the string, allowing you to grow results before returning from a method. This example displays "Hello World. The results are in now."

```
NSMutableString *myString = [NSMutableString stringWithString:
    @"Hello World. "];
[myString appendFormat:@"The results are %@ now.", @"in"];
NSLog(@"%@", myString);
```

Numbers and Dates

Foundation offers a large family of value classes. Among these are numbers and dates. Unlike standard C floats, integers, and so forth, these elements are all objects. They can be allocated and released as well as used in collections such as arrays, dictionaries, and sets. The following examples show numbers and dates in action, providing a basic overview of these classes.

Working with Numbers

The `NSNumber` class lets you treat numbers as objects. You can create new `NSNumber` instances using a variety of convenience methods, namely `numberWithInt:`, `numberWithFloat:`, `numberWithBool:`, and so forth. Once they are set, you extract those values via `intValue`, `floatValue`, `boolValue`, and so on, and use normal C-based math to perform your calculations.

You are not limited to extracting the same data type an object was set with. You can set a float and extract the integer value, for example. Numbers can also convert themselves into strings:

```
NSNumber *number = [NSNumber numberWithFloat:3.141592];
NSLog(@"%d", [number intValue]);
NSLog(@"%@", [number stringValue]);
```

One of the biggest reasons for using `NSNumber` objects rather than `int`s, `float`s, and so forth, is that they are objects. You can use them with Cocoa routines and classes. For example, you cannot set a user default (that is, a preference value) to, say, the integer 23, as in "You have used this program 23 times." You can, however, store an object `[NSNumber numberWithInt:23]` and later recover the integer value from that object to produce the same user message.

If, for some reason, you must stick with C-style variables or byte arrays, consider using `NSValue`. The `NSValue` class allows you to encapsulate C data, including `int`, `float`, `char`, structures, and pointers into an Objective-C wrapper.

> **Note**
>
> The `NSDecimalNumber` class provides a handy object-oriented wrapper for base-10 arithmetic.

Working with Dates

As with standard C and `time()`, `NSDate` objects use the number of seconds since an epoch, which is a standardized universal time reference, to represent the current date. The iOS SDK epoch was at midnight on January 1, 2001. The standard UNIX epoch took place at midnight on January 1, 1970.

Each `NSTimeInterval` represents a span of time in seconds, stored with subsecond floating-point precision. The following code shows how to create a new date object using the current time and how to use an interval to reference some time in the future (or past):

```
// current time
NSDate *date = [NSDate date];

// time 10 seconds from now
date = [NSDate dateWithTimeIntervalSinceNow:10.0f];
```

You can compare dates by setting or checking the time interval between them. This snippet forces the application to sleep until 5 seconds into the future and then compares the date to the one stored in `date`:

```
// Sleep 5 seconds and check the time interval
[NSThread sleepUntilDate:[NSDate dateWithTimeIntervalSinceNow:5.0f]];
NSLog(@"Slept %f seconds", [[NSDate date] timeIntervalSinceDate:date]);
```

The standard description method for dates returns a somewhat human-readable string, showing the current date and time:

```
// Show the date
NSLog(@"%@" [date description]);
```

To convert dates into fully formatted strings rather than just using the default description, use an instance of `NSDateFormatter`. You specify the format (for example, YY for two-digit years, and YYYY for four-digit years) using the object's date format property. A full list of format specifiers is offered in the built-in Xcode documentation. In addition to producing formatted output, this class can also be used to read preformatted dates from strings, although that is left as an exercise for the reader.

```
// Produce a formatted string representing the current date
NSDateFormatter *formatter = [[[NSDateFormatter alloc] init]
    autorelease];
formatter.dateFormat = @"MM/dd/YY HH:mm:ss";
NSString *timestamp = [formatter stringFromDate:[NSDate date]];
NSLog(@"%@", timestamp);
```

> **Note**
>
> See the `NSDate` category at http://github.com/erica for examples of using common date scenarios (such as yesterday, tomorrow, and so forth). Chapter 11, "Creating and Managing Table Views," contains a handy table of date formatter codes.

