Move into iOS development by getting a firm grasp of its fundamentals, including the Xcode IDE, the Cocoa Touch framework, and Swift—Apple’s new programming language. With this thoroughly updated guide, you’ll learn Swift’s object-oriented concepts, understand how to use Apple’s development tools, and discover how Cocoa provides the underlying functionality iOS apps need to have.

- Explore Swift’s object-oriented concepts: variables and functions, scopes and namespaces, object types and instances
- Become familiar with built-in Swift types such as numbers, strings, ranges, tuples, Optionals, arrays, and dictionaries
- Learn how to declare, instantiate, and customize Swift object types—enums, structs, and classes
- Discover powerful Swift features such as protocols and generics
- Tour the lifecycle of an Xcode project from inception to App Store
- Create app interfaces with nibs and the nib editor, Interface Builder
- Understand Cocoa’s event-driven model and its major design patterns and features
- Find out how Swift communicates with Cocoa’s C and Objective-C APIs

Once you master the fundamentals, you’ll be ready to tackle the details of iOS app development with author Matt Neuburg’s companion guide, Programming iOS 8.

Matt Neuburg has a PhD in Classics and has taught at many colleges and universities. He has served as editor of MacTech magazine and as contributing editor for TidBITS. He has written many OS X and iOS applications. Previous books include Programming iOS 7, REALbasic: The Definitive Guide, and AppleScript: The Definitive Guide.
iOS 8 Programming Fundamentals with Swift

Move into iOS development by getting a firm grasp of its fundamentals, including the Xcode IDE, the Cocoa Touch framework, and Swift—Apple’s new programming language. With this thoroughly updated guide, you’ll learn Swift’s object-oriented concepts, understand how to use Apple’s development tools, and discover how Cocoa provides the underlying functionality iOS apps need to have.

- Explore Swift’s object-oriented concepts: variables and functions, scopes and namespaces, object types and instances
- Become familiar with built-in Swift types such as numbers, strings, ranges, tuples, Optionals, arrays, and dictionaries
- Learn how to declare, instantiate, and customize Swift object types—enums, structs, and classes
- Discover powerful Swift features such as protocols and generics
- Tour the lifecycle of an Xcode project from inception to App Store
- Create app interfaces with nibs and the nib editor, Interface Builder
- Understand Cocoa’s event-driven model and its major design patterns and features
- Find out how Swift communicates with Cocoa’s C and Objective-C APIs

Once you master the fundamentals, you’ll be ready to tackle the details of iOS app development with author Matt Neuburg’s companion guide, Programming iOS 8.

Matt Neuburg has a PhD in Classics and has taught at many colleges and universities. He has served as editor of MacTech magazine and as contributing editor for TidBITS. He has written many OS X and iOS applications. Previous books include Programming iOS 7, REALbasic: The Definitive Guide, and AppleScript: The Definitive Guide.

“Neuburg is my favorite programming book writer, period.” —John Gruber, Daring Fireball

Matt Neuburg

ISBN: 978-1-491-90890-7
US $49.99  CAN $57.99

Twitter: @oreillymedia
facebook.com/oreilly
Want to read more?

You can buy this book at oreilly.com in print and ebook format.

Buy 2 books, get the 3rd FREE!
Use discount code OPC10
All orders over $29.95 qualify for free shipping within the US.

It's also available at your favorite book retailer, including the iBookstore, the Android Marketplace, and Amazon.com.

©2015 O'Reilly Media, Inc. The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. 15055
# Table of Contents

**Preface.** ................................................................. xi

**Part I. Language**

1. **The Architecture of Swift.** ........................................... 3
   - Ground of Being ..................................................... 3
   - Everything Is an Object? ........................................... 5
   - Three Flavors of Object Type ..................................... 6
   - Variables ............................................................. 6
   - Functions ............................................................ 8
   - The Structure of a Swift File ..................................... 9
   - Scope and Lifetime ................................................ 11
   - Object Members .................................................... 12
   - Namespaces .......................................................... 13
   - Modules .............................................................. 14
   - Instances ............................................................ 14
   - Why Instances? ..................................................... 16
   - self ................................................................. 19
   - Privacy .............................................................. 20
   - Design ............................................................... 22
     - Object Types and APIs ......................................... 22
     - Instance Creation, Scope, and Lifetime .................. 24
     - Summary and Conclusion ..................................... 26

2. **Functions.** ............................................................ 27
   - Function Parameters and Return Value ....................... 27
   - Void Return Type and Parameters .............................. 30
5. Flow Control and More. ......................................................................................... 225
   Flow Control ................................................................. 225
     Branching ........................................................... 226
     Loops .................................................................. 239
     Jumping ............................................................... 243
     Aborting .............................................................. 244
   Operators ................................................................. 245
   Privacy ........................................................................ 248
     Private Declaration ........................................ 249
     Public Declaration ............................................. 251
     Privacy Rules .......................................................... 252
   Introspection ........................................................... 253
   Memory Management .................................................. 254
     Weak References ................................................ 255
     Unowned References ............................................ 257
     Weak and Unowned References in Anonymous Functions 258
     Memory Management of Protocol-Typed References 261

