O'REILLY®

Covers Swift 1.2, iOS 8.3, Xcode 6.3

Programming iOS 8

DIVE DEEP INTO VIEWS, VIEW CONTROLLERS, AND FRAMEWORKS

Matt Neuburg

Programming iOS 8

Matt Neuburg

Preface

Aut lego vel scribo; doceo scrutorve sophian.

—Sedulius Scottus

On June 2, 2014, Apple's WWDC keynote address ended with a shocking announcement: "We have a new programming language." This was surprising in several ways:

- Apple buried the lede (or, looking at it another way, they saved the biggest until last).
- Like the weather, everyone had long talked about the need for a new language to replace Objective-C, but no one believed Apple would ever actually *do* anything about it.
- How on earth had Apple done all the groundwork needed to design, prepare, and implement a whole new programming language without the least rumor leaking out?

Having picked themselves up off the floor, developers immediately began to examine this new language — Swift — studying it, critiquing it, and deciding whether to adopt it. My own first move was to translate all my existing iOS apps into Swift; this was enough to convince me that, for all its faults, Swift deserved to be adopted by new students of iOS programming, and that my books, therefore, should henceforth assume that readers are using Swift.

Therefore, Swift is the programming language used throughout this book. Nevertheless, the reader will also need some awareness of Objective-C (including C). There are two chief reasons for this:

- The Foundation and Cocoa APIs, the built-in commands with which your code must interact in order to make anything happen on an iOS device, are still written in C and Objective-C. In order to interact with them, you have to know what those languages would expect. For example, in order to pass a Swift array where an NSArray is expected, you need to know what consitutes an object acceptable as an element of an Objective-C NSArray.
- Swift can't do everything that C and Objective-C can do. Apple likes to claim that Swift can "access all of the Cocoa Touch frameworks" and that it constitutes "a complete replacement for both the C and Objective-C languages" — I'm quoting from the Xcode 6 release notes — but the truth is that Swift can't interface directly with every facet of the Cocoa Touch APIs. Swift can't form a C function or obtain the address of such a function; Swift can't declare a property `@dynami` in the Objective-C sense; in certain situations, Swift can be prohibitively slow in comparison to equivalent Objective-C code; and so on. Thus, I occasionally show some Objective-C code in this edition, in order to do things that Swift alone can't accomplish.

Because Swift is new, I have had to settle on my own conventions for presenting and describing code. The most important question has been how to state the name of a method. My solution is to give *the method's Objective-C name*. As an example, what is the name of the method that you call when you set a value using key–value coding? The actual call, in Swift, would be something like this:

```
someObject.setValue(someValue, forKey:"someKey")
```

But when I *give the name* of the method being called here, that name is `setValue:forKey:`. This choice has several advantages:

- It is clear, compact, and predictable, being formed according to perfectly consistent and well-established rules.
- For those who have been programming in Objective-C all these years, it's what we're used to. The way Objective-C methods are named is a thoroughly entrenched convention; even Swift programmers must accommodate themselves to seeing Objective-C method names plastered all over the Internet and Apple's own documentation.
- That name *is* in fact its name as far as Objective-C and the Cocoa APIs are concerned, and even in Swift you would need to know this. For example, if you wanted to specify this method when providing its name as a selector parameter (or as a string representing a selector), it is the Objective-C name that you would have to provide.
- The translation from the Objective-C name into practical Swift syntax is mechanical and unambiguous — the first colon is replaced by parentheses embracing the first parameter plus the names and values of any remaining parameters — and is something that every Swift programmer needs to know how to do.

Similarly, I will speak of the `setValue:` parameter, even though, in an actual Swift method call, the name of that parameter appears before the parentheses, with no colon.

The grand exception to this rule is my treatment of initializers. Here, the difference between Objective-C and Swift is greater than a simple shorthand can readily encompass, so I give the initializer's name *as you would implement it*. For example, in Objective-C, the default initializer for a UIView is `initWithFrame:`. In this book, I will call that initializer `init(frame:)`. The same thing applies to what in Objective-C would be class factory methods with a corresponding initializer: when I want to mention what in Objective-C would be the UIImage class factory method `imageNamed:`, I'll call it `init(named:)`.

This treatment of initializer names has the great virtue that the name clarifies instantly that such a method *is* an initializer. Swift effectively abolishes class factory methods and brings such methods into the fold of formal initializers; my treatment of their names is a way of supporting and adopting that innovation.

Again, this convention requires some mental translation on the reader's part, because in Swift the way you would name an initializer when you *implement* it is not the same as the way you would name it when you *call* it. If you were to override this method in a UIView subclass, you'd override `init(frame:)`; but if you were to *call* it to create a new UIView, you'd say `UIView(frame:)`. This mental translation, however, is hardly objectionable, as it is a practical fact to which every Swift user is already habituated.

So much for Swift. A new language is big news, but Swift is not the only big news connected with iOS 8. Nor is it the only welcome news. I've been arguing for years that Apple needs to check the rampan

growth and evolution of the basic Cocoa Touch APIs and concentrate, for one release at least, on *rationalizing* the changes that have accumulated, often seemingly by random accretion, over the year iOS 8 appears to do exactly that. To give just a couple of obvious examples:

- An app, when it rotates to compensate for a change in the orientation of the device, no longer does so by applying a transform to the root view. Instead, in iOS 8, rotation is a feature of the app *as a whole* — the window itself, and indeed the screen, is resized.

- A popover, in iOS 8, is just a variety of presented view controller. Not only does this vastly simplify popover management (whose wretchedness I've been complaining about since the first edition of this book), but also it eliminates the need, in a universal app, for large chunks of conditional code depending on what type of device the app runs on: a presented view controller is legal everywhere. Similarly, alerts and action sheets are now forms of presented view controller.