Timers

When working with time, you may need to request that some action occur in the future. Cocoa offers an easy-to-use timer that triggers at an interval you specify; use the `NSTimer` class. The timer shown here triggers after 1 second and repeats until the timer is disabled:

```
[NSTimer scheduledTimerWithTimeInterval: 1.0f target: self
    selector: @selector(handleTimer:) userInfo: nil repeats: YES];
```

Each time the timer activates, it calls its target sending the selector message it was initialized with. The callback method takes one argument (notice the single colon), which is

the timer itself. To disable a timer, send it the invalidate message; this releases the timer object and removes it from the current run loop.

```
- (void) handleTimer: (NSTimer *) timer
{
    printf("Timer count: %d\n", count++);
    if (count > 3)
    {
        [timer invalidate];
        printf("Timer disabled\n");
    }
}
```

Recovering Information from Index Paths

The `NSIndexPath` class is used with iOS tables. It stores the section and row number for a user selection—that is, when a user taps on the table. When provided with index paths, you can recover these numbers via the `myIndexPath.row` and `myIndexPath.section` properties. Learn more about this class and its use in Chapter 11.

Collections

The iOS SDK primarily uses three kinds of collections: arrays, dictionaries, and sets. Arrays act like C arrays. They provide an indexed list of objects, which you can recover by specifying which index to look at. Dictionaries, in contrast, store values that you can look up by keys. For example, you might store a dictionary of ages, where Dad's age is the `NSNumber` 57 and a child's age is the `NSNumber` 15. Sets offer an unordered group of objects and are usually used in the iOS SDK in connection with recovering user touches from the screen. Each of these classes offers regular and mutable versions, just as the `NSString` class does.

Building and Accessing Arrays

Create arrays using the `arrayWithObjects:` convenience method, which returns an autoreleased array. When calling this method, list any objects you want added to the array and finish the list with `nil`. (If you do not include `nil` in your list, you'll experience a runtime crash.) You can add any kind of object to an array, including other arrays and dictionaries. This example showcases the creation of a three-item array:

```
NSArray *array = [NSArray arrayWithObjects:@"One", @"Two", @"Three", nil];
```

The `count` property returns the number of objects in an array. Arrays are indexed starting with 0, up to one less than the count. Attempting to access `[array objectAtIndex:array.count]` causes an "index beyond bounds" exception and crashes. So always use care when retrieving objects, making sure not to cross either the upper or lower boundary for the array.

```
NSLog(@"%d", array.count);
NSLog(@"%@", [array objectAtIndex:0]);
```

Mutable arrays are editable. The mutable form of `NSArray` is `NSMutableArray`. With mutable arrays, you can add and remove objects at will. This snippet copies the previous array into a new mutable one and then edits the array by adding one object and removing another one. This returns an array of `[@"One", @"Two", @"Four]`.

```
NSMutableArray *marray = [NSMutableArray arrayWithArray:array];
[marray addObject:@"Four"];
[marray removeObjectAtIndex:2];
NSLog(@"%@", marray);
```

Whether or not you're working with mutable arrays, you can always combine arrays to form a new version containing the components from each. No checks are done about duplicates. This code produces a six-item array, including one, two, and three from the original array, and one, two, and four from the mutable array:

```
NSLog(@"%@", [array arrayByAddingObjectsFromArray:marray]);
```

Checking Arrays

You can test whether an array contains an object and recover the index of a given object. This code searches for the first occurrence of "Four" and returns the index for that object. The test in the `if` statement ensures that at least one occurrence exists:

```
if ([marray containsObject:@"Four"])
      NSLog(@"The index is %d",
        [marray indexOfObject:@"Four"]);
```

Converting Arrays into Strings

As with other objects, sending a description to an array returns an `NSString` that describes an array. In addition, you can use `componentsJoinedByString` to transform an `NSArray` into a string. The following code returns `@"One Two Three"`:

```
NSArray *array = [NSArray arrayWithObjects:@"One", @"Two", @"Three", nil];
NSLog(@"%@", [array componentsJoinedByString:@" "]);
```

Building and Accessing Dictionaries

`NSDictionary` objects store keys and values, enabling you to look up objects using strings. The mutable version of dictionaries, `NSMutableDictionary`, lets you modify these dictionaries by adding and removing elements on demand. In iOS programming, you use the mutable class more often the static one, so these examples showcase mutable versions.