Part II. IDE

6. Anatomy of an Xcode Project. .............................................................................. 265
   New Project ................................................................ 265
   The Project Window .................................................... 268
     The Navigator Pane .................................................. 269
     The Utilities Pane .................................................... 274
     The Editor ............................................................... 276
   The Project File and Its Dependents ................................ 279
   The Target .................................................................. 282
     Build Phases .......................................................... 282
     Build Settings ............................................................. 285
     Configurations .......................................................... 286
     Schemes and Destinations ........................................ 287
   From Project to Running App ........................................... 289
     Build Settings .......................................................... 292
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property List Settings</td>
<td>292</td>
</tr>
<tr>
<td>Nib Files</td>
<td>293</td>
</tr>
<tr>
<td>Additional Resources</td>
<td>294</td>
</tr>
<tr>
<td>Code Files and the App Launch Process</td>
<td>296</td>
</tr>
<tr>
<td>Frameworks and SDKs</td>
<td>301</td>
</tr>
<tr>
<td>Renaming Parts of a Project</td>
<td>304</td>
</tr>
<tr>
<td>Bilingual Targets</td>
<td>305</td>
</tr>
<tr>
<td><strong>7. Nib Management</strong></td>
<td>309</td>
</tr>
<tr>
<td>A Tour of the Nib Editor Interface</td>
<td>310</td>
</tr>
<tr>
<td>The Document Outline</td>
<td>312</td>
</tr>
<tr>
<td>Canvas</td>
<td>314</td>
</tr>
<tr>
<td>Inspectors and Libraries</td>
<td>317</td>
</tr>
<tr>
<td>Nib Loading</td>
<td>318</td>
</tr>
<tr>
<td>When Nibs Are Loaded</td>
<td>319</td>
</tr>
<tr>
<td>Manual Nib Loading</td>
<td>320</td>
</tr>
<tr>
<td>Connections</td>
<td>322</td>
</tr>
<tr>
<td>Outlets</td>
<td>323</td>
</tr>
<tr>
<td>The Nib Owner</td>
<td>323</td>
</tr>
<tr>
<td>Automatically Configured Nibs</td>
<td>327</td>
</tr>
<tr>
<td>Misconfigured Outlets</td>
<td>328</td>
</tr>
<tr>
<td>Deleting an Outlet</td>
<td>330</td>
</tr>
<tr>
<td>More Ways to Create Outlets</td>
<td>330</td>
</tr>
<tr>
<td>Outlet Collections</td>
<td>333</td>
</tr>
<tr>
<td>Action Connections</td>
<td>334</td>
</tr>
<tr>
<td>More Ways to Create Actions</td>
<td>336</td>
</tr>
<tr>
<td>Misconfigured Actions</td>
<td>337</td>
</tr>
<tr>
<td>Connections Between Nibs — Not!</td>
<td>338</td>
</tr>
<tr>
<td>Additional Configuration of Nib-Based Instances</td>
<td>338</td>
</tr>
<tr>
<td><strong>8. Documentation</strong></td>
<td>343</td>
</tr>
<tr>
<td>The Documentation Window</td>
<td>344</td>
</tr>
<tr>
<td>Class Documentation Pages</td>
<td>346</td>
</tr>
<tr>
<td>Sample Code</td>
<td>349</td>
</tr>
<tr>
<td>Quick Help</td>
<td>349</td>
</tr>
<tr>
<td>Symbols</td>
<td>351</td>
</tr>
<tr>
<td>Header Files</td>
<td>352</td>
</tr>
<tr>
<td>Internet Resources</td>
<td>353</td>
</tr>
</tbody>
</table>
9. **Life Cycle of a Project**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device Architecture and Conditional Code</td>
<td>355</td>
</tr>
<tr>
<td>Backward Compatibility</td>
<td>356</td>
</tr>
<tr>
<td>Device Type</td>
<td>358</td>
</tr>
<tr>
<td>Version Control</td>
<td>360</td>
</tr>
<tr>
<td>Editing and Navigating Your Code</td>
<td>362</td>
</tr>
<tr>
<td>Autocompletion</td>
<td>363</td>
</tr>
<tr>
<td>Snippets</td>
<td>365</td>
</tr>
<tr>
<td>Fix-it and Live Syntax Checking</td>
<td>365</td>
</tr>
<tr>
<td>Navigation</td>
<td>366</td>
</tr>
<tr>
<td>Finding</td>
<td>368</td>
</tr>
<tr>
<td>Running in the Simulator</td>
<td>369</td>
</tr>
<tr>
<td>Debugging</td>
<td>370</td>
</tr>
<tr>
<td>Caveman Debugging</td>
<td>370</td>
</tr>
<tr>
<td>The Xcode Debugger</td>
<td>373</td>
</tr>
<tr>
<td>Unit Testing</td>
<td>379</td>
</tr>
<tr>
<td>Clean</td>
<td>383</td>
</tr>
<tr>
<td>Running on a Device</td>
<td>384</td>
</tr>
<tr>
<td>Obtaining a Certificate</td>
<td>386</td>
</tr>
<tr>
<td>Obtaining a Development Provisioning Profile</td>
<td>389</td>
</tr>
<tr>
<td>Running the App</td>
<td>390</td>
</tr>
<tr>
<td>Profile and Device Management</td>
<td>390</td>
</tr>
<tr>
<td>Profiling</td>
<td>391</td>
</tr>
<tr>
<td>Gauges</td>
<td>391</td>
</tr>
<tr>
<td>Instruments</td>
<td>392</td>
</tr>
<tr>
<td>Localization</td>
<td>395</td>
</tr>
<tr>
<td>Localizing the Info.plist</td>
<td>396</td>
</tr>
<tr>
<td>Localizing a Nib File</td>
<td>398</td>
</tr>
<tr>
<td>Localizing Code Strings</td>
<td>399</td>
</tr>
<tr>
<td>Localizing With XML Files</td>
<td>401</td>
</tr>
<tr>
<td>Archiving and Distribution</td>
<td>403</td>
</tr>
<tr>
<td>Ad Hoc Distribution</td>
<td>405</td>
</tr>
<tr>
<td>Final App Preparations</td>
<td>407</td>
</tr>
<tr>
<td>Icons in the App</td>
<td>407</td>
</tr>
<tr>
<td>Other Icons</td>
<td>408</td>
</tr>
<tr>
<td>Launch Images</td>
<td>409</td>
</tr>
<tr>
<td>Screenshots and Video Previews</td>
<td>410</td>
</tr>
<tr>
<td>Property List Settings</td>
<td>411</td>
</tr>
<tr>
<td>Submission to the App Store</td>
<td>412</td>
</tr>
</tbody>
</table>
Part III. Cocoa

10. Cocoa Classes ................................................................. 417
    Subclassing ........................................... 417
    Categories and Extensions .................. 420
        How Swift Uses Extensions ............. 421
        How You Use Extensions ............... 421
        How Cocoa Uses Categories .......... 422
    Protocols .................................................. 423
        Informal Protocols ..................... 425
        Optional Methods ...................... 425
    Some Foundation Classes ..................... 428
        Useful Structs and Constants ........ 428
        NSString and Friends .................. 429
        NSDate and Friends .................... 432
        NSNumber .................................. 433
        NSValue .................................. 435
        NSData ...................................... 436
        Equality and Comparison .......... 436
        NSIndexSet .................................. 438
        NSArray and NSMutableArray ........ 439
        NSDictionary and NSMutableDictionary 440
        NSSet and Friends ...................... 441
        NSNull ........................................ 442
        Immutable and Mutable ............... 443
        Property Lists ............................ 444
    Accessors, Properties, and Key–Value Coding 445
        Swift Accessors .............. 446
        Key–Value Coding ............. 448
        Uses of Key–Value Coding ...... 449
        KVC and Outlets ..................... 451
        Key Paths .............................. 451
        Array Accessors ...................... 452
    The Secret Life of NSObject .................. 453

11. Cocoa Events ............................................................. 457
    Reasons for Events .............................. 457
    Subclassing ........................................ 458
    Notifications ...................................... 459
On June 2, 2014, Apple’s WWDC keynote address ended with a shocking announcement: “We have a new programming language.” This came as a huge surprise to the developer community, which was accustomed to Objective-C, warts and all, and doubted that Apple could ever possibly relieve them from the weight of its venerable legacy. The developer community, it appeared, had been wrong.

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.

The Swift language is designed from the ground up with these salient features:

**Object-orientation**
Swift is a modern, object-oriented language. It is *purely* object-oriented: “Everything is an object.”

**Clarity**
Swift is easy to read and easy to write, with minimal syntactic sugar and few hidden shortcuts. Its syntax is clear, consistent, and explicit.

**Safety**
Swift enforces strong typing to ensure that it knows, and that you know, what the type of every object reference is at every moment.

**Economy**
Swift is a fairly small language, providing some basic types and functionalities and no more. The rest must be provided by your code, or by libraries of code that you use — such as Cocoa.
Memory management
Swift manages memory automatically. You will rarely have to concern yourself with memory management.

Cocoa compatibility
The Cocoa APIs are written in C and Objective-C. Swift is explicitly designed to interface with most of the Cocoa APIs.

These features make Swift an excellent language for learning to program iOS.

The alternative, Objective-C, still exists, and you can use it if you like. Indeed, it is easy to write an app that includes both Swift code and Objective-C code; and you may have reason to do so. Objective-C offers direct compatibility with the Cocoa APIs; Objective-C is C, and is sometimes needed in order to do things with C functions that are required by Cocoa but impossible using Swift alone.

Objective-C, however, lacks the very the advantages that Swift offers. Objective-C agglomerates object-oriented features onto C. It is therefore only partially object-oriented; it has both objects and scalar data types, and its objects have to be slotted into one particular C data type (pointers). Its syntax can be difficult and tricky; reading and writing nested method calls can make one's eyes glaze over, and it invites hacky habits such as implicit nil-testing. Its type checking can be and frequently is turned off, resulting in programmer errors where a message is sent to the wrong type of object and the program crashes. It uses manual memory management; the recent introduction of ARC (automatic reference counting) has alleviated some of the programmer tedium and has greatly reduced the likelihood of programmer error, but errors are still possible, and memory management ultimately remains manual.