Such rationalization permeates iOS 8, and I have eagerly adopted it, even where the prior approach is not yet deprecated. You *could* still create popovers and alerts the way we created them in iOS 7 and before; and obviously if you wanted your code to run on iOS 7 as well as iOS 8, you'd have to. But in this edition I behave, in effect, as if the old way didn't exist. It's the business of this edition to teach iOS 8, not to help you write backwards-compatible code; if you want to know how to deal with popovers (or anything else) in iOS 7 or before, consult an earlier edition of the book.

This book retains the structure of the previous edition. Like Homer's *Iliad*, it begins in the middle of the story, with the reader jumping with all four feet into views and view controllers, and with a knowledge of the language and the Xcode IDE already presupposed. As in *Programming iOS 7*, discussion of the programming language, the Xcode IDE (including the nature of nibs, outlets, and actions, and the mechanics of nib loading), and the fundamental conventions, classes, and architectures of the Cocoa Touch framework (including delegation, the responder chain, key–value coding, memory management, and so on), has been expunged and relegated to a different book. This material that constituted Chapters 1–13 of earlier editions, before *Programming iOS 7*, but whose presence was deemed to be making this book unwieldy in size and scope.

Here's a summary of this book's major sections:

- Part I describes *views*, the fundamental units of an iOS app's interface. Views are what the user ca see and touch in an iOS app. To make something appear before the user's eyes, you need a view. T let the user interact with your app, you need a view. This part of the book explains how views are created, arranged, drawn, layered, animated, and touched.

- Part II starts by discussing *view controllers*. Perhaps the most remarkable and important feature o iOS programming, view controllers enable views to come and go coherently within the interface, thus allowing a single-windowed app running on what may be a tiny screen to contain multiple screens of material. This part of the book talks about all the ways in which view controllers can be manipulated in order to make their views appear. It also describes *every kind of view* provided by the Cocoa framework — the built-in building blocks with which you'll construct an app's interfac

- Part III surveys the most important secondary *frameworks* provided by iOS. These are clumps of code, sometimes with built-in interface, that are not part of your app by default, but are there for the asking if you need them, allowing you to work with such things as sound, video, user libraries,

mail, maps, and the device's sensors.

- Part IV wraps up the book with some miscellaneous but important topics: files, networking, threading, and how to implement Undo.

- Appendix A summarizes the most important lifetime event messages sent to your app delegate.

- Appendix B, new in this edition, appends some useful Swift utility functions. You should keep an eye on this appendix, consulting it whenever a mysterious method name appears. For instance, my example code frequently uses my `delay` function, which embraces `dispatch_after` with a convenient shorthand; when I use `delay` and you don't know what it is, consult this appendix.

Someone who has read this book will, I believe, be capable of writing a real-life iOS app, and to do s with a clear understanding of what he or she is doing and where the app is going as it grows and develops. The book itself doesn't show how to write any particularly interesting iOS apps (though it i backed by dozens of example projects that you can download from my GitHub site, http://github.com/mattneub/Programming-iOS-Book-Examples), but it does constantly use my own real apps and real programming situations to illustrate and motivate its explanations.

Just as important, this book is intended to prepare you for your own further explorations. In the case some topics, especially in Parts III and IV, I guide you past the initial barrier of no knowledge into an understanding of the topic, its concepts, its capabilities, and its documentation, along with some code examples; but the topic itself may be so huge that there is room only to introduce it here. Your feet, nevertheless, will now be set firmly on the path, and you will know enough that you can now proceed further down that path on your own whenever the need or interest arises.

Indeed, there is *always* more to learn about iOS. iOS is vast! It is all too easy to find areas of iOS tha have had to be ruled outside the scope of this book. In Part IV, for example, I peek at Core Data, and demonstrate its use in code, but a true study of Core Data would require an entire book of its own (an such books exist); so, having opened the door, I quickly close it again, lest this book suddenly double in size. By the same token, many areas of iOS are not treated at all in this book:

OpenGL
: An open source C library for drawing, including 3D drawing, that takes full advantage of graphics hardware. This is often the most efficient way to draw, especially when animation is involved. iO incorporates a simplified version of OpenGL called OpenGL ES. See Apple's *OpenGL Programming Guide for iOS*. Open GL interface configuration, texture loading, shading, and calculation are simplified by the GLKit framework; see the *GLKit Framework Reference*. New in iOS 8 are the Metal classes, which you'll also want to investigate.

Sprite Kit
: Sprite Kit provides a built-in framework for designing 2D animated games.

Scene Kit
: New in iOS 8, Scene Kit is ported from OS X, making it much easier to create 3D games and interactive graphics.

Accelerate

Certain computation-intensive processes will benefit from the vector-based Accelerate framewor See the *vDSP Programming Guide*.

Game Kit

The Game Kit framework covers three areas that can enhance your user's game experience: Wireless or Bluetooth communication directly between devices (peer-to-peer); voice communication across an existing network connection; and Game Center, which facilitates these and many other aspects of interplayer communication, such as posting and viewing high scores and setting up competitions. See the *Game Kit Programming Guide*.

Advertising

The iAD framework lets your free app attempt to make money by displaying advertisements provided by Apple. See the *iAD Programming Guide*.

Newsstand

Your app may represent a subscription to something like a newspaper or magazine. See the *Newsstand Kit Framework Reference*.

Printing

See the "Printing" chapter of the *Drawing and Printing Guide for iOS*.

Security

This book does not discuss security topics such as keychains, certificates, and encryption. See the *Security Overview* and the Security framework.