Creating Dictionaries

Use the dictionary convenience method to create a new mutable dictionary, as shown here. This returns a new initialized dictionary that you can start to edit. Populate the dictionary using `setObject:forKey:`.

```
NSMutableDictionary *dict = [NSMutableDictionary dictionary];
[dict setObject:@"1" forKey:@"A"];
[dict setObject:@"2" forKey:@"B"];
[dict setObject:@"3" forKey:@"C"];
```

```
NSLog(@"%@", [dict description]);
```

Searching Dictionaries

Searching the dictionary means querying the dictionary by key name. Use `objectForKey:` to find the object that matches a given key. When a key is not found, the dictionary returns `nil`. This returns `@"1"` and `nil`:

```
NSLog(@"%@", [dict objectForKey:@"A"]);
NSLog(@"%@", [dict objectForKey:@"F"]);
```

Replacing Objects

When you set a new object for the same key, Cocoa replaces the original object in the dictionary. This code replaces `"3"` with `"foo"` for the key `"C"`:

```
[dict setObject:@"foo" forKey:@"C"];
NSLog(@"%@", [dict objectForKey:@"C"]);
```

Removing Objects

You can also remove objects from dictionaries. This snippet removes the object associated with the `"B"` key. Once removed, both the key and the object no longer appear in the dictionary.

```
[dict removeObjectForKey:@"B"];
```

Listing Keys

Dictionaries can report the number of entries they store plus they can provide an array of all the keys currently in use. This key list lets you know what keys have already been used. It lets you test against the list before adding an item to the dictionary, avoiding overwriting an existing key/object pair.

```
NSLog(@"The dictionary has %d objects", [dict count]);
NSLog(@"%@", [dict allKeys]);
```

Accessing Set Objects

Sets store unordered collections of objects. You encounter sets when working with the iPhone device family's multitouch screen. The `UIView` class receives finger movement updates that deliver touches as an `NSSet`. To work with touches, you typically issue `allObjects` and work with the array that gets returned. Once converted, use standard array calls to list, query, and iterate through the touches. You can also use fast enumeration directly with sets.

Memory Management with Collections

Arrays, sets, and dictionaries automatically retain objects when they are added and release those objects when they are removed from the collection. Releases are also sent when the collection is deallocated. Collections do not copy objects. Instead, they rely on retain counts to hold onto objects and use them as needed.

Writing Out Collections to File

Both arrays and dictionaries can store themselves into files using `writeToFile:-atomically:` methods so long as the types within the collections belong to the set of `NSData`, `NSDate`, `NSNumber`, `NSString`, `NSArray`, and `NSDictionary`. Pass the path as the first argument and a Boolean as the second. As when saving strings, the second argument determines whether the file is first stored to a temporary auxiliary and then renamed. The method returns a Boolean value: `YES` if the file was saved, `NO` if not. Storing arrays and dictionaries create standard property lists files.

```
NSString *path = [NSHomeDirectory()
    stringByAppendingPathComponent:@"Documents/ArraySample.txt"];
if ([array writeToFile:path atomically:YES])
    NSLog(@"File was written successfully");
```

To recover an array or dictionary from file, use the convenience methods `arrayWith-ContentsOfFile:` and `dictionaryWithContentsOfFile:`. If the methods return `nil`, the file could not be read.

```
NSArray *newArray = [NSArray arrayWithContentsOfFile:path];
NSLog(@"%@", newArray);
```