Recent revisions and additions to Objective-C — ARC, synthesis and autosynthesis, improved literal array and dictionary syntax, blocks — have made it easier and more convenient, but such patches have also made the language even larger and possibly even more confusing. Because Objective-C must encompass C, there are limits to how far it can be extended and revised. Swift, on the other hand, is a clean start. If you were to dream of completely revising Objective-C to create a better Objective-C, Swift might be what you would dream of. It puts a modern, rational front end between you and the Cocoa Objective-C APIs.

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 constitutes 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 @dynamic in the Objective-C sense; and so on.

For these reasons, Objective-C still lurks in the shadows of this edition, and will occasionally leap out and make its presence known. I do not attempt to teach Objective-C — for that, see the previous edition of this book — but I describe it in enough detail to allow you to read it when you encounter it in the documentation and on the Internet, and I occasionally show some Objective-C code. Chapter 6 describes how your app can be written partly in Swift and partly in Objective-C. Part III, on Cocoa, is really all about learning to think the way Objective-C thinks — because the structure and behavior of the Cocoa APIs are fundamentally based on Objective-C. And the book ends with an appendix that details how Swift and Objective-C communicate with one another.

### The Scope of This Book

This book is actually one of a pair with my *Programming iOS 8*, which picks up exactly where this book leaves off. They complement and supplement one another. The two-book architecture should, I believe, render the size and scope of each book tractable for readers. Together, they provide a complete grounding in the knowledge needed to begin writing iOS apps; thus, when you do start writing iOS apps, you’ll have a solid and rigorous understanding of what you are doing and where you are heading. If writing an iOS program is like building a house of bricks, this book teaches you what a brick is and how to handle it, while *Programming iOS 8* hands you some actual bricks and tells you how to assemble them.

When you have read this book, you’ll know about Swift, Xcode, and the underpinnings of the Cocoa framework, and you will be ready to proceed directly to *Programming iOS 8*. Conversely, *Programming iOS 8* assumes a knowledge of this book; it begins, like Homer’s *Iliad*, 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. If you started reading *Programming iOS 8* and wondered about such unexplained matters as Swift language basics, the UIApplicationMain function, the nib-loading mechanism, Cocoa patterns of delegation and notification, and retain cycles, wonder no longer — I didn’t explain them there because I do explain them here.

The three parts of this book teach the underlying basis of all iOS programming:
• **Part I** introduces the Swift language, from the ground up — I do not assume that you know any other programming languages. My way of teaching Swift is different from other treatments, such as Apple's; it is systematic and Euclidean, with pedagogical building blocks piled on one another in what I regard as the most helpful order. At the same time, I have tried to confine myself to the essentials. Swift is not a big language, but it has some subtle and unusual corners. You don't need to dive deep into all of these, and my discussion will leave many of them unexplored. You will probably never encounter them, and if you do, you will have entered an advanced Swift world outside the scope of this discussion. To give an obvious example, readers may be surprised to find that I never mention Swift playgrounds or the REPL. My focus here is real-life iOS programming, and my explanation of Swift therefore concentrates on those common, practical aspects of the language that, in my experience, actually come into play in the course of programming iOS.

• **Part II** turns to Xcode, the world in which all iOS programming ultimately takes place. It explains what an Xcode project is and how it is transformed into an app, and how to work comfortably and nimbly with Xcode to consult the documentation and to write, navigate, and debug code, as well as how to bring your app through the subsequent stages of running on a device and submission to the App Store. There is also a very important chapter on nibs and the nib editor (Interface Builder), including outlets and actions as well as the mechanics of nib loading; however, such specialized topics as autolayout constraints in the nib are postponed to the other book.

• **Part III** introduces the Cocoa Touch framework. When you program for iOS, you take advantage of a suite of frameworks provided by Apple. These frameworks, taken together, constitute Cocoa; the brand of Cocoa that provides the API for programming iOS is Cocoa Touch. Your code will ultimately be almost entirely about communicating with Cocoa. The Cocoa Touch frameworks provide the underlying functionality that any iOS app needs to have. But to use a framework, you have to think the way the framework thinks, put your code where the framework expects it, and fulfill many obligations imposed on you by the framework. To make things even more interesting, Cocoa uses Objective-C, while you'll be using Swift: you need to know how your Swift code will interface with Cocoa's features and behaviors. Cocoa provides important foundational classes and adds linguistic and architectural devices such as categories, protocols, delegation, and notifications, as well as the pervasive responsibilities of memory management. Key–value coding and key–value observing are also discussed here.

The reader of this book will thus get a thorough grounding in the fundamental knowledge and techniques that any good iOS programmer needs. The book itself doesn't show how to write any particularly interesting iOS apps, but it does constantly use my own real apps and real programming situations to illustrate and motivate its explanations. And then you'll be ready for *Programming iOS 8*, of course!
Versions

This book is geared to Swift 1.2, iOS 8.3, and Xcode 6.3, which were in early beta at the time the book was completed. In general, only very minimal attention is given to earlier versions of iOS and Xcode. 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 com‐
pendiously available in my earlier books. The book does contain, nevertheless, a few words of advice about backward compatibility (especially in Chapter 9).

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 patterns of protocol and delegate; but that’s no problem, because I can illustrate them for you (Chapter 11). The loss of the Empty Application template 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, in Chapter 6, I give instructions for turning a Single View Application–based template into something sim‐
ilar to what the Empty Application template would have given you. Also, the Xcode 6 templates are based primarily on storyboards; although I treat storyboards as the pri‐
mary Interface Builder design milieu, I still demonstrate how to make and work with a .xib file.

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, and Swift 1.2, which I have documented in this book, was still in beta. 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.

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 ed‐
tion, but whose influence is present throughout.

Some details of the presentation of the Swift language came from suggestions by Andrew Duncan, whose book on Objective-C is a classic in its own right.

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)
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 period 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 (UIView-Controller) 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 felt 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 of 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 views 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.
Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at http://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: “iOS 8 Programming Fundamentals with Swift by Matt Neuburg (O’Reilly). Copyright 2015 Matt Neuburg, 978-1-491-90890-7.”

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.


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/ios8-prog-fundamentals.

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
This part of the book teaches the Swift language, from the ground up. The description is rigorous and orderly. Here you’ll become sufficiently conversant with Swift to be comfortable with it, so that you can proceed to the practical business of actual programming.

- **Chapter 1** surveys the structure of a Swift program, both physically and conceptually. You’ll learn how Swift code files are organized, and you’ll be introduced to the most important underlying concepts of the object-oriented Swift language: variables and functions, scopes and namespaces, object types and their instances.

- **Chapter 2** explores Swift functions. We start with the basics of how functions are declared and called; then we discuss parameters — external parameter names, default parameters, and variadic parameters. Then we dive deep into the power of Swift functions, with an explanation of functions inside functions, functions as first-class values, anonymous functions, functions as closures, and curried functions.

- **Chapter 3** starts with Swift variables — their scope and lifetime, and how they are declared and initialized, along with important Swift features such as computed variables and setter observers. Then some important built-in Swift types are introduced, including Booleans, numbers, strings, ranges, tuples, and Optionals.

- **Chapter 4** is all about Swift object types — classes, structs, and enums. It explains how these three object types work, and how you declare, instantiate, and use them. Then it proceeds to polymorphism and casting, protocols, generics, and extensions. The chapter concludes with a discussion of Swift’s umbrella types (such as Any-Object) and collection types (Array, Dictionary, and Set).
• **Chapter 5** is a miscellany. We start with Swift’s flow control structures for branching and looping. Then I’ll explain how to create your own Swift operators. The chapter concludes by describing Swift access control (privacy), introspection (reflection), and memory management.
It will be useful at the outset for you to have a general sense of how the Swift language is constructed and what a Swift-based iOS program looks like. This chapter will survey the overall architecture and nature of the Swift language. Subsequent chapters will fill in the details.