Accessibility

VoiceOver assists visually impaired users by describing the interface aloud. To participate, views must be configured to describe themselves usefully. Built-in views already do this to a large extent, and you can extend this functionality. See the *Accessibility Programming Guide for iOS*.

Telephone

The Core Telephony framework lets your app get information about a particular cellular carrier and call.

Pass Kit

The Pass Kit framework allows creation of downloadable passes to go into the user's Passbook app. See the *Passbook Programming Guide*.

Health Kit

New in iOS 8, the Health Kit framework lets your app obtain, store, share, and present data and statistics related to body activity and exercise. See the *HealthKit Framework Reference*.

External accessories

The user can attach an external accessory to the device, either directly via USB or wirelessly via Bluetooth. Your app can communicate with such an accessory. See *External Accessory Programming Topics*. New in iOS 8, the Home Kit framework lets the user communicate with devices in the physical world, such as light switches and door locks. See the *HomeKit Framework Reference*.

Handoff

New in iOS 8, Handoff permits your app to post to the user's iCloud account a record of what the user is doing, so that the user can switch to another copy of your app on another device and resum doing the same thing. See the *Handoff Programming Guide*.

Versions

This book was originally geared to iOS 8.1 and Xcode 6.1, both of which became publicly available i October, 2014. Subsequently, it was updated for iOS 8.3 and Swift 1.2, which is part of Xcode 6.3, released in April, 2015. Because of changes in the Swift language and in the Swift version of the Cocoa APIs, most of the code now will *not* compile in any version of Xcode earlier than 6.3!

In general, only very minimal attention is given to earlier versions of iOS and Xcode. As I've already said, it is not my intention to embrace in this book any detailed knowledge about earlier versions of the software, which is, after all, readily and compendiously available in my earlier books. The book does contain, nevertheless, a few words of advice about backward compatibility, and now and then I will call out a particularly noteworthy change from earlier system versions, especially where your existing iOS 7 code is likely to break or behave differently when compiled against iOS 8.

Xcode 6 has eliminated some of the templates that you choose from when creating a new project. The loss of the Utility Application template is a pity, because it embodied and illustrated the standard techniques for passing data to and from a presented view controller; but it hasn't affected this book, because the template itself had become so ugly and crufty that in the previous edition I didn't mentic it in any case. The loss of the Empty Application template, on the other hand, is severe; it is, after all perfectly reasonable to write an app without a storyboard (several of my own apps are structured in that way). Accordingly, near the start of the first chapter, I have given instructions for turning a Sing View Application–based template into something similar to what the Empty Application template would have given you. Also, although I treat storyboards as the primary Interface Builder design milieu, I still do also assume that you know how to make and work with a *.xib* file when that is what you need or prefer.

At the time of this writing, Apple was still making frequent adjustments to the Swift language and to the way the Objective-C APIs are bridged to it. I have tried to keep my code up-to-date, but please make allowances, and be prepared to compensate, for the possibility that my examples may contain slight occasional impedance mismatches, such as the lack of a needed Optional unwrap or the presence of a superfluous one.

Screenshots of Xcode were originally taken using Xcode 6.1 under OS X 10.9 Mavericks, and have *n* been updated to Xcode 6.3, which runs only under OS X 10.10 and later. This means that your

interface will naturally look slightly different from the screenshots, but this difference will be minimal and shouldn't cause any confusion.

Acknowledgments

My thanks go first and foremost to the people at O'Reilly Media who have made writing a book so delightfully easy: Rachel Roumeliotis, Sarah Schneider, Kristen Brown, Dan Fauxsmith, and Adam Witwer come particularly to mind. And let's not forget my first and long-standing editor, Brian Jepson, who had nothing whatever to do with this edition, but whose influence is present throughout.

As in the past, I have been greatly aided by some fantastic software, whose excellences I have appreciated at every moment of the process of writing this book. I should like to mention, in particular:

- git (http://git-scm.com)
- SourceTree (http://www.sourcetreeapp.com)
- TextMate (http://macromates.com)
- AsciiDoc (http://www.methods.co.nz/asciidoc)
- BBEdit (http://barebones.com/products/bbedit/)
- Snapz Pro X (http://www.ambrosiasw.com)
- GraphicConverter (http://www.lemkesoft.com)
- OmniGraffle (http://www.omnigroup.com)

The book was typed and edited entirely on my faithful Unicomp Model M keyboard (http://pckeyboard.com), without which I could never have done so much writing over so long a perio so painlessly. For more about my physical work environment, see http://matt.neuburg.usesthis.com.

From the Programming iOS 4 Preface

A programming framework has a kind of personality, an overall flavor that provides an insight into the goals and mindset of those who created it. When I first encountered Cocoa Touch, my assessment of its personality was: "Wow, the people who wrote this are really clever!" On the one hand, the number of built-in interface objects was severely and deliberately limited; on the other hand, the power and flexibility of some of those objects, especially such things as UITableView, was greatly enhanced over their OS X counterparts. Even more important, Apple created a particularly brilliant way (UIViewController) to help the programmer make entire blocks of interface come and go and supplant one another in a controlled, hierarchical manner, thus allowing that tiny iPhone display to unfold virtually into multiple interface worlds within a single app without the user becoming lost or confused.

The popularity of the iPhone, with its largely free or very inexpensive apps, and the subsequent popularity of the iPad, have brought and will continue to bring into the fold many new programmers who see programming for these devices as worthwhile and doable, even though they may not have fe the same way about OS X. Apple's own annual WWDC developer conventions have reflected this trend, with their emphasis shifted from OS X to iOS instruction.