Building URLs

`NSURL` objects point to resources. These resources can refer to both local files and to URLs on the Web. Create `url` objects by passing a string to class convenience functions. Separate functions have been set up to interpret each kind of URL. Once built, however, `NSURL` objects are interchangeable. Cocoa does not care if the resource is local or points to an object only available via the Net. This code demonstrates building URLs of each type, path and Web:

```
NSString *path = [NSHomeDirectory()
    stringByAppendingPathComponent:@"Documents/foo.txt"];
NSURL *url1 = [NSURL fileURLWithPath:path];
NSLog(@"%@", url1);

NSString *urlpath = @"http://ericasadun.com";
NSURL *url2 = [NSURL URLWithString:urlpath];
NSLog(@"%d characters read",
    [[NSString stringWithContentsOfURL:url2] length]);
```

Working with `NSData`

If `NSString` objects are analogous to zero-terminated C strings, then `NSData` objects correspond to buffers. `NSData` provides data objects that store and manage bytes. Often, you fill `NSData` with the contents of a file or URL. The data returned can report its length, letting you know how many bytes were retrieved. This snippet retrieves the contents of a URL and prints the number of bytes that were read:

```
NSData *data = [NSData dataWithContentsOfURL:url2];
NSLog(@"%d", [data length]);
```

To access the core byte buffer that underlies an `NSData` object, use bytes. This returns a `(const void *)` pointer to actual data.

As with many other Cocoa objects, you can use the standard `NSData` version of the class or its mutable child, `NSMutableData`. Most Cocoa programs that access the Web, particularly those that perform asynchronous downloads, pull in a bit of data at a time. For those cases, `NSMutableData` objects prove useful. You can keep growing mutable data by issuing `appendData:` to add the new information as it is received.

File Management

The iOS SDK's file manager is a singleton provided by the `NSFileManager` class. It can list the contents of folders to determine what files are found and perform basic file system tasks. The following snippet retrieves a file list from two folders. First, it looks in the sandbox's Documents folder and then inside the application bundle itself.

```
NSFileManager *fm = [NSFileManager defaultManager];

// List the files in the sandbox Documents folder
NSString *path = [NSHomeDirectory() stringByAppendingPathComponent:@"Documents"];
NSLog(@"%@",[fm directoryContentsAtPath:path]);
// List the files in the application bundle
path = [[NSBundle mainBundle] bundlePath];
NSLog(@"%@",[fm directoryContentsAtPath:path]);
```

Note the use here of `NSBundle`. It lets you find the application bundle and pass its path to the file manager. You can also use `NSBundle` to retrieve the path for any item included in your app bundle. (You cannot, however, write to the application bundle at any time.) This code returns the path to the application's Default.png image. Note that the file and extension names are separated and that each is case sensitive.

```
NSBundle *mb = [NSBundle mainBundle];
NSLog(@"%@", [mb pathForResource:@"Default" ofType:@"png"]);
```

The file manager offers a full suite of file-specific management. It can move, copy, and remove files as well as query the system for file traits and ownership. Here are some examples of the simpler routines you may use in your applications:

```
// Create a file
NSString *docspath = [NSHomeDirectory()
    stringByAppendingPathComponent:@"Documents"];
NSString *filepath = [NSHomeDirectory()
    stringByAppendingPathComponent:@"Documents/testfile"];
NSArray *array = [@"One Two Three" componentsSeparatedByString:@" "];
[array writeToFile:filepath atomically:YES];
NSLog(@"%@", [fm directoryContentsAtPath:docspath]);

// Copy the file
NSString *copypath = [NSHomeDirectory()
    stringByAppendingPathComponent:@"Documents/copied"];
if (![fm copyItemAtPath:filepath toPath:copypath error:&error])
{
    NSLog(@"Copy Error: %@", [error localizedDescription]);
    return;
```

```
}
NSLog(@"%@", [fm directoryContentsAtPathdocspath]);

// Move the file
NSString *newpath = [NSHomeDirectory()
    stringByAppendingPathComponent:@"Documents/renamed"];
if (![fm moveItemAtPath:filepath toPath:newpath error:&error])
{
    NSLog(@"Move Error: %@", [error localizedDescription]);
    return;
}
NSLog(@"%@", [fm directoryContentsAtPath:docspath]);

// Remove a file
if (![fm removeItemAtPath:copypath error:&error])
{
    NSLog(@"Remove Error: %@", [error localizedDescription]);
    return;
}
NSLog(@"%@", [fm directoryContentsAtPath:docspath]);
```