**Ground of Being**

A complete Swift command is a *statement*. A Swift text file consists of multiple *lines* of text. Line breaks are meaningful. The typical layout of a program is one statement, one line:

```swift
print("hello ")
print("world")
```

However, that’s not a hard and fast rule. You can combine more than one statement on a line, but then you need to put a semicolon between them:

```swift
print("hello "); print("world")
```

You are free to put a semicolon at the end of a statement that is last or alone on its line, but no one ever does (except out of habit, because C and Objective-C *require* the semicolon):

```swift
print("hello ");
print("world");
```

Conversely, a single statement can be broken into multiple lines, in order to prevent long statements from becoming long lines. But you should try to do this at sensible places so as not to confuse Swift. For example, after an opening parenthesis is a good place:

```swift
print(
    "world"
)
Comments are everything after two slashes in a line (so-called C++-style comments):

```swift
print("world") // this is a comment, so Swift ignores it
```

You can also enclose comments in /*...*/, as in C. Unlike C, C-style comments can be nested.

Many constructs in Swift use curly braces as delimiters:

```swift
class Dog {
    func bark() {
        println("woof")
    }
}
```

By convention, the contents of curly braces are preceded and followed by line breaks and are indented for clarity, as shown in the preceding code. Xcode will help impose this convention, but the truth is that Swift doesn't care, and layouts like this are legal (and are sometimes more convenient):

```swift
class Dog { func bark() { println("woof") }}
```

Instant feedback in the Xcode console is provided by the `print` and `println` commands. The difference is that `println` appends a linefeed to its output.

Swift is a compiled language. This means that your code must build — passing through the compiler and being turned from text into some lower-level form that a computer can understand — before it can run and actually do the things it says to do. The Swift compiler is very strict; in the course of writing a program, you will often try to build and run, only to discover that you can't even build in the first place, because the compiler will flag some error, which you will have to fix if you want the code to run. Less often, the compiler will let you off with a warning; the code can run, but in general you should take warnings seriously and fix whatever they are telling you about. The strictness of the compiler is one of Swift's greatest strengths, and provides your code with a large measure of audited correctness even before it ever starts running.

As of this writing, the Swift compiler's error and warning messages range from the insightful to the obtuse to the downright misleading. You will often know that something is wrong with a line of code, but the Swift compiler will not be telling you clearly exactly what is wrong or even where in the line to focus your attention. My advice in these situations is to pull the line apart into several lines of simpler code until you reach a point where you can guess what the issue is. Try to love the compiler despite the occasional unhelpful nature of its messages. Remember, it knows more than you do, even if it is sometimes rather inarticulate about its knowledge.
Everything Is an Object?

In Swift, “everything is an object.” That’s a boast common to various modern object-oriented languages, but what does it mean? Well, that depends on what you mean by “object” — and what you mean by “everything.”

Let's start by stipulating that an object, roughly speaking, is something you can send a message to. A message, roughly speaking, is an imperative instruction. For example, you can give commands to a dog: “Bark!” “Sit!” In this analogy, those phrases are messages, and the dog is the object to which you are sending those messages.

In Swift, the syntax of message-sending is *dot-notation*. We start with the object; then there’s a dot (a period); then there's the message. (Some messages are also followed by parentheses, but ignore them for now; the full syntax of message-sending is one of those details we’ll be filling in later.) This is valid Swift syntax:

```
  fido.bark()
  rover.sit()
```

The idea of *everything* being an object is a way of suggesting that even “primitive” linguistic entities can be sent messages. Take, for example, 1. It appears to be a literal digit and no more. It will not surprise you, if you’ve ever used any programming language, that you can say things like this in Swift:

```
  let sum = 1 + 2
```

But it is surprising to find that 1 can be followed by a dot and a message. This is legal and meaningful in Swift (don't worry about what it actually means):

```
  let x = 1.successor()
```

Similarly, a piece of text expressed as a literal — a *string* — is an object. For example, "hello" is a literal string, and this is legal Swift (again, never mind what this code means):

```
  let y = "hello".generate()
```

But we can go further. Return to that innocent-looking 1 + 2 from our earlier code. It turns out that this is actually a kind of syntactic trickery, a convenient way of expressing and hiding what's really going on. Just as 1 is actually an object, + is actually a message; but it's a message with special syntax (*operator* syntax). In Swift, every noun is an object, and every verb is a message.

Perhaps the ultimate acid test for whether something is an object in Swift is whether you can modify it. An object type can be *extended* in Swift, meaning that you can define your own messages on that type. For example, you can't normally send the *sayHello* message to a number. But you can change a number type so that you can:
extension Int {
    func sayHello() {
        println("Hello, I'm \(self)")
    }
}  
1.sayHello() // outputs: "Hello, I'm 1"

I rest my case.

In Swift, then, 1 is an object. In some languages, such as Objective-C, it clearly is not; it is a “primitive” or scalar built-in data type. The distinction being drawn here, then, when we say that “everything is an object,” is between object types on the one hand and scalars on the other. In Swift, there are no scalars; all types are ultimately object types. That’s what “everything is an object” really means.

**Three Flavors of Object Type**

If you know Objective-C or some other object-oriented language, you may be surprised by Swift’s notion of what kind of object 1 is. In many languages, such as Objective-C, an object is a class or an instance of a class. Swift has classes and instances, and you can send messages to them; but 1 in Swift is neither of those: it’s a struct. And Swift has yet another kind of thing you can send messages to, called an enum.

So Swift has three kinds of object type: classes, structs, and enums. I like to refer to these as the three flavors of object type. Exactly how they differ from one another will emerge in due course. But they are all very definitely object types, and their similarities to one another are far stronger than their differences. For now, just bear in mind that these three flavors exist.

(The fact that a struct or enum is an object type in Swift will surprise you particularly if you know Objective-C. Objective-C has structs and enums, but they are not objects. Swift structs, in particular, are much more important and pervasive than Objective-C structs. This difference between how Swift views structs and enums and how Objective-C views them can matter when you are talking to Cocoa.)

**Variables**

A variable is a name for an object. Technically, it refers to an object; it is an object reference. Nontechnically, you can think of it as a shoebox into which an object is placed. The object may undergo changes, or it may be replaced inside the shoebox by another object, but the name has an integrity all its own.

In Swift, no variable comes implicitly into existence; all variables must be declared. If you need a name for something, you must say “I’m creating a name.” You do this with one of two keywords: let or var. In Swift, declaration is usually accompanied by
*initialization* — you use an equal sign to give the variable a *value*, right there as part of the declaration. These are both variable declarations (and initializations):

```swift
let one = 1
var two = 2
```

Once the name exists, you are free to use it. For example, we can change the value of what's in `two` to be the same as the value of what's in `one`:

```swift
let one = 1
var two = 2
two = one
```

The last line of that code uses both the name `one` and the name `two` declared in the first two lines: the name `one`, on the right side of the equal sign, is used merely to *refer* to the value inside the shoebox (namely `1`); but the name `two`, on the left side of the equal sign, is used to *replace* the value inside the shoebox. A statement like that, with a variable name on the left side of an equal sign, is called an *assignment*, and the equal sign is the *assignment operator*. The equal sign is not an assertion of equality, as it might be in an algebraic formula; it is a command. It means: “Get the value of what's on the right side of me, and use it to replace the value inside what's on the left side of me.”

The two kinds of variable declaration differ in that a name declared with `let` cannot have its object replaced. A variable declared with `let` is a *constant*; its value is assigned once and stays. This won't even compile:

```swift
let one = 1
var two = 2
one = two // compile error
```

It is always possible to declare a name with `var` to give yourself the most flexibility, but if you know you’re never going to replace the initial value of a variable, it's better to use `let`, as this permits Swift to behave much more efficiently.

Variables also have a type. This type is established when the variable is declared and *can never change*. For example, this won't compile:

```swift
var two = 2
two = "hello"
```

Once `two` is declared and initialized as `2`, it is a number (properly speaking, an `Int`) and it must always be so. You can replace its value with `1` because that's also an `Int`, but you can't replace its value with "hello" because that's a string (properly speaking, a `String`) — and a `String` is not an `Int`.

Variables literally have a life of their own — more accurately, a *lifetime* of their own. As long as a variable exists, it keeps its value alive. Thus, a variable can be not only a way of conveniently naming something, but also a way of preserving it. I’ll have more to say about that later.
By convention, type names such as String or Int (or Dog or Cat) start with a capital letter; variable names start with a small letter. *Do not violate this convention.* If you do, your code might still compile and run just fine, but I will personally send agents to your house to remove your kneecaps.

## Functions

Executable code, like `fido.bark()` or `one = two`, cannot go just anywhere. In general, it must live inside the body of a *function*. A function is a batch of code, a succession of lines of code that can be told, as a batch, to run. Typically, a function has a name, and it gets that name through a function declaration. Function declaration syntax is another of those details that will be filled in later, but here’s an example:

```swift
func go() {
    let one = 1
    var two = 2
    two = one
}
```

That describes a sequence of things to do — declare one, declare two, change the value of two to match the value of one — and it gives that sequence a name, go; but it doesn’t *perform* the sequence. The sequence is performed when someone *calls* the function. Thus, we might say, elsewhere:

```swift
go()
```

That is a command to the go function that it should actually run. But again, that command is itself executable code, so it cannot live on its own either. It might live in a different function declaration:

```swift
func doGo() {
    go()
}
```

But wait! This is getting a little nutty. That, too, is just a function declaration; to run it, someone must call doGo, and that’s executable code too. This seems like some kind of infinite regression; it looks like none of our code will *ever* run. If all executable code has to live in a function, who will tell *any* function to run? The initial impetus must come from somewhere.

In real life, fortunately, this regression problem doesn’t arise. Remember that your goal is ultimately to write an iOS app. Thus, your app will be run on an iOS device (or the Simulator) by a runtime that already wants to call certain functions. So you start by writing special functions that you know the runtime itself will call. That gives your app a way to get started and gives you places to put functions that will be called by the runtime at key moments — such as when the app launches, or when the user taps a button in your app’s interface.
Swift also has a special rule that a file called `main.swift`, exceptionally, can have executable code at its top level, outside any function body, and this is the code that actually runs when the program runs. You can construct your app with a `main.swift` file, but in general you won’t need to. Also, Xcode allows you to create playgrounds. A playground file has the special property that it acts like a `main.swift` file, so that you are allowed to put executable code at the top level of the file, outside any function body. But playgrounds can’t be part of an iOS app, and are not considered in this book.

The Structure of a Swift File

A Swift program can consist of one file or many files. In Swift, a file is a meaningful unit, and there are definite rules about the structure of the Swift code that can go inside it. (I’m assuming that we are not in a `main.swift` file or a playground.) Then only certain things can go at the top level of this file:

**Module import statements**

A module is an even higher-level unit than a file. A module can consist of multiple files, and in Swift, the files within a module can all see each other automatically; but a module can’t see another module without an `import` statement. For example, that is how you are able to talk to Cocoa in an iOS program: the first line of your file says `import UIKit`.

**Variable declarations**

A variable declared at the top level of a file is a global variable: it lives as long as the program runs.

**Function declarations**

A function declared at the top level of a file is a global function: all code will be able to see and call it, without sending a message to any object.

**Object type declarations**

The declaration for a class, a struct, or an enum.

For example, this is a legal Swift file containing (just to demonstrate that it can be done) an `import` statement, a variable declaration, a function declaration, a class declaration, a struct declaration, and an enum declaration:

```swift
import UIKit
var one = 1
func changeOne() {
}
class Manny {
}
```
struct Moe {
}

enum Jack {
}

That’s a very silly and mostly empty example, but remember, our goal is to survey the parts of the language and the structure of a file, and the example shows them.

Furthermore, the curly braces for each of the things in that example can all have variable declarations, function declarations, and object type declarations within them! Indeed, any structural curly braces can contain such declarations. So, for example, the keyword if (which is part of Swift’s flow control, discussed in Chapter 5) is followed by structural curly braces, and they can contain variable declarations, function declarations, and object type declarations. This code, while silly, is legal:

```swift
func silly() {
    if true {
        class Cat {}
        var one = 1
        one = one + 1
    }
}
```

But only a function declaration, remember, can contain executable code. It can contain executable code at any depth within itself; in the preceding code, the line `one = one + 1`, which is executable code, is legal because it is inside the if construct, which is inside a function declaration. But the line `one = one + 1 cannot go at the top level of the file; and it cannot go directly inside the Cat declaration’s curly braces.

**Example 1-1** is a legal Swift file, schematically illustrating the structural possibilities. (Ignore the hanky-panky with the `name` variable declaration inside the enum declaration for Jack; enum top-level variables have some special rules that I’ll explain later.)

**Example 1-1. Schematic structure of a legal Swift file**

```swift
import UIKit
var one = 1
func changeOne() {
    let two = 2
    func sayTwo() {
        println(two)
    }
    class Klass {}
    struct Struct {}
    enum Enum {}
    one = two
}

class Manny {
    let name = "manny"
    func sayName() {
        println(name)
    }
```
class Klass {}
struct Struct {}
enum Enum {}

struct Moe {
let name = "moe"
func sayName() {
    println(name)
}
class Klass {}
struct Struct {}
enum Enum {}
}

enum Jack {
    var name : String {
        return "jack"
    }
    func sayName() {
        println(name)
    }
class Klass {}
struct Struct {}
enum Enum {}
}

Obviously, we can recurse down as far we like: we could have a class declaration containing a class declaration containing a class declaration…and so on. But there’s no point illustrating that.

Scope and Lifetime

In a Swift program, things have a scope. This refers to their ability to be seen by other things. Things are nested inside of other things, making a nested hierarchy of things. The rule is that things can see things at their own level and higher. The levels are:

- A module is a scope.
- A file is a scope.
- An object declaration is a scope.
- Curly braces are a scope.

When something is declared, it is declared at some level within that hierarchy. Its place in the hierarchy — its scope — determines whether it can be seen by other things.

For example, look again at Example 1-1. Inside the declaration of Manny is a name variable declaration and a sayName function declaration; the code inside sayName’s curly braces can see things outside those curly braces at a higher level, and can therefore see
the *name* variable. Similarly, the code inside the body of the `changeOne` function declaration can see the `one` variable declared at the top level of the file; indeed, *everything* throughout this file can see the `one` variable declared at the top level of the file.

Scope is thus a very important way of *sharing information*. Two different functions declared inside Manny would *both* be able to see the `name` declared at Manny’s top level. Code inside Jack and code inside Moe can *both* see the `one` declared at the file’s top level.

Things also have a *lifetime*, which is effectively equivalent to their scope. A thing lives as long as its surrounding scope lives. Thus, the variable `one` lives as long as the file lives — namely, as long the program runs. It is global and permanent. But the variable `name` declared at the top level of Manny exists only so long as Manny exists (I’ll talk in a moment about what that means). Things declared at a deeper level live even shorter lifetimes; for example, let’s return to this code:

```swift
func silly() {
    if true {
        class Cat {}
        var one = 1
        one = one + 1
    }
}
```

In that code, the class `Cat` and the variable `one` exist only during the brief instant that the path of code execution passes through the if construct. When the function `silly` is called, `Cat` is declared and comes into existence; then `one` is declared and comes into existence; then the executable line `one = one + 1` is executed; and then the scope ends and both `Cat` and `one` vanish in a puff of smoke.

## Object Members

Inside the three object types (class, struct, and enum), things declared at the top level have special names, mostly for historical reasons. Let’s use the Manny class as an example:

```swift
class Manny {
    let name = "manny"
    func sayName() {
        println(name)
    }
}
```

In that code:

- `name` is a variable declared at the top level of an object declaration, so it is called a *property* of that object.
• **sayName** is a function declared at the top level of an object declaration, so it is called a *method* of that object.

Things declared at the top level of an object declaration — properties, methods, and any objects declared at that level — are collectively the *members* of that object. Members have a special significance, because they define the *messages* you are allowed to send to that object!

### Namespaces

A *namespace* is a named region of a program. A namespace has the property that the names of things inside it cannot be reached by things outside it without somehow first passing through the barrier of saying that region’s name. This is a good thing because it allows the same name to be used in different places without a conflict. Clearly, namespaces and scopes are closely related notions.

Namespaces help to explain the significance of declaring an object at the top level of an object, like this:

```javascript
class Manny {
    class Klass {}
}
```

This way of declaring Klass effectively “hides” it inside Manny. Manny is a namespace! Code *inside* Manny can see (and say) Klass directly. But code outside Manny can’t do that. It has to specify the namespace *explicitly* in order to pass through the barrier that the namespace represents. To do so, it must say Manny’s name first, followed by a dot, followed by the term Klass. In short, it has to say `Manny.Klass`.

The namespace does not, of itself, provide secrecy or privacy; it’s a convenience. Thus, in *Example 1-1*, I gave Manny a Klass class, and I also gave Moe a Klass class. But they don’t conflict, because they are in different namespaces, and I can differentiate them, if necessary, as `Manny.Klass` and `Moe.Klass`.

It will not have escaped your attention that the syntax for diving explicitly into a namespace is the message-sending dot-notation syntax. They are, in fact, the same thing.

In effect, message-sending allows you to see into scopes you can’t see into otherwise. Code *inside* Moe can’t *automatically* see the Klass declared inside Manny, but it *can* see it by taking one easy extra step, namely by speaking of `Manny.Klass`. It can do *that* because it *can* see Manny (because Manny is declared at a level that code inside Moe can see).
Modules

The top-level namespaces are *modules*. By default, your app is a module and hence a namespace; that namespace’s name is, roughly speaking, the name of the app. For example, if my app is called *MyApp*, then if I declare a class *Manny* at the top level of a file, that class’s *real* name is *MyApp.Manny*. But I don’t usually need to use that real name, because my code is already inside the same namespace, and can see the name *Manny* directly.

Frameworks are also modules, and hence they are also namespaces. For example, Cocoa’s Foundation framework, where *NSString* lives, is a module. When you program iOS, you will say `import Foundation` (or, more likely, you’ll say `import UIKit`, which itself imports Foundation), thus allowing you to speak of *NSString* without saying `Foundation.NSString`. But you could say `Foundation.NSString`, and if you were so silly as to declare a different *NSString* in your own module, you would have to say `Foundation.NSString`, in order to differentiate them. You can also create your own frameworks, and these, too, will be modules.

Thus, above and beyond the level of the file, as shown in Example 1-1, are any libraries (modules) that the file imports. Your code *always implicitly imports* Swift itself. You could make this explicit by starting a file with the line `import Swift`; there is no need to do this, but it does no harm either.

That fact is important, because it solves a major mystery: where do things like `println` come from, and why is it possible to use them outside of any message to any object? `println` is in fact a function declared at the top level of the `Swift.h` file — which your file can see exactly because it imports Swift. It is thus an ordinary top-level function like any other. You could say things like `Swift.println("hello")`, but you probably never will, because there’s no name conflict to resolve.

You can actually see the `Swift.h` file and read it and study it, and this can be a useful thing to do. To do so, Command-click the term `println` in your code. Alternatively, explicitly `import Swift` and Command-click the term `Swift`. Behold, there it is! You won’t see any executable Swift code here, but you will see the declarations for all the available Swift terms, including top-level functions like `println`, operators like `+`, and declarations of built-in types such as `Int` and `String` (look for `struct Int`, `struct String`, and so on).

Instances

Object types — class, struct, and enum — have an important feature in common: they can be *instantiated*. In effect, when you declare an object type, you are only defining a *type*. To instantiate a type is to make a thing — an *instance* — of that type.
So, for example, I can declare a Dog class:

```swift
class Dog {
}
```

And I can give my class a method:

```swift
class Dog {
    func bark() {
        print("woof")
    }
}
```

But I don't have actually have any Dog objects in my program yet. I have merely described the type of thing a Dog would be if I had one. To get an actual Dog, I have to make one. The process of making an actual Dog object whose type is the Dog class is the process of instantiating Dog. The result is a new object — a Dog instance.

In Swift, instances are created by using the object type's name as a function name and calling the function. This involves using parentheses. When you append parentheses to the name of an object type, you are sending a very special kind of message to that object type: Instantiate yourself! So now I'm going to make a Dog instance:

```swift
let fido = Dog()
```

There's a lot going on in that code! I did two things. I instantiated Dog, thus causing me to end up with a Dog instance. I also put that Dog instance into a shoebox called fido — I declared a variable and initialized the variable by assigning my new Dog instance to it. Now fido is a Dog instance. (Moreover, because I used `let`, fido will always be this same Dog instance. I could have used `var` instead, but even then, initializing fido as a Dog instance would have meant fido could only be some Dog instance after that.)

Now that I have a Dog instance, I can send instance messages to it. And what do you suppose they are? They are Dog's properties and methods! For example:

```swift
let fido = Dog()
fido.bark()
```

That code is legal. Not only that, it is effective: it actually does cause "woof" to appear in the console. I made a Dog and I made it bark! (See Figure 1-1.)

There's an important lesson here, so let me pause to emphasize it. By default, properties and methods are instance properties and methods. You can't use them as messages to the object type itself; you have to have an instance to send those messages to. As things stand, this is illegal and won't compile:

```swift
Dog.bark() // compile error
```

It is possible to declare a function bark in such a way that saying Dog.bark() is legal, but that would be a different kind of function — a class function or a static function — and you would need to say so when you declare it.
The same thing is true of properties. The only respect in which any Dog has had a name up to now has been the name of the variable to which it is assigned. But that name is not intrinsic to the Dog object itself. Let’s give Dog a name property:

```swift
class Dog {
    var name = ""
}
```

That allows me to set a Dog’s name, but it needs to be an instance of Dog:

```swift
let fido = Dog()
fido.name = "Fido"
```

It is possible to declare a property name in such a way that saying `Dog.name` is legal, but that would be a different kind of property — a class property or a static property — and you would need to say so when you declare it.

## Why Instances?

Even if there were no such thing as an instance, an object type is itself an object. We know this because it is possible to send a message to an object type: it is possible to treat an object type as a namespace and to dive explicitly into that namespace (the phrase `Manny.Klass` is a case in point). Moreover, since class and static members exist, it is possible to call a method directly on a class, a struct, or an enum type, and to refer to a property of a class, a struct, or an enum type. Why, then, do instances exist at all?
The answer has mostly to do with the nature of instance properties. The value of an instance property is defined with respect to a particular instance. This is where instances get their real usefulness and power.

Consider again our Dog class. I’ll give it a name property and a bark method; remember, these are an instance property and an instance method:

```swift
class Dog {
    var name = ""
    func bark() {
        println("woof")
    }
}
```

A Dog instance comes into existence with a blank name (an empty string). But its name property is a var, so once we have any Dog instance, we can assign to its name a new String value:

```swift
let dog1 = Dog()
dog1.name = "Fido"
```

We can also ask for a Dog instance’s name:

```swift
let dog1 = Dog()
dog1.name = "Fido"
println(dog1.name) // "Fido"
```

The important thing is that we can make more than one Dog instance, and that two different Dog instances can have two different name property values (Figure 1-2):

```swift
let dog1 = Dog()
dog1.name = "Fido"
let dog2 = Dog()
dog2.name = "Rover"
println(dog1.name) // "Fido"
println(dog2.name) // "Rover"
```

Note that a Dog instance’s name property has nothing to do with the name of the variable to which a Dog instance is assigned. The variable is just a shoebox. You can pass an instance from one shoebox to another. But the instance itself maintains its own internal integrity:

```swift
let dog1 = Dog()
dog1.name = "Fido"
var dog2 = Dog()
dog2.name = "Rover"
println(dog1.name) // "Fido"
println(dog2.name) // "Rover"
dog2 = dog1
println(dog2.name) // "Fido"
```
That code didn’t change Rover’s name; it changed which dog was inside the `dog2` shoe-box, replacing Rover with Fido.

The full power of object-based programming has now emerged. There is a Dog object type which defines *what it is to be a Dog*. Our declaration of Dog says that a Dog instance — *any* Dog instance, *every* Dog instance — has a `name` property and a `bark` method. But *each* Dog instance can have its own `name` property value. They are *different* instances and maintain their own internal *state*. So multiple instances of the same object type *behave* alike — both Fido and Rover can bark, and will do so when they are sent the `bark` message — but they are different instances and can have different property values: Fido’s `name` is "Fido" while Rover’s `name` is "Rover".

(The same thing is true of `1` and `2`, though this fact is somewhat more opaque. An Int has a `value` property. `1` is an Int whose `value` is `1`, and `2` is an Int whose `value` is `2`. However, this fact is of less interest in real life, because obviously you’re not going to change the value of `1`!)

So an instance is a reflection of the instance methods of its type, but that isn’t *all* it is; it’s also a collection of instance properties. The object type is responsible for *what* properties the instance has, but not necessarily for the *values* of those properties. The values can change as the program runs, and apply only to a particular instance. An instance is a cluster of particular property values.

An instance is responsible not only for the values but also for the *lifetimes* of its properties. Suppose we bring a Dog instance into existence and assign to its `name` property
the value "Fido". Then this Dog instance is keeping the string "Fido" alive just so long as we do not replace the value of its name with some other value and just so long as this instance lives.

In short, an instance is both code and data. The code it gets from its type and in a sense is shared with all other instances of that type, but the data belong to it alone. The data can persist as long as the instance persists. The instance has, at every moment, a state — the complete collection of its own personal property values. An instance is a device for maintaining state. It’s a box for storage of data.

**self**

An instance is an object, and an object is the recipient of messages. Thus, an instance needs a way of sending a message to itself. This is made possible by the magic word self. This word can be used wherever the name of an instance is expected (an instance of the appropriate type, that is).

For example, let’s say I want to keep the thing that a Dog says when it barks — namely "woof" — in a property. Then in my implementation of bark I need to refer to that property. I can do it like this:

```swift
class Dog {
    var name = ""
    var whatADogSays = "woof"
    func bark() {
        println(self.whatADogSays)
    }
}
```

Similarly, let’s say I want to write an instance method speak which is merely a synonym for bark. My speak implementation can consist of simply calling my own bark method. I can do it like this:

```swift
class Dog {
    var name = ""
    var whatADogSays = "woof"
    func bark() {
        println(self.whatADogSays)
    }
    func speak() {
        self.bark()
    }
}
```

Observe that the term self in that example appears only in instance methods. When an instance’s code says self, it is referring to this instance. If the expression self.name appears in a Dog instance method’s code, it means the name of this Dog instance, the one whose code is running at that moment.
It turns out that every use of the word `self` I’ve just illustrated is completely optional. You can omit it and all the same things will happen:

```swift
class Dog {
    var name = ""
    var whatADogSays = "woof"
    func bark() {
        println(whatADogSays)
    }
    func speak() {
        bark()
    }
}
```

The reason is that if you omit the message recipient and the message you’re sending can be sent to `self`, the compiler supplies `self` as the message’s recipient under the hood. However, I never do that (except by mistake). As a matter of style, I like to be explicit in my use of `self`. I find code that omits `self` harder to read and understand. And there are situations where you must say `self`, so I prefer to use it whenever I’m allowed to use it.

### Privacy

Earlier, I said that a namespace is not, of itself, an insuperable barrier to accessing the names inside it. But it can act as a barrier if you want it to. For example, not all data stored by an instance is intended for alteration by, or even visibility to, another instance. And not every instance method is intended to be called by other instances. Any decent object-based programming language needs a way to endow its object members with privacy — a way of making it harder for other objects to see those members if they are not supposed to be seen.

Consider, for example:

```swift
class Dog {
    var name = ""
    var whatADogSays = "woof"
    func bark() {
        println(self.whatADogSays)
    }
    func speak() {
        println(self.whatADogSays)
    }
}
```

Here, other objects can come along and change my property `whatADogSays`. Since that property is used by both `bark` and `speak`, we could easily end up with a Dog that, when told to `bark`, says "meow". This seems somehow undesirable:
let dog1 = Dog()
dog1.whatADogSays = "meow"
dog1.bark() // meow

You might reply: Well, silly, why did you declare whatADogSays with var? Declare it with let instead. Make it a constant! Now no one can change it:

```swift
class Dog {
    var name = ""
    let whatADogSays = "woof"
    func bark() {
        println(self.whatADogSays)
    }
    func speak() {
        println(self.whatADogSays)
    }
}
```

That is a good answer, but it is not quite good enough. There are two problems. Suppose I want a Dog instance itself to be able to change self.whatADogSays. Then whatADogSays has to be a var; otherwise, even the instance itself can't change it. Also, suppose I don't want any other object to know what this Dog says, except by calling bark or speak. Even when declared with let, other objects can still read the value of whatADogSays. Maybe I don't like that.

To solve this problem, Swift provides the private keyword. I’ll talk later about all the ramifications of this keyword, but for now it’s enough to know that it solves the problem:

```swift
class Dog {
    var name = ""
    private var whatADogSays = "woof"
    func bark() {
        println(self.whatADogSays)
    }
    func speak() {
        println(self.whatADogSays)
    }
}
```

Now name is a public property, but whatADogSays is a private property: it can't be seen by other objects. A Dog instance can speak of self.whatADogSays, but a different object with a reference to a Dog instance as, say, dog1 cannot say dog1.whatADogSays.

The important lesson here is that object members are public by default, and if you want privacy, you have to ask for it. The class declaration defines a namespace; this namespace requires that other objects use an extra level of dot-notation to refer to what's inside the namespace, but other objects can still refer to what's inside the namespace; the namespace does not, in and of itself, close any doors of visibility. The private keyword lets you close those doors.
Design

You now know what an object is, and what an instance is. But what object types will your program need, what methods and properties should they have, when and how will they be instantiated, and what should you do with those instances when you have them? Unfortunately I can’t tell you that; it’s an art — the art of object-based programming. What I can tell you is what your chief considerations are going to be as you design and implement an object-based program — the process that I call growing a program.

Object-based program design must be founded upon a secure understanding of the nature of objects. You want to design object types that encapsulate the right sort of functionality (methods) accompanied by the right set of data (properties). Then, when you instantiate those object types, you want to make sure that your instances have the right lifetimes, sufficient exposure to one another, and an appropriate ability to communicate with one another.

Object Types and APIs

Your program files will have very few, if any, top-level functions and variables. Methods and properties of object types — in particular, instance methods and instance properties — will be where most of the action is. Object types give each actual instance its specialized abilities. They also help to organize your program’s code meaningfully and maintainably.

We may summarize the nature of objects in two phrases: encapsulation of functionality, and maintenance of state. (I first used this summary many years ago in my book REALbasic: The Definitive Guide.)

Encapsulation of functionality

Each object does its own job, and presents to the rest of the world — to other objects, and indeed in a sense to the programmer — an opaque wall whose only entrances are the methods to which it promises to respond and the actions it promises to perform when the corresponding messages are sent to it. The details of how, behind the scenes, it actually implements those actions are secreted within itself; no other object needs to know them.

Maintenance of state

Each individual instance is a bundle of data that it maintains. Often that data is private, which means that it’s encapsulated as well; no other object knows what that data is or in what form it is kept. The only way to discover from outside what data an object is maintaining is if there’s a method or public property that reveals it.

As an example, imagine an object whose job is to implement a stack — it might be an instance of a Stack class. A stack is a data structure that maintains a set of data in LIFO order (last in, first out). It responds to just two messages: push and pop. Push means to
add a given piece of data to the set. Pop means to remove from the set the piece of data that was most recently pushed and hand it out. It's like a stack of plates: plates are placed onto the top of the stack or removed from the top of the stack one by one, so the first plate to go onto the stack can't be retrieved until all other subsequently added plates have been removed (Figure 1-3).

The stack object illustrates encapsulation of functionality because the outside world knows nothing of how the stack is actually implemented. It might be an array, it might be a linked list, it might be any of a number of other implementations. But a client object — an object that actually sends a push or pop message to the stack object — knows nothing of this and cares less, provided the stack object adheres to its contract of behaving like a stack. This is also good for the programmer, who can, as the program develops, safely substitute one implementation for another without harming the vast machinery of the program as a whole. And just the other way round, the stack object
knows nothing and cares less about who is telling it to push or to pop, and why. It just hums along and does its job in its reliable little way.

The stack object illustrates maintenance of state because it isn’t just the gateway to the stack data — it is the stack data. Other objects can get access to that data, but only by virtue of having access to the stack object itself, and only in the manner that the stack object permits. The stack data is effectively inside the stack object; no one else can see it. All that another object can do is push or pop. If a certain object is at the top of our stack object’s stack right now, then whatever object sends the pop message to this stack object will receive that object in return. If no object sends the pop message to this stack object, then the object at the top of the stack will just sit there, waiting.

The sum total of messages that each object type is eligible to be sent by other objects — its API (application programming interface) — is like a list or menu of things you can ask this type of object to do. Your object types divide up your code; their APIs form the basis of communication between those divisions.

In real life, when you’re programming iOS, the vast majority of object types you’ll be working with will not be yours but Apple’s. Swift itself comes with a few useful object types, such as String and Int; you’ll also import UIKit, which includes a huge number of object types, all of which spring to life in your program. You didn’t create any of those object types, so to learn to use them, you consult the published APIs, also known as the documentation. Apple’s own Cocoa documentation consists largely of pages where each page lists and describes the properties and methods supplied by one object type. For example, to know what messages you can send to an NSString instance, you’d start by studying the NSString class documentation. That page is really just a big list of properties and methods, so it tells you what an NSString object can do. That isn’t everything in the world there is to know about an NSString, but it’s a big percentage of it.

Thus, in real life, the “wise programmer” of whom I spoke a moment ago will be, in large part, Apple. Your wisdom will lie not in creating new object types but in using the object types that Apple has already given you. You can also create new object types, and you will do so, but proportionately you will do this vastly less than you will use the object types that exist already.

**Instance Creation, Scope, and Lifetime**

The important moment-to-moment entities in a Swift program are mostly instances. Object types define what kinds of instances there can be and how each kind of instance behaves. But the actual instances of those types are the state-carrying individual “things” that the program is all about, and instance methods and properties are messages that can be sent to instances. So there need to be instances in order for the program to do anything.
By default, however, there are no instances! Looking back at Example 1-1, we defined some object types, but we made no instances of them. If we were to run this program, our object types would exist from the get-go, but that’s all that would exist. We’ve created a world of pure potentiality — some types of object that might exist. In that world, nothing would actually happen.

Instances do not come into being by magic. You have to instantiate a type in order to obtain an instance. Much of the action of your program, therefore, will consist of instantiating types. And of course you will want those instances to persist, so you will also assign each newly created instance to a variable as a shoebox to hold it, name it, and give it a lifetime. The instance will persist according to the lifetime of the variable that refers to it. And the instance will be visible to other instances according to the scope of the variable that refers to it.

Much of the art of object-based programming turns out to be exactly here, in giving instances a sufficient lifetime and making them visible to one another. You will often put an instance into a particular shoebox — assigning it to a particular variable, declared at a certain scope — exactly so that, thanks to the rules of variable lifetime and scope, this instance will persist long enough to keep being useful to your program while it will still be needed, and so that other code can get a reference to this instance and talk to it later.

Planning how you’re going create instances, and working out the lifetimes and communication between those instances, may sound daunting. Fortunately, in real life, when you’re programming iOS, the Cocoa framework itself will once again provide an initial scaffolding for you.

For example, you’ll know from the start that, for an iOS app, you need an app delegate type and a view controller type, and in fact when you create an iOS app project, Xcode will give them to you. Moreover, as your app launches, the runtime will instantiate those object types for you, and will place those instances into a fixed and useful relationship. The runtime will make an app delegate instance and assign it in such a way that it lives for the lifetime of the app; it will create a window instance and assign it to a property of the app delegate; and it will create a view controller instance and assign it to a property of the window. Finally, the view controller instance has a view, which automatically appears in the window.

Thus, without your doing any work at all, you already have some objects that will persist for the lifetime of the app, including one that is the basis of your visible interface. Just as important, you have well-defined globally available ways of referring to all these objects. This means that, without writing any code, you already have access to some important objects, and you have an initial place to put any other objects with long lifetimes and any other visible bits of interface that your app may need.
Summary and Conclusion

As we imagine constructing an object-based program for performing a particular task, we bear in mind the nature of objects. There are types and instances. A type is a set of methods describing what all instances of that type can do (encapsulation of functionality). Instances of the same type differ only in the value of their properties (maintenance of state). We plan accordingly. Objects are an organizational tool, a set of boxes for encapsulating the code that accomplishes a particular task. They are also a conceptual tool. The programmer, being forced to think in terms of discrete objects, must divide the goals and behaviors of the program into discrete tasks, each task being assigned to an appropriate object.

At the same time, no object is an island. Objects can cooperate with one another, namely by communicating with one another — that is, by sending messages to one another. The ways in which appropriate lines of communication can be arranged are innumerable. Coming up with an appropriate arrangement — an architecture — for the cooperative and orderly relationship between objects is one of the most challenging aspects of object-based programming. In iOS programming, you get a boost from the Cocoa framework, which provides an initial set of object types and a practical basic architectural scaffolding.

Using object-based programming effectively to make a program do what you want it to do while keeping it clear and maintainable is itself an art; your abilities will improve with experience. Eventually, you may want to do some further reading on effective planning and construction of the architecture of an object-based program. I recommend in particular two classic, favorite books. Refactoring, by Martin Fowler (Addison-Wesley, 1999), describes why you might need to rearrange what methods belong to what classes (and how to conquer your fear of doing so). Design Patterns, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (also known as “the Gang of Four”), is the bible on architecting object-based programs, listing all the ways you can arrange objects with the right powers and the right knowledge of one another (Addison-Wesley, 1994).
When you buy an ebook through oreilly.com you get lifetime access to the book, and whenever possible we provide it to you in four DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, and DAISY—that you can use on the devices of your choice. Our ebook files are fully searchable, and you can cut-and-paste and print them. We also alert you when we’ve updated the files with corrections and additions.

Learn more at ebooks.oreilly.com

You can also purchase O’Reilly ebooks through the iBookstore, the Android Marketplace, and Amazon.com.