The widespread eagerness to program iOS, however, though delightful on the one hand, has also fostered a certain tendency to try to run without first learning to walk. iOS gives the programmer mighty powers that can seem as limitless as imagination itself, but it also has fundamentals. I often see questions online from programmers who are evidently deep into the creation of some interesting app, but who are stymied in a way that reveals quite clearly that they are unfamiliar with the basics o the very world in which they are so happily cavorting.

It is this state of affairs that has motivated me to write this book, which is intended to ground the reader in the fundamentals of iOS. I love Cocoa and have long wished to write about it, but it is iOS and its popularity that has given me a proximate excuse to do so. Here I have attempted to marshal and expound, in what I hope is a pedagogically helpful and instructive yet ruthlessly Euclidean and logical order, the principles and elements on which sound iOS programming rests. My hope, as with my previous books, is that you will both read this book cover to cover (learning something new often enough to keep you turning the pages) and keep it by you as a handy reference.

This book is not intended to disparage Apple's own documentation and example projects. They are wonderful resources and have become more wonderful as time goes on. I have depended heavily on them in the preparation of this book. But I also find that they don't fulfill the same function as a reasoned, ordered presentation of the facts. The online documentation must make assumptions as to how much you already know; it can't guarantee that you'll approach it in a given order. And online documentation is more suitable to reference than to instruction. A fully written example, no matter how well commented, is difficult to follow; it demonstrates, but it does not teach.

A book, on the other hand, has numbered chapters and sequential pages; I can assume you know view before you know view controllers for the simple reason that Part I precedes Part II. And along with facts, I also bring to the table a degree of experience, which I try to communicate to you. Throughout this book you'll find me referring to "common beginner mistakes"; in most cases, these are mistakes that I have made myself, in addition to seeing others make them. I try to tell you what the pitfalls are because I assume that, in the course of things, you will otherwise fall into them just as naturally as I did as I was learning. You'll also see me construct many examples piece by piece or extract and explain just one tiny portion of a larger app. It is not a massive finished program that teaches programming, but an exposition of the thought process that developed that program. It is this thought process, more than anything else, that I hope you will gain from reading this book.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at https://github.com/mattneub/Programming-iOS-Book-Examples.

This book is here to help you get your job done. In general, if example code is offered with this book you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Programming iOS 8* by Matt Neuburg (O'Reilly). Copyright 201 Matt Neuburg, 978-1-491-90873-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at http://bit.ly/programmingiOS8_5E.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at http://www.oreilly.com.

Find us on Facebook: http://facebook.com/oreilly

Follow us on Twitter: http://twitter.com/oreillymedia

Watch us on YouTube: http://www.youtube.com/oreillymedia

Part I. Views

The things that appear in your app's interface are, ultimately, *views*. A view is a unit of your app that knows how to draw itself. A view also knows how to sense that the user has touched it. Views are wh your user sees on the screen, and what your user interacts with by touching the screen. Thus, views ar the primary constituent of an app's visible, touchable manifestation. They *are* your app's interface. S it's going to be crucial to know how views work.

- Chapter 1 discusses views in their most general aspect — their hierarchy, visibility, and position, including an explanation of autolayout.
- Chapter 2 is about drawing. A view knows how to draw itself; this chapter explains how to tell a view what you want it to draw, from displaying an existing image to constructing a drawing in code.
- Chapter 3 explains about layers. The drawing power of a view comes ultimately from its layer. To put it another way, a layer is effectively the aspect of a view that knows how to draw — with even more power.
- Chapter 4 tells about animation. An iOS app's interface isn't generally static; it's lively. Much of that liveliness comes from animation. iOS gives you great power to animate your interface with remarkable ease; that power, it turns out, resides ultimately in layers.
- Chapter 5 is about touches. A view knows how to sense that the user is touching it. This chapter explains the iOS view-based mechanisms for sensing and responding to touches, with details on how touches are routed to the appropriate view and how you can customize that routing.

Chapter 1. Views

A *view* (an object whose class is UIView or a subclass of UIView) knows how to draw itself into a rectangular area of the interface. Your app has a visible interface thanks to views. Creating and configuring a view can be extremely simple: “Set it and forget it.” For example, you can drag an interface object, such as a UIButton, into a view in the nib editor; when the app runs, the button appears, and works properly. But you can also manipulate views in powerful ways, in real time. Your code can do some or all of the view’s drawing of itself (Chapter 2); it can make the view appear and disappear, move, resize itself, and display many other physical changes, possibly with animation (Chapter 4).

A view is also a responder (UIView is a subclass of UIResponder). This means that a view is subject to user interactions, such as taps and swipes. Thus, views are the basis not only of the interface that the user sees, but also of the interface that the user touches (Chapter 5). Organizing your views so tha the correct view reacts to a given touch allows you to allocate your code neatly and efficiently.

The *view hierarchy* is the chief mode of view organization. A view can have subviews; a subview has exactly one immediate superview. Thus there is a tree of views. This hierarchy allows views to come and go together. If a view is removed from the interface, its subviews are removed; if a view is hidde (made invisible), its subviews are hidden; if a view is moved, its subviews move with it; and other changes in a view are likewise shared with its subviews. The view hierarchy is also the basis of, though it is not identical to, the responder chain.

A view may come from a nib, or you can create it in code. On balance, neither approach is to be preferred over the other; it depends on your needs and inclinations and on the overall architecture of your app.

The Window

The top of the view hierarchy is the app’s window. It is an instance of UIWindow (or your own subclass thereof), which is a UIView subclass. Your app should have *exactly one main window*. It is created at launch time and is never destroyed or replaced. It occupies the entire screen and forms the background to, and is the ultimate superview of, all your other visible views. Other views are visible by virtue of being subviews, at some depth, of your app’s window.