Note

As another convenient file trick, use tildes in path names (for example, ~/Library/Preferences/foo.plist) and apply the `NSString` method `stringByExpandingTildeInPath`.

One More Thing: Message Forwarding

Although Objective-C does not provide true multiple-inheritance, it offers a workaround that lets objects respond to messages that are implemented in other classes. If you want your object to respond to another class's messages, you can add message forwarding to your applications and gain access to that object's methods.

Normally, sending an unrecognized message produces a runtime error, causing an application to crash. But before the crash happens, iOS's runtime system gives each object a second chance to handle a message. Catching that message lets you redirect it to an object that understands and can respond to that message.

Consider the `Car` example used throughout this chapter. The `carInfo` property introduced midway through the examples returns a string that describes the car's make, model, and year. Now imagine if a `Car` instance could respond to `NSString` messages by passing them to that property. Send `length` to a `Car` object and instead of crashing, the object would return the length of the `carInfo` string. Send `stringByAppendingString:` and the object adds that string to the property string. It would be as if the `Car` class inherited (or at least borrowed) the complete suite of string behavior.

Objective-C provides this functionality through a process called "message forwarding." When you send a message to an object that cannot handle that selector, the selector gets forwarded to a `forwardInvocation:` method. The object sent with this message—namely an `NSInvocation` instance—stores the original selector and arguments that were

requested. You can override `forwardInvocation:` and send that message on to another object.

Implementing Message Forwarding

To add message forwarding to your program, you must override two methods—namely, `methodSignatureForSelector:` and `forwardInvocation:`. The former creates a valid method signature for messages implemented by another class. The latter forwards the selector to an object that actually implements that message.

Building a Method Signature

This first method returns a method signature for the requested selector. For this example, a `Car` instance cannot properly create a signature for a selector implemented by another class (in this case, `NSString`). Adding a check for a malformed signature (that is, returning `nil`) gives this method the opportunity to iterate through each pseudo-inheritance and attempt to build a valid result. This example draws methods from just one other class via `self.carInfo`:

```
- (NSMethodSignature*) methodSignatureForSelector:(SEL)selector
{
    // Check if car can handle the message
    NSMethodSignature* signature = [super
        methodSignatureForSelector:selector];

    // If not, can the car info string handle the message?
    if (!signature)
        signature = [self.carInfo methodSignatureForSelector:selector];

    return signature;
}
```

Forwarding

The second method you need to override is `forwardInvocation:`. This method only gets called when an object has been unable to handle a message. This method gives the object a second chance, allowing it to redirect that message. The method checks to see whether the `self.carInfo` string responds to the selector. If it does respond, it tells the invocation to invoke itself using that object as its receiver.

```
- (void)forwardInvocation:(NSInvocation *)invocation
{
    SEL selector = [invocation selector];

    if ([self.carInfo respondsToSelector:selector])
    {
        printf("[forwarding from %s to %s] ", [[[self class] description]
            UTF8String], [[NSString description] UTF8String]);
        [invocation invokeWithTarget:self.carInfo];
    }
}
```

Using Forwarded Messages

Calling nonclass messages such as `UTF8String` and `length` produces compile-time warnings, which you can ignore. The code shown in Figure 3-4 causes two compiler warnings. The code, however, compiles and (more importantly) runs without error. As the figure shows, you can send `Car` instance methods that are defined by the class itself and also those implemented by `NSString`.