NOTE

If your app can display views on an external screen, you'll create an additional UIWindow to contain those views; but in this chapter I'll behave as if there were just one screen, the device's own screen, and just one window. I repeat: if there is just one screen, your app should create just one UIWindow. You may encounter online tutorials that advise explicit creation of a second UIWindow as a way of making content appear in front of the app's main interface; *such statements are wrong* and should be disregarded. To make content appear in front of the interface, add a view, not another entire window.

The window must fill the device's screen. Therefore, its size and position must be identical to the siz and position of the screen. This is done by setting the window's frame to the screen's bounds as the window is instantiated. (I'll explain later in this chapter what "frame" and "bounds" are.) If you're using a main storyboard, that's taken care of for you automatically behind the scenes by the `UIApplicationMain` function as the app launches; but an app without a main storyboard is possible, and in that case you'd need to set the window's frame yourself, very early in the app's lifetime, like this:

```
let w = UIWindow(frame: UIScreen.mainScreen().bounds)
```

The window must also persist for the lifetime of the app. To make this happen, the app delegate class has a `window` property with a strong retain policy. As the app launches, the `UIApplicationMain` function instantiates the app delegate class and retains the resulting instance. This is the app delegate instance; it is never released, so it persists for the lifetime of the app. The window instance is then assigned to the app delegate instance's `window` property; therefore it, too, persists for the lifetime of the app.

You will typically not put any view content manually and directly inside your main window. Instead, you'll obtain a view controller and assign it to the main window's `rootViewController` property. Once again, if you're using a main storyboard, this is done automatically behind the scenes; the view controller in question will be your storyboard's initial view controller.

When a view controller becomes the main window's `rootViewController`, its main view (its `view`) made the one and only immediate subview of your main window — the main window's *root view*. Al other views in your main window will be subviews of the root view. Thus, the root view is the highes object in the view hierarchy that the user will usually see. There might be just a chance, under certain circumstances, that the user will catch a glimpse of the window, behind the root view; for this reason you may want to assign the main window a reasonable `backgroundColor`. But this seems unlikely, and in general you'll have no reason to change anything about the window itself.

Your app's interface is not visible until the window, which contains it, is made the app's key window This is done by calling the UIWindow instance method `makeKeyAndVisible`.

Let's summarize how all the initial creation, configuration, and showing of the main window happen There are two cases to consider:

App with a main storyboard

If your app has a main storyboard, as specified by its *Info.plist* key "Main storyboard file base

name" (`UIMainStoryboardFile`) — the default for all Xcode 6 app templates — then `UIApplicationMain` instantiates UIWindow, sets its frame correctly, and assigns that instance to the app delegate's `window` property. It also instantiates the storyboard's initial view controller an assigns that instance to the window's `rootViewController` property. All of that happens *before* the app delegate's `application:didFinishLaunchingWithOptions:` is called (Appendix A).

Finally, `UIApplicationMain` calls `makeKeyAndVisible` to display your app's interface. This in turn automatically causes the root view controller to obtain its main view (typically by loading it from a nib), which the window adds as its own root view. That happens *after* `application:didFinishLaunchingWithOptions:` is called.

App without a main storyboard

If your app has no main storyboard, then creation and configuration of the window must be done some other way. Typically, it is done in code. No Xcode 6 app template lacks a main storyboard, but if you start with, say, the Single View Application template, you can experiment as follows:

1. Edit the target. In the General pane, select "Main" in the Main Interface field and delete it (and press Tab to set this change).
2. Delete *Main.storyboard* and *ViewController.swift* from the project.
3. Delete the entire content of *AppDelegate.swift*.

You now have a project with an app target but no storyboard and no code. To make a minimal working app, you need to edit *AppDelegate.swift* in such a way as to recreate the AppDelegate class with just enough code to create and show the window, as shown in Example 1-1. The result a minimal working app with an empty white window; you can prove to yourself that your code is creating the window by changing its `backgroundColor` to something else (such as `UIColor.redColor()`) and running the app again. We didn't set a root view controller, so you wi also see a warning about that in the console ("Application windows are expected to have a root view controller at the end of application launch"); I'll explain in a moment what to do about that.

Example 1-1. An App Delegate class with no storyboard

```
import UIKit
@UIApplicationMain
class AppDelegate : UIResponder, UIApplicationDelegate {
    var window : UIWindow?
    func application(application: UIApplication,
        didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
        -> Bool {
            self.window = UIWindow(frame:UIScreen.mainScreen().bounds)
            self.window!.backgroundColor = UIColor.whiteColor()
            self.window!.makeKeyAndVisible()
            return true
    }
}
```

It is extremely improbable that you would ever need to subclass UIWindow. If, however, you wanted

to create a UIWindow subclass and make an instance of that subclass your app's main window, then how you proceed depends on how the window is instantiated in the first place:

App with a main storyboard

As the app launches, after `UIApplicationMain` has instantiated the app delegate, it asks the app delegate instance for the value of its `window` property. If that value is nil, `UIApplicationMain` instantiates UIWindow and assigns that instance to the app delegate's `window` property. If that value is *not* nil, `UIApplicationMain` leaves it alone and uses it as the app's main window. Therefore, to make your app's main window be an instance of your UIWindow subclass, you'll make that instance the default value for the app delegate's `window` property, like this:

```
lazy var window : UIWindow? = {
    return MyWindow(frame: UIScreen.mainScreen().bounds)
}()
```

App without a main storyboard

You're already instantiating UIWindow and assigning that instance to the app delegate's `self.window` property, in code (Example 1-1). So instantiate your UIWindow subclass instead:

```
// ...
self.window = MyWindow(frame:UIScreen.mainScreen().bounds)
// ...
```

Once the app is running, there are various ways to refer to the window:

- If a UIView is in the interface, it automatically has a reference to the window through its own `window` property.

  You can also use a UIView's `window` property as a way of asking whether it is ultimately embedde in the window; if it isn't, its `window` property is nil. A UIView whose `window` property is nil canno be visible to the user.

- The app delegate instance maintains a reference to the window through its `window` property. You can get a reference to the app delegate from elsewhere through the shared application's `delegate` property, and through it you can refer to the window:

  ```
  let w = UIApplication.sharedApplication().delegate!.window!!
  ```

  If you prefer something less generic (and requiring less extreme unwrapping of Optionals), cast th `delegate` explicitly to your app delegate class:

  ```
  let w = (UIApplication.sharedApplication().delegate as! AppDelegate).window!
  ```

- The shared application maintains a reference to the window through its `keyWindow` property:

```
let w = UIApplication.sharedApplication().keyWindow!
```

That reference, however, is slightly volatile, because the system can create temporary windows an interpose them as the application's key window.

Experimenting With Views

In the course of this and subsequent chapters, you may want to experiment with views in a project of your own. Since view controllers aren't formally explained until Chapter 6, I'll just outline two simp approaches.

One way is to start your project with the Single View Application template. It gives you a main storyboard containing one scene containing one view controller instance containing its own main view; when the app runs, that view controller will become the app's main window's `rootViewController`, and its main view will become the window's root view. You can drag a view from the Object library into the main view as a subview, and it will be instantiated in the interface when the app runs. Alternatively, you can create views and add them to the interface in code; the simplest place to do this, for now, is the view controller's `viewDidLoad` method, which has a referen to the view controller's main view as `self.view`. For example:

```
override func viewDidLoad() {
    super.viewDidLoad()
    let mainview = self.view
    let v = UIView(frame:CGRectMake(100,100,50,50))
    v.backgroundColor = UIColor.redColor() // small red square
    mainview.addSubview(v) // add it to main view
}
```

Alternatively, you can start with the empty application without a storyboard that I described in Example 1-1. It has no *.xib* or *.storyboard* file, so your views will have to be created entirely in code. As I mentioned a moment ago, we did not assign any view controller to the window's `rootViewController` property, which causes the runtime to complain when the application is launched. A simple solution is to add a line of code in the app delegate's `application:didFinishLaunchingWithOptions:`, creating a minimal root view controller. We can then access its main view through its `view` property. For example:

```
func application(application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [NSObject: AnyObject]?)
    -> Bool {
        self.window = UIWindow(frame:UIScreen.mainScreen().bounds)
        self.window!.rootViewController = UIViewController() // *
        // and now we can add subviews
        let mainview = self.window!.rootViewController!.view
        let v = UIView(frame:CGRectMake(100,100,50,50))
        v.backgroundColor = UIColor.redColor() // small red square
        mainview.addSubview(v) // add it to main view
        // and the rest is as before...
        self.window!.backgroundColor = UIColor.whiteColor()
```

```
        self.window!.makeKeyAndVisible()
        return true
    }
```

Subview and Superview

Once upon a time, and not so very long ago, a view owned precisely its rectangular area. No part of any view that was not a subview of this view could appear inside it, because when this view redrew it rectangle, it would erase the overlapping portion of the other view. No part of any subview of this view could appear outside it, because the view took responsibility for its own rectangle and no more.

Those rules, however, were gradually relaxed, and starting in OS X 10.5, Apple introduced an entirely new architecture for view drawing that lifted those restrictions completely. iOS view drawing is based on this revised architecture. In iOS, some or all of a subview can appear outside its superview, and a view can overlap another view and can be drawn partially or totally in front of it without being its subview.

For example, Figure 1-1 shows three overlapping views. All three views have a background color, so each is completely represented by a colored rectangle. You have no way of knowing, from this visual representation, exactly how the views are related within the view hierarchy. In actual fact, the view in the middle (horizontally) is a sibling view of the view on the left (they are both direct subviews of the root view), and the view on the right is a subview of the middle view.

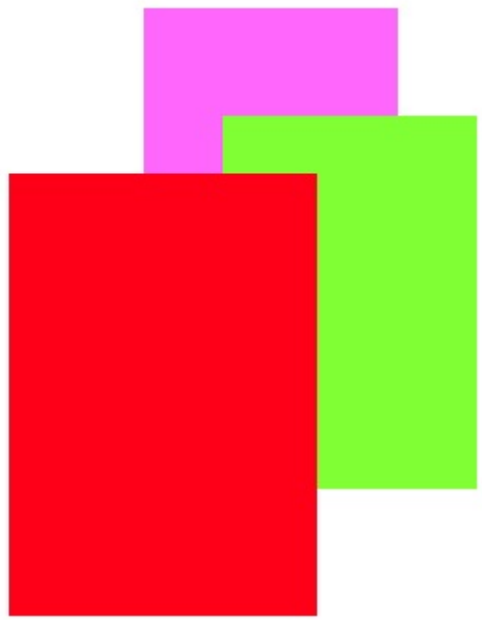

Figure 1-1. Overlapping views

When views are created in the nib, you can examine the view hierarchy in the nib editor's document outline to learn their actual relationship (Figure 1-2). When views are created in code, you know their hierarchical relationship because you created that hierarchy. But the visible interface doesn't tell you because view overlapping is so flexible.

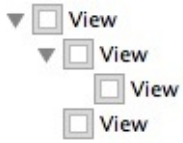

Figure 1-2. A view hierarchy as displayed in the nib editor

NOTE

To study your app's view hierarchy at runtime while paused in the debugger, choose Debug → View Debugging → Capture View Hierarchy (new in Xcode 6). I'll talk more about this feature later in this chapter.

Nevertheless, a view's position within the view hierarchy is extremely significant. For one thing, the view hierarchy dictates the *order* in which views are drawn. Sibling subviews of the same superview have a definite order: one is drawn before the other, so if they overlap, it will appear to be behind its sibling. Similarly, a superview is drawn before its subviews, so if they overlap it, it will appear to be behind them.

You can see this illustrated in Figure 1-1. The view on the right is a subview of the view in the middl and is drawn on top of it. The view on the left is a sibling of the view in the middle, but it is a later sibling, so it is drawn on top of the view in the middle and on top of the view on the right. The view o the left *cannot* appear behind the view on the right but in front of the view in the middle, because those two views are subview and superview and are drawn together — both are drawn either before o after the view on the left, depending on the ordering of the siblings.

This layering order can be governed in the nib editor by arranging the views in the document outline. (If you click in the canvas, you may be able to use the menu items of the Editor → Arrange menu instead — Send to Front, Send to Back, Send Forward, Send Backward.) In code, there are methods f arranging the sibling order of views, which we'll come to in a moment.

Here are some other effects of the view hierarchy:

- If a view is removed from or moved within its superview, its subviews go with it.
- A view's degree of transparency is inherited by its subviews.
- A view can optionally limit the drawing of its subviews so that any parts of them outside the view are not shown. This is called *clipping* and is set with the view's `clipsToBounds` property.
- A superview *owns* its subviews, in the memory-management sense, much as an array owns its elements; it retains them and is responsible for releasing a subview when that subview ceases to b its subview (it is removed from the collection of this view's subviews) or when it itself goes out o existence.
- If a view's size is changed, its subviews can be resized automatically (and I'll have much more to say about that later in this chapter).

A UIView has a `superview` property (a UIView) and a `subviews` property (an array of UIView objects, in back-to-front order), allowing you to trace the view hierarchy in code. There is also a method `isDescendantOfView:` letting you check whether one view is a subview of another at any depth. If you need a reference to a particular view, you will probably arrange this beforehand as a property, perhaps through an outlet. Alternatively, a view can have a numeric tag (its `tag` property), and can then be referred to by sending any view higher up the view hierarchy the `viewWithTag:` message. Seeing that all tags of interest are unique within their region of the hierarchy is up to you.

Manipulating the view hierarchy in code is easy. This is part of what gives iOS apps their dynamic quality, and it compensates for the fact that there is basically just a single window. It is perfectly reasonable for your code to rip an entire hierarchy of views out of the superview and substitute another! You can do this directly; you can combine it with animation (Chapter 4); you can govern it through view controllers (Chapter 6).

The method `addSubview:` makes one view a subview of another; `removeFromSuperview` takes a subview out of its superview's view hierarchy. In both cases, if the superview is part of the visible interface, the subview will appear or disappear; and of course this view may itself have subviews that accompany it. Just remember that removing a subview from its superview releases it; if you intend to reuse that subview later on, you will wish to retain it first. This is often taken care of by assignment to a property.

Events inform a view of these dynamic changes. To respond to these events requires subclassing. Then you'll be able to override any of these methods:

- `didAddSubview:`, `willRemoveSubview:`
- `didMoveToSuperview`, `willMoveToSuperview:`
- `didMoveToWindow`, `willMoveToWindow:`

When `addSubview:` is called, the view is placed last among its superview's subviews; thus it is drawn last, meaning that it appears frontmost. A view's subviews are indexed, starting at `0`, which is rearmost. There are additional methods for inserting a subview at a given index, or below (behind) or above (in front of) a specific view; for swapping two sibling views by index; and for moving a subview all the way to the front or back among its siblings:

- `insertSubview:atIndex:`
- `insertSubview:belowSubview:`, `insertSubview:aboveSubview:`
- `exchangeSubviewAtIndex:withSubviewAtIndex:`
- `bringSubviewToFront:`, `sendSubviewToBack:`

Oddly, there is no command for removing all of a view's subviews at once. However, a view's `subviews` array is an immutable copy of the internal list of subviews, so it is legal to cycle through it and remove each subview one at a time:

```
for v in myView.subviews as! [UIView] {
    v.removeFromSuperview()
}
```

Or, more compactly (deliberately misusing `map`):

```
(myView.subviews as! [UIView]).map{$0.removeFromSuperview()}
```

Visibility and Opacity

A view can be made invisible by setting its `hidden` property to `true`, and visible again by setting it to `false`. This takes it (and its subviews, of course) out of the visible interface without the overhead of actually removing it from the view hierarchy. A hidden view does not (normally) receive touch event so to the user it really is as if the view weren't there. But it is there, so it can still be manipulated in code.

A view can be assigned a background color through its `backgroundColor` property. A color is a UIColor; this is not a difficult class to use, and I'm not going to go into details. A view whose background color is nil (the default) has a transparent background. It is perfectly reasonable for a vie to have a transparent background and to do no additional drawing of its own, just so that it can act as convenient superview to other views, making them behave together.

A view can be made partially or completely transparent through its `alpha` property: `1.0` means opaque, `0.0` means transparent, and a value may be anywhere between them, inclusive. This affects subviews: if a superview has an `alpha` of `0.5`, none of its subviews can have an *apparent* opacity of more than `0.5`, because whatever `alpha` value they have will be drawn relative to `0.5`. (Just to make matters more complicated, colors have an alpha value as well. So, for example, a view can have an `alpha` of `1.0` but still have a transparent background because its `backgroundColor` has an alpha less than `1.0`.) A view that is completely transparent (or very close to it) is like a view whose `hidden` is `true`: it is invisible, along with its subviews, and cannot (normally) be touched.

A view's `alpha` property value affects both the apparent transparency of its background color and the apparent transparency of its contents. For example, if a view displays an image and has a background color and its `alpha` is less than 1, the background color will seep through the image (and whatever is behind the view will seep through both).

A view's `opaque` property, on the other hand, is a horse of a different color; changing it has no effect on the view's appearance. Rather, this property is a hint to the drawing system. If a view completely fills its bounds with ultimately opaque material and its `alpha` is `1.0`, so that the view has no effectiv transparency, then it can be drawn more efficiently (with less drag on performance) if you inform the drawing system of this fact by setting its `opaque` to `true`. Otherwise, you should set its `opaque` to `false`. The `opaque` value is *not* changed for you when you set a view's `backgroundColor` or `alpha`! Setting it correctly is entirely up to you; the default, perhaps surprisingly, is `true`.

Frame

A view's `frame` property, a CGRect, is the position of its rectangle within its superview, *in the superview's coordinate system.* By default, the superview's coordinate system will have the origin at its top left, with the x-coordinate growing positively rightward and the y-coordinate growing positively downward.

Setting a view's frame to a different CGRect value repositions the view, or resizes it, or both. If the view is visible, this change will be visibly reflected in the interface. On the other hand, you can also set a view's frame when the view is not visible — for example, when you create the view in code. In that case, the frame describes where the view *will* be positioned within its superview when it is given superview. UIView's designated initializer is `init(frame:)`, and you'll often assign a frame this wa especially because the default frame might otherwise be `(0.0,0.0,0.0,0.0)`, which is rarely what you want.

> **NOTE**
>
> Forgetting to assign a view a frame when creating it in code, and then wondering why it isn't appearing when added to a superview, is a common beginner mistake. A view with a zero-size frame is effectively invisible. If a view has a standard size that you want it to adopt, especially in relation to its contents (like a UIButton in relation to its title), an alternative is to call its `sizeToFit` method.

We are now in a position to generate programmatically the interface displayed in Figure 1-1:

```
let v1 = UIView(frame:CGRectMake(113, 111, 132, 194))
v1.backgroundColor = UIColor(red: 1, green: 0.4, blue: 1, alpha: 1)
let v2 = UIView(frame:CGRectMake(41, 56, 132, 194))
v2.backgroundColor = UIColor(red: 0.5, green: 1, blue: 0, alpha: 1)
let v3 = UIView(frame:CGRectMake(43, 197, 160, 230))
v3.backgroundColor = UIColor(red: 1, green: 0, blue: 0, alpha: 1)
mainview.addSubview(v1)
v1.addSubview(v2)
mainview.addSubview(v3)
```

In that code, we determined the layering order of `v1` and `v3` (the middle and left views, which are siblings) by the order in which we inserted them into the view hierarchy with `addSubview:`.

Bounds and Center

Suppose we have a superview and a subview, and the subview is to appear inset by 10 points, as in Figure 1-3. The Foundation utility function `CGRectInset` and the Swift CGRect method `rectByInsetting` make it easy to derive one rectangle as an inset from another, so we'll use one of them to determine the subview's frame. But *what* rectangle should this be inset from? Not the superview's frame; the frame represents a view's position within *its* superview, and in that superview's coordinates. What we're after is a CGRect describing our superview's rectangle in its *ov*

coordinates, because those are the coordinates in which the subview's frame is to be expressed. The CGRect that describes a view's rectangle in its own coordinates is the view's `bounds` property.

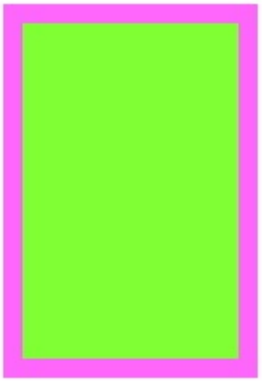

Figure 1-3. A subview inset from its superview

So, the code to generate Figure 1-3 looks like this:

```
let v1 = UIView(frame:CGRectMake(113, 111, 132, 194))
v1.backgroundColor = UIColor(red: 1, green: 0.4, blue: 1, alpha: 1)
let v2 = UIView(frame:v1.bounds.rectByInsetting(dx: 10, dy: 10))
v2.backgroundColor = UIColor(red: 0.5, green: 1, blue: 0, alpha: 1)
mainview.addSubview(v1)
v1.addSubview(v2)
```

You'll very often use a view's `bounds` in this way. When you need coordinates for drawing inside a view, whether drawing manually or placing a subview, you'll often refer to the view's `bounds`.

Interesting things happen when you set a view's bounds. If you change a view's bounds *size*, you change its *frame*. The change in the view's frame takes place around its *center*, which remains unchanged. So, for example:

```
let v1 = UIView(frame:CGRectMake(113, 111, 132, 194))
v1.backgroundColor = UIColor(red: 1, green: 0.4, blue: 1, alpha: 1)
let v2 = UIView(frame:v1.bounds.rectByInsetting(dx: 10, dy: 10))
v2.backgroundColor = UIColor(red: 0.5, green: 1, blue: 0, alpha: 1)
mainview.addSubview(v1)
v1.addSubview(v2)
v2.bounds.size.height += 20
v2.bounds.size.width += 20
```

What appears is a single rectangle; the subview completely and exactly covers its superview, its fram being the same as the superview's bounds. The call to `rectByInsetting` started with the superview's bounds and shaved 10 points off the left, right, top, and bottom to set the subview's frame (Figure 1-3). But then we added 20 points to the subview's bounds height and width, and thus added 20 points t the subview's frame height and width as well (Figure 1-4). The center didn't move, so we effectively put the 10 points back onto the left, right, top, and bottom of the subview's frame.