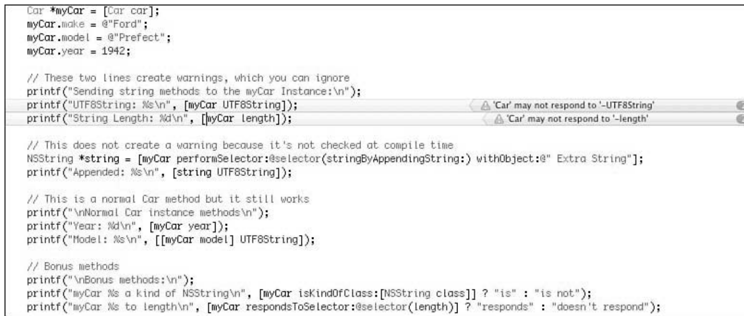


Figure 3-4 The compiler issues warnings for forwarded methods, but the code runs without error.

House Cleaning

Although invocation forwarding mimics multiple inheritance, `NSObject` never confuses the two. Methods such as `respondsToSelector:` and `isKindOfClass:` only look at the inheritance hierarchy and not at the forwarding change.

A couple of optional methods allow your class to better express its message compliance to other classes. Reimplementing `respondsToSelector:` and `isKindOfClass:` lets other classes query your class. In return, the class announces that it responds to all string methods (in addition to its own) and that it is a "kind of" string, further emphasizing the pseudo-multiple inheritance approach.

```
// Extend selector compliance
- (BOOL)respondsToSelector:(SEL)aSelector
{
    // Car class can handle the message
    if ( [super respondsToSelector:aSelector] )
        return YES;

    // CarInfo string can handle the message
    if ([self.carInfo respondsToSelector:aSelector])
        return YES;

    // Otherwise...
    return NO;
}
```

```
// Allow posing as class
- (BOOL)isKindOfClass:(Class)aClass
{
    // Check for Car
    if (aClass == [Car class]) return YES;
    if ([super isKindOfClass:aClass]) return YES;

    // Check for NSString
    if ([self.carInfo isKindOfClass:aClass]) return YES;

    return NO;
}
```

Super-easy Forwarding

The method signature/forward invocation pair of methods provides a robust and approved way to add forwarding to your classes. A simpler approach is also available on iOS 4.0 and later devices. You can replace both those methods with this single one, which does all the same work with less coding and less operational expense. According to Apple, this approach is "an order of magnitude faster than regular forwarding."

```
- (id)forwardingTargetForSelector:(SEL)sel
{
    if ([self.carInfo respondsToSelector:sel]) return self.carInfo;
    return nil;
}
```

Summary

This chapter provided an abridged, high-octane introduction to Objective-C and Foundation. In it, you read about the way that Objective-C extends C and provides support for object-oriented programming. You discovered properties and memory management and were subjected to a speedy review of the most important Foundation classes. So what can you take away from this chapter? Here are a few final thoughts:

- The sample code for this chapter contains all the examples used throughout this introduction. Try testing this material directly in Xcode. Mess around with the material, add your own examples, or expand the ones you've been given. A hands-on approach offers the best way to gain critical skills you need for iOS development.
- Learning Objective-C and Cocoa takes more than just a chapter. If you're serious about learning iOS programming, and these concepts are new to you, consider seeking out single-topic books that are dedicated to introducing these technologies to developers new to the platform. Consider Aaron Hillegass's *Cocoa Programming for Mac OS X, 3rd Edition*, or Stephen Kochan's *Programming in Objective-C 2.0, 2nd Edition*, or Fritz Anderson's *Xcode 3 Unleashed*. Apple has an excellent Objective-C 2.0 overview at http://developer.apple.com/Mac/library/documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html.

- This chapter mentioned Core Foundation and Carbon but did not delve into these technologies in any depth. You can and will experience C-based APIs in the iOS SDK, particularly when you work with the address book, with Quartz 2D graphics, and with Core Audio, among other frameworks. Each of these specific topic areas are documented exhaustively at Apple's developer website, complete with sample code. A strong grounding in C (and sometimes C++) programming will help you work through the specific implementation